

ARMY RESEARCH LABORATORY



**A Programmer's Guide to the Bounding Overwatch
Behavior Software**

by MaryAnne Fields

ARL-MR-589

June 2004

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

Army Research Laboratory

Aberdeen Proving Ground, MD 21005-5066

ARL-MR-589

June 2004

A Programmer's Guide to the Bounding Overwatch Behavior Software

MaryAnne Fields
Weapons and Materials Research Directorate, ARL

Report Documentation Page

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE (DD-MM-YYYY) June 2004		2. REPORT TYPE Final		3. DATES COVERED (From - To) November 2003–January 2004	
4. TITLE AND SUBTITLE A Programmer's Guide to the Bounding Overwatch Behavior Software				5a. CONTRACT NUMBER	
				5b. GRANT NUMBER	
				5c. PROGRAM ELEMENT NUMBER	
6. AUTHOR(S) MaryAnne Fields				5d. PROJECT NUMBER AH03	
				5e. TASK NUMBER	
				5f. WORK UNIT NUMBER	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) U.S. Army Research Laboratory ATTN: AMSRD-ARL-WM-BF Aberdeen Proving Ground, MD 21005-5066				8. PERFORMING ORGANIZATION REPORT NUMBER ARL-MR-589	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)				10. SPONSOR/MONITOR'S ACRONYM(S)	
				11. SPONSOR/MONITOR'S REPORT NUMBER(S)	
12. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.					
13. SUPPLEMENTARY NOTES					
14. ABSTRACT This report is a programmer's guide to the software developed for the BoundingMovement behavior implemented on the ATRV-Jr platforms. The BoundingMovement algorithm is documented, and a detailed example is provided for other researchers trying to develop computer programs for the iRobot platforms. An overview of the behavior algorithm, details on the computer code developed to implement the algorithm, and a discussion of future research are also provided.					
15. SUBJECT TERMS robot, behavior algorithms, bounding overwatch					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON MaryAnne Fields
a. REPORT UNCLASSIFIED	b. ABSTRACT UNCLASSIFIED	c. THIS PAGE UNCLASSIFIED			19b. TELEPHONE NUMBER (Include area code) 410-278-6675

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18

Contents

List of Figures	v
List of Tables	v
Acknowledgments	vi
1. Introduction	1
2. Servers	4
2.1 Information Server	4
2.2 Map Server	4
3. BoundingMovement Functions	5
3.1 Image Processing.....	5
3.1.1 GetImage	5
3.1.2 ClassifyPixel.....	6
3.1.3 IsPixelYellow	6
3.1.4 FindWayPoint.....	7
3.1.5 GetWayPointBearing	8
3.1.6 FindNearestWall.....	8
3.1.7 KeepEyesOnWaypoint	9
3.1.8 LookForMovement.....	10
3.1.9 FollowMovement	10
3.2 Sonar Processing	11
3.2.1 CheckSonar	11
3.2.2 Process_Sonar	12
3.3 Movement.....	13
3.3.1 OrientVehicle	13
3.3.2 RotateIntoWall	13
3.3.3 MoveToWaypoint	14
3.3.4 MoveToWall	14
3.3.5 MoveIntoOpen.....	15
3.3.6 BoundForward.....	15

3.4	Communication	16
3.4.1	PublishMessage	16
3.4.2	GetMessage	16
3.4.3	ParseMessage	17
3.5	Miscellaneous	18
3.5.1	StartServers	18
3.5.2	PrintImage	18
3.5.3	PrintYellowImage	19
4.	Conclusions	20
	Appendix. Defined Constants for the BoundingMovement Code	21
	Distribution List	23

List of Figures

Figure 1. The state diagram for the BoundingMovement behavior.....	2
Figure 2. The process flow diagram for the Move state.	3
Figure 3. The process flow diagram for the Watch state.	3

List of Tables

Table A-1. List of the defined constants used in the BoundingMovement software.....	22
---	----

Acknowledgments

The author would like to acknowledge MyVan H. Baranoski for writing the ParseMessage function and Richard J. Pearson for writing the ProcessSonar function.

1. Introduction

One of the goals of the U.S. Army Ground Robotics Research Program is to develop individual and group behaviors that allow the robot to contribute to battlefield missions such as reconnaissance. As a part of this research program at the Weapons Technology Analysis Branch, we have developed a behavior to demonstrate aspects of a bounding overwatch maneuver. The behavior is a cooperative mission, with one robot acting as a stationary observer, while the other robot bounds to its next destination. The robots alternate roles until the mission is complete. The behavior was developed in simulation using One Semi-Automated Forces (OneSAF). This allowed us to develop an algorithm that was not tied to a specific robotic hardware configuration. Two ATRV-Jr* robots from iRobot, Inc. were selected to demonstrate the behavior as surrogate robotic platforms for the experimental unmanned vehicle (XUV). The robots are four-wheeled, skid-steered platforms that can be used indoors and outdoors. The ATRV-Jr's sensors include visible spectrum cameras, ultrasonic range sensor array, GPS, an inertial measurement unit, a compass, and a tilt sensor. All sensor data analysis and mobility control is performed by a single on-board processor.

We have implemented some simplifications in our demonstration scenario and environment that enable us to experiment with the behavior indoors using laboratory robots with limited sensor capabilities and processing power. First, due to the inability to navigate GPS waypoints indoors, the endpoint is a visual beacon that the robots can use to orient themselves as they perform the behavior. Secondly, yellow walls, easily identified by the robots, simulate concealed locations along the way to the endpoint. The planned addition of a single line laser radar (LADAR) will make the indoor navigation more robust. These simplifications allow us to focus on the algorithm and not be distracted by the integration of additional sensors or processors.

This document is a programmer's guide to the software developed for the BoundingMovement behavior implemented on the ATRV-Jr platforms. This guide documents the BoundingMovement algorithm and provides a detailed example for other researchers trying to develop computer programs for the iRobot platforms.

The remainder of this section is an overview of the behavior algorithm. We describe the behavior in terms of a state diagram and a process flow diagram. Sections 2 and 3 provide details on the computer code developed to implement the algorithm. The last section is a discussion of planned experiments for the bounding overwatch maneuver.

Figure 1 shows a state diagram of the bounding movement behavior. There are four states: START, BOUND FORWARD, WATCH, and END. These are shown as ovals in the diagrams.

* ATRV-Jr is a trademark of iRobot, Inc.

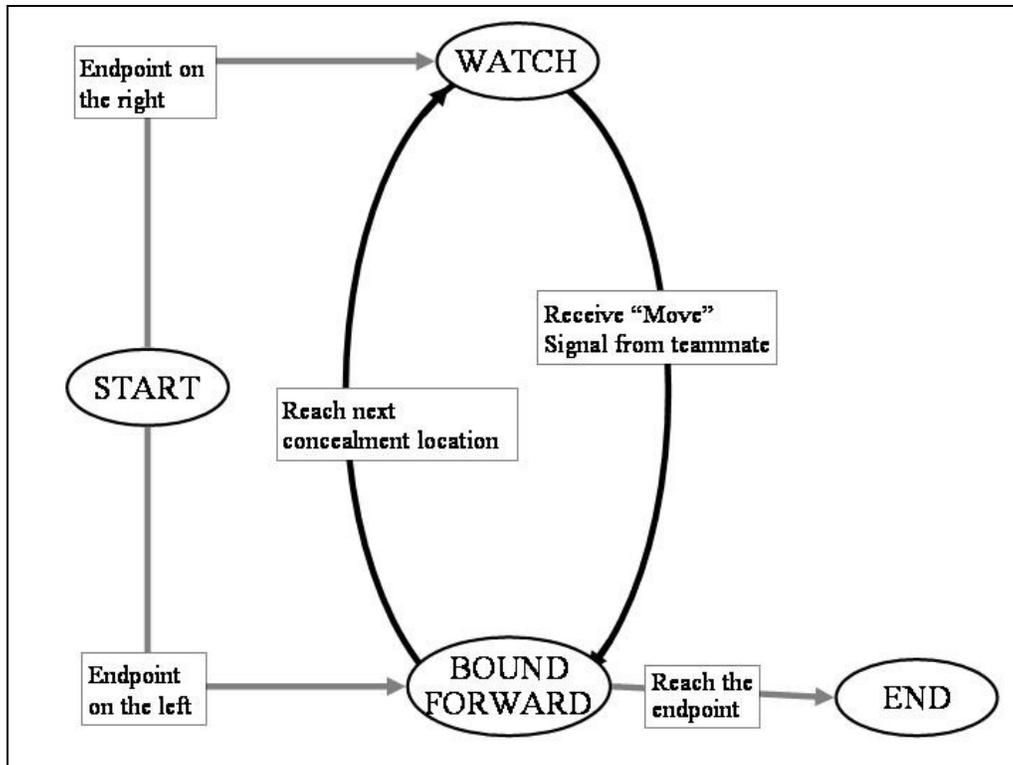


Figure 1. The state diagram for the BoundingMovement behavior.

The dashed boxes give the condition for each transition. The START state transitions to either the BOUND FORWARD or WATCH state, depending on the position of the robot with respect to the endpoint. Transitions from the BOUND FORWARD to the WATCH state occur when the robot reaches a wall. Transitions from the BOUND FORWARD to the END state occur when the robot reaches the endpoint. In this behavior algorithm, a robot does not transition from the WATCH state to the BOUND FORWARD state on its own; it must receive a message from its companion indicating that it is time to switch states.

Figure 2 shows an overview process flow diagram for the BOUND FORWARD state. In the diagram, activities executed serially like “MoveIntoOpen” and “FindNearestWall” are connected with straight lines. Activities executed in parallel, such as “MoveToWall” and “GetMessage” are connected by diamonds. In the setup phase, the robot moves into the open to prepare for its next move. It determines its next destination using its cameras to find the nearest wall if one exists. The solid lines show the algorithm that the robot uses to move to a wall. The dashed lines show the algorithm used to travel to the endpoint. Once the robot begins moving, it uses its cameras to provide steering information for driving system and its sonar array to determine distance to nearby obstacles. During this phase, the robot also monitors its message queue for a danger signal published by its companion. A danger signal causes the robot to speed up so that it reaches a place of cover quickly. In the wrap-up phase, the robot assumes a watching position and signals its companion that it is time to switch states.

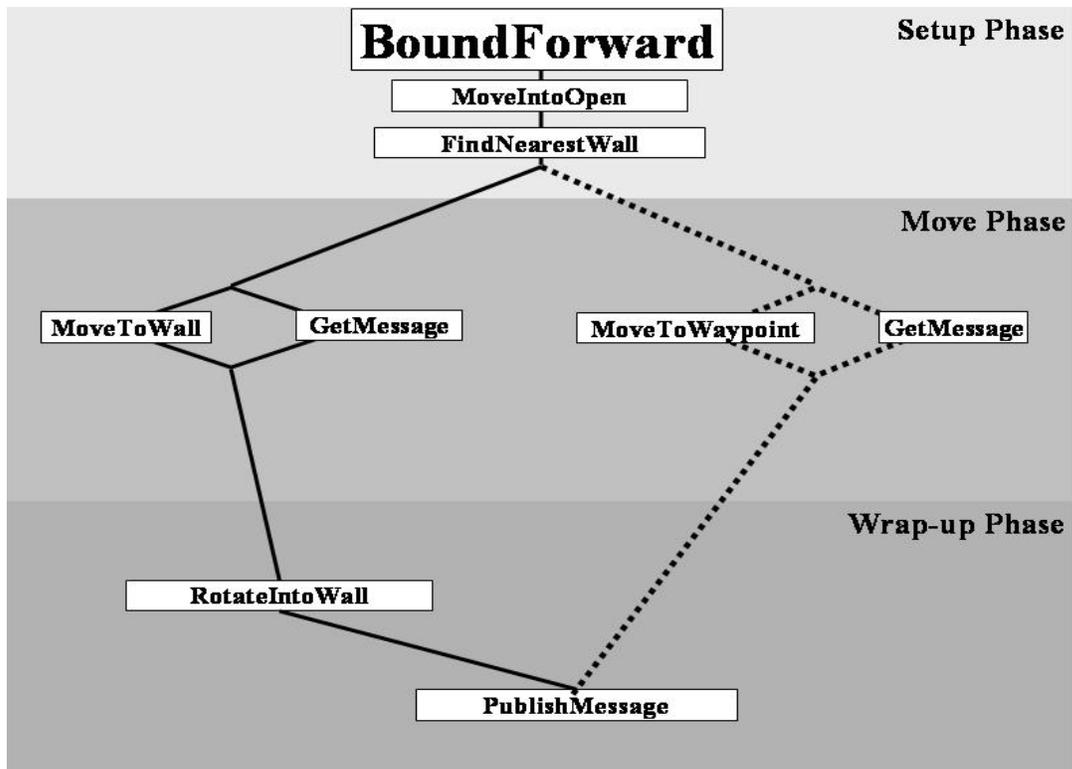


Figure 2. The process flow diagram for the Move state.

Figure 3 shows a flowchart for the WATCH state. Just as in figure 2, lines link activities that are executed serially, while diamonds link activities that are executed in parallel. In this state, the robot's camera mount is oriented toward the endpoint and the images are analyzed for evidence of moving objects. If the robot detects movement, it sends a danger signal to its companion. The robot continues to watch until it receives a move signal from its companion.

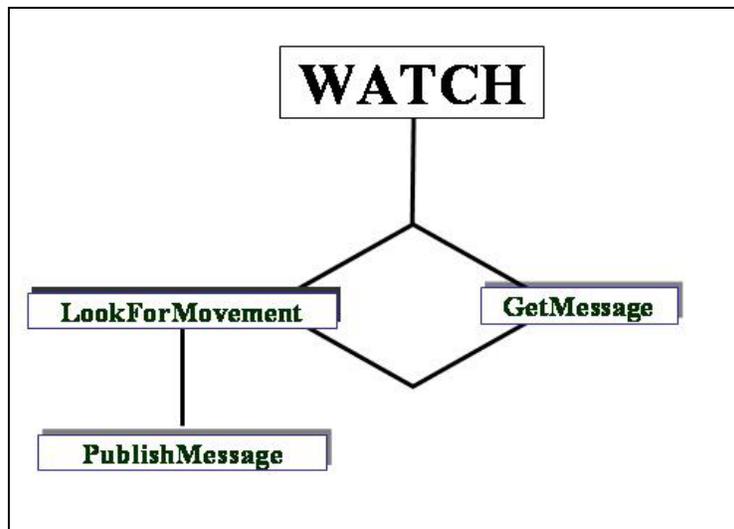


Figure 3. The process flow diagram for the Watch state.

2. Servers

A server is an independent process that provides information and control for systems available to the robots. The Mobility* software package provides servers that interface to sensors and actuator systems such as the drive motors or the pan-tilt unit. In this section, we describe two servers that provide useful information for the robots. The first server handles communication between the two robots. The second server maintains a shared obstacle map.

2.1 Information Server

The information server allows the robots to communicate with each other. The server is an object derived from the `ActiveSystemComponent` class of the Mobility software library. The `BoundingMovement` program accesses the server by linking variables within the server object to local variables. For this software project, there are two variables of interest: `R4-Message` maintains messages from the R4 robot and `R5-Message` maintains messages from the R5 robot. Each of these objects consist of an 80-character string variable and timestamp that provides the time of message generation, in nanoseconds, and a message number. Messages are placed on the server, or published, using the `new_sample` method for the information server class. This makes the messages available to programs running on the robot or other computers within the local area network. Messages are read by programs using `update_sample` method for the information server class. Both of these methods were inherited from the `ActiveSystemComponent` class and are used frequently to pass information to and from many servers included in the Mobility software package.

This server implements a broadcast strategy. A robot only publishes messages on its specific message line. Other robots must monitor that message line for new information. Note that this strategy does not guarantee message delivery.

The server does not control message content. The applications, such as the `BoundingMovement` behavior, determine the message set. The message set for this behavior is described in section 3.4.2.

2.2 Map Server

The map server maintains a shared map for up to five robots. It collects two types of information, position information and obstacle information. The server is an object derived from the `ActiveSystemComponent` class of the Mobility software library. The server constructs the obstacle map from three types of information; position data, sonar data, and laser data. The position data gives the location of the robot on the map. Sensors such as the odometry sensor or the Global Positioning System (GPS) sensor supply position information. The sonar data and the

* Mobility is a trademark of iRobot, Inc.

laser data give the location of obstacles relative to the robot. The map server uses both sources of information to construct an obstacle map showing the location of walls and other obstructions.

In the BoundingMovement behavior, the map server is used as a diagnostic tool that allows the researcher to visualize information from several sensors at one time. As we continue to develop the map server, we intend to include *a priori* knowledge and use the map as an information source for robotic planning algorithms.

3. BoundingMovement Functions

This section describes the C++ functions developed for the BoundingMovement behavior. The functions are grouped by category: Image processing, Sonar processing, Movement, Communication, and Miscellaneous. Within each category, functions are described in order of complexity with the least complex functions described first. Each function description contains five sections: Function Prototype, Description, Input Variables, Output Variables, and Return Value. The Function Prototype provides the call syntax that gives the argument list and the return type. The arguments are described in the Input Variable and Output Variable section. The Description section provides a short description of each function. Most of the functions return status information; defined constants with descriptive names are more useful as status information than the actual integer value of the constant. This section provides the defined constant names and their interpretation. The appendix provides a table of defined constants and their numerical values.

3.1 Image Processing

3.1.1 GetImage

Function Prototype:

- `int GetImage(int CameraNumber, double period).`

Description:

- *GetImage* – updates the stored image array from the camera specified by the CameraNumber variable. The period variable determines how often *new* images are retrieved. This allows calling routines flexibility in using images; the *GetImage* routine can be called from inside a high-frequency loop, such as a driving loop, without requiring new images to be generated at the same frequency.

Input Variables:

- CameraNumber – integer variable specifying the desired camera.

- period – double-length floating point number that sets the image retrieval period in seconds.

Possible Return Values:

- YES – a new image has been generated.
- NO – no new image is available.

3.1.2 ClassifyPixel

Function Prototype:

- int ClassifyPixel(int red, int green, and int blue).

Description:

- *ClassifyPixel* determines the color of a pixel based on the RGB color scale. Possible return values are red, blue, or neutral.

Input Variables:

- red, green, and blue integer variables describing the RGB color of the pixel.

Possible Return Values:

- RED – the pixel is red.
- BLUE – the pixel is blue.
- NEUTRAL – the pixel is not red or blue.

3.1.3 IsPixelYellow

Function Prototype:

- int IsPixelYellow(int red, int green, and int blue).

Description:

- IsPixelYellow determines the color of the pixel using the RGB color scale. The routine returns the integer constant YES if the pixel is yellow, NO otherwise.

Input Variables:

- red, green, and blue integer variables describing the RGB color of the pixel.

Possible Return Values:

- YES – the pixel is yellow.
- NO – the pixel is not yellow.

3.1.4 FindWayPoint

Function Prototype:

- int *FindWayPoint*(int ImageNumber, int CameraNumber, int DisplayPicture, float period, int DesiredWayPointLocation, int *cx, int *cy, int *cbx, int *cby, int *TotalHits, and int *MaxBinHits).

Description:

- *FindWayPoint* determines the location of the endpoint using the CameraNumber camera. It makes two simultaneous estimations of the position: one using the entire image and one using 20×20 image bins. This routine calls *ClassifyPixel* to determine the color of each pixel (red, blue, or neutral). A pixel is considered a candidate pixel if it is part of a red/blue checkerboard pattern. The routine returns two estimates: one based on the position the number of candidate points for the entire image and the other based on the higher number of candidate points for any single bin.

Input Variables:

- ImageNumber – integer variable giving the image number that is used to tag stored images.
- CameraNumber – an integer variable specifying the desired camera.
- DisplayPicture – an integer flag which determines if the system uses xv to show images while the behavior is running.
- Period – double-length floating point number that sets the image retrieval period.
- DesiredWayPointLocation – desired x location of the endpoint in the image plane. This variable is passed to the graphics routine, *PrintImage* (described in the following), to draw a vertical reference line on the image.

Output Variables:

- cx and cy – location, in the image plane, of the endpoint using the entire image to estimate location.
- cbx and cby – location, in the image plane, of the endpoint using the image bin with the highest number of candidate points to estimate the location.
- TotalHits – total number of candidate points.
- MaxBinHits – largest number of candidate points within a single image bin.

Possible Return Values:

- FoundWayPoint – indicates that there are enough candidate pixels to identify the endpoint in the image.
- NotEnoughPixels – there are too few candidate pixels to identify the endpoint in the image.

3.1.5 GetWayPointBearing

Function Prototype:

- int GetWayPointBearing(int CameraNumber, double *localbearing, and double *bearing).

Description:

- *GetWayPointBearing* gives the bearing to the endpoint. The routine uses the compass to determine the heading of the vehicle. It contains a while-loop structure that uses the *FindWayPoint* function to find the endpoint in the camera image and continually adjusts the camera pan angle until the endpoint is centered in the image. The camera pan angle and the vehicle heading are added to produce the bearing to endpoint.

Input Variables:

- CameraNumber – integer variable specifying the desired camera.

Output Variables:

- localbearing – double-length floating point number giving the vehicle centric bearing to the endpoint based on the orientation of the pan tilt unit.
- bearing – double-length floating point number giving the absolute bearing to the endpoint based on the compass reading and the orientation of the pan tilt unit.

Possible Return Values:

- NotEnoughPixels – the endpoint is not visible in the image.
- FoundWaypoint – the endpoint is visible in image.

3.1.6 FindNearestWall

Function Prototype:

- int *FindNearestWall*(int ImageNumber, int CameraNumber, int LoX, int HiX, int WayPointLocation, int DesiredWallLocation, int *cx, int *cy, and double *PerCentCoverage).

Description:

- *FindNearestWall* finds the nearest yellow wall in the rectangular region of the image plane bounded by LoX and HiX. Typically, one boundary is set to the current location of the endpoint, and the other boundary is set to the appropriate edge of the image plane. The image plane is divided into thin rectangular cells (5 pixels wide × 40 pixels high). Cells with more than 40 yellow pixels (using *IsPixelYellow* to classify the pixels) are considered part of a wall. *FindNearestWall* reports the location of the wall cell nearest to the endpoint.

Input Variables:

- ImageNumber – integer variable giving the image number that is used to tag stored images.
- CameraNumber – an integer variable specifying the desired camera.
- LoX, HiX – integer variables specifying the boundaries of the search region in the image plane.
- WayPointLocation – x-position of the endpoint in the image plane.
- DesiredWallLocation – desired x-position of the wall in the image plane.

Output Variables:

- cx and cy – location in the image plane of the point on the wall closest to the endpoint.
- PerCentCoverage – the percent of the image plane that is classified as yellow.

Possible Return Values:

- NotEnoughPixels – there are not enough yellow pixels in the image to identify a wall.
- EnoughPixels – there are enough yellow pixels in the image to identify a wall.

3.1.7 KeepEyesOnWaypoint

Function Prototype:

- int *KeepEyesOnWayPoint*(int CameraNumber).

Description:

- *KeepEyesOnWayPoint* attempts to center the endpoint in the image plane by panning the camera. The function uses *FindWayPoint* to determine the location of the endpoint. The *KeepEyesOnWayPoint* routine pans the cameras so that the endpoint is driven towards the center of the image.

Input Variables:

- CameraNumber – integer variable specifying the desired camera.

Possible Return Values:

- NotEnoughPixels – endpoint is not visible in the image.
- FoundWaypoint – endpoint is visible in image.

3.1.8 LookForMovement

Function Prototype

- int *LookForMovement*(int ImageNumber, int CameraNumber, float *AverageDiff, float *BinAverageDiff, int *bx, and int *by).

Description:

- *LookForMovement* looks for evidence of movement in successive images taken from the same camera. For each pixel in the image plane, the routine computes a “color distance,”

$$D_{\text{color}} = \sqrt{(R_p - R_c)^2 + (G_p - G_c)^2 + (B_p - B_c)^2}, \quad (1)$$

where R_c , G_c , and B_c are the colors of the pixel in the current image and R_p , G_p , and B_p are the colors of the pixel in the previous image. Large D_{color} values can indicate movement (large values can also indicate shadowing or other changes in lighting). This routine classifies pixels with large D_{color} as “moving” pixels. These pixels are used to compute the average D_{color} value and the centroid of the “moving” pixels.

Input Variables:

- ImageNumber – integer variable giving the image number that is used to tag stored images.
- CameraNumber – an integer variable specifying the desired camera.

Output Variables:

- AverageDiff – the average color distance.
- BinAverageDiff – the largest average color difference for a 20×20 bin.
- bx, by – the location, in the image plane, of the centroid of changed pixels based on the entire image.

Possible Return Values:

- total number of pixels that have changed in the image.

3.1.9 FollowMovement

Function Prototype:

- void *FollowMovement*(int ImageNumber, int CameraNumber).

Description:

- *FollowMovement* attempts to track movement in the image. The function contains a while-loop structure that uses *LookForMovement* to determine centroid of the moving pixels. For each iteration of the loop, the *FollowMovement* routine pans the cameras so that the centroid of moving pixels is driven towards the center of the image. The function stops when it can no longer detect movement.

Input Variables:

- ImageNumber – integer variable giving the image number that is used to tag stored images.
- CameraNumber – an integer variable specifying the desired camera.

Possible Return Values:

- None.

3.2 Sonar Processing

3.2.1 CheckSonar

Function Prototype:

- int CheckSonar(double *FrontDist,double *LeftDist,double and *RightDist,double *RearDist,int *BestDirection).

Description:

- *CheckSonar* grabs the most current set of sonar readings to determine distance to nearby obstacles. The current set of distance readings are given in inches and are stored in Dist array. *CheckSonar* divides the 17 sonar's into 4 sets, front (sonar's 6, 7, 8, 9, and 10), rear (sonar's 0 and 16), left (sonar's 1, 2, 3, 4, and 5), and right (sonar's 11, 12, 13, 14, and 15). It returns the smallest distance for each of these sets.

Output Variables:

- FrontDist – double-length floating point number giving the closest obstacle distance to the front of the vehicle.
- RightDist – double-length floating point number giving the closest obstacle distance to the right side of the vehicle.
- LeftDist – double-length floating point number giving the closest obstacle distance to the left side of the vehicle.
- RearDist – double-length floating point number giving the closest obstacle distance to the rear of the vehicle.

- BestDirection – double-length floating point number giving the best direction to move to avoid nearby obstacles.

Possible Return Values:

- SAFE – there are no objects within 15 in of the robot.
- TooCloseFront – there is an object within 15 in of the front of the robot.
- TooCloseLeft – there is an object within 15 in of the left side of the robot.
- TooCloseRight – there is an object within 15 in of the right side of the robot.
- TooCloseRear – there is an object within 15 in of the rear of the robot.

3.2.2 Process_Sonar

Function Prototype:

- void *Process_Sonar* (int NumberFront, int *front, int NumberRight, int *right, int NumberLeft, int *left, int NumberRear, int *rear, int PairedData, int DeadRecon, double Time, double *DDist, double *Dist_Dot, double *FrontDist, double *LeftDist, double *RightDist, and double *RearDist).

Description:

- *Process_Sonar* processes input from the robots sonar sensors. It reports the overall minimum distance to nearby obstacles as well as a minimum distance for the front, rear, right, and left set of sensors. The routine maintains an array of the last five sonar samples which it uses to calculate smoothed sonar readings for each of the four sides.

Input Variables:

- NumberFront, NumberRight, NumberLeft, NumberRear - integer variable specifying the number of sonars on each side.
- front, right, left, rear – integer array giving the sonar number for each sonar on the given side.

Output Variables:

- FrontDist – double-length floating point number giving the closest obstacle distance to the front of the vehicle.
- RightDist – double-length floating point number giving the closest obstacle distance to the right side of the vehicle.
- LeftDist – double-length floating point number giving the closest obstacle distance to the left side of the vehicle.

- RearDist – double-length floating point number giving the closest obstacle distance to the rear of the vehicle.

Possible Return Values:

- None.

3.3 Movement

3.3.1 OrientVehicle

Function Prototype:

- `int OrientVehicle(int CameraNumber).`

Description:

- *OrientVehicle* rotates the robot so that it is facing the endpoint. It contains a while-loop structure that adjusts the angular velocity to drive the endpoint towards the center of the camera image. The function terminates when the endpoint is centered in the image.

Input Variables:

- CameraNumber – an integer variable designating which camera to use for image processing. Possible values are 0, indicating the camera on the left, and 1, indicating the camera on the right.

Possible Return Values:

- 0 – meaningless.

3.3.2 RotateIntoWall

Function Prototype:

- `int RotateIntoWall(int WayPointSide).`

Description:

- *RotateIntoWall* rotates the robot so that it is approximately parallel to the wall. The function contains a while-loop structure that uses *CheckSonar* to stop the rotation of the robot. At the same time, the routine tries to maintain visual contact with the endpoint using the routine *KeepEyesOnWayPoint*. The direction of rotation depends on the value of *WayPointSide*. If *WayPointSide* is POSITIVE, the rotation is clockwise; if *WayPointSide* is NEGATIVE, the rotation is counterclockwise.

Input Variables:

- *WayPointSide* – an integer variable designating the desired location of the endpoint. Possible values are POSITIVE, indicating that the robot should keep the endpoint on its right as it drives, and NEGATIVE indicating that the robot should keep the endpoint on its left as it drives forward.

Possible Return Values:

- 0 – meaningless.

3.3.3 MoveToWaypoint

Function Prototype:

- `int MoveToWaypoint(int WayPointSide).`

Description:

- *MoveToWaypoint* moves the robot to the endpoint and ends the mission. The function contains a while-loop structure that drives the robot towards the endpoint using *FindWayPoint* to adjust the angular velocity and *CheckSonar* to adjust the forward velocity.

Input Variables:

- *WayPointSide* – an integer variable designating the desired location of the endpoint. Possible values are POSITIVE, indicating that the robot should keep the endpoint on its right as it drives, and NEGATIVE, indicating that the robot should keep the endpoint on its left as it drives forward.

Possible Return Values:

- *ProblemDetected* – move cannot be completed.
- *ReachedDestination* – move is successfully completed.

3.3.4 MoveToWall

Function Prototype:

- `int MoveToWall(int WayPointSide).`

Description:

- *MoveToWall* uses visual and sonar information to move the robot to the nearest wall. A while-loop structure drives the robot towards the wall by using *FindNearestWall* to determine angular velocity adjustments and *CheckSonar* to determine forward velocity

adjustments. The loop terminates when the robots is approximately 15 in from the wall. *MoveToWall* calls *RotateIntoWall* to reposition the robot so that it is parallel to the wall.

Input Variables:

- *WayPointSide* – an integer variable designating the desired location of the endpoint. Possible values are POSITIVE, indicating that the robot should keep the endpoint on its right as it drives, and NEGATIVE, indicating that the robot should keep the endpoint on its left as it drives forward.

Possible Return Values

- *ProblemDetected* – move cannot be completed.
- *ReachedDestination* – move is successfully completed.

3.3.5 MoveIntoOpen

Function Prototype:

- `int MoveIntoOpen(int WayPointSide).`

Description:

- *MoveIntoTheOpen* moves the robot to a position that is at least 30 in from any detected obstacle. It also orients the robot so that it is facing the waypoint.
- The program contains a while-loop structure that uses *CheckSonar* to determine its distance to nearby obstacles. The while-loop also calls *KeepEyesOnWayPoint* to keep its cameras facing the waypoint. Once the robot is 30 in from all obstacles, it calls *OrientVehicle* to orient its body toward the waypoint.

Input Variables:

- *WayPointSide* – an integer variable designating the desired location of the endpoint. Possible values are POSITIVE, indicating that the robot should keep the endpoint on its right as it drives, and NEGATIVE, indicating that the robot should keep the endpoint on its left as it drives forward.

Possible Return Values:

- 0 – meaningless.

3.3.6 BoundForward

Function Prototype:

- `int BoundForward(int WayPointSide).`

Description:

- *BoundForward* allows a robot to determine and move to its next destination. The function uses *MoveIntoOpen* to reposition the robot away from obstacles, such as walls, so that it is set up for its next move. It calls *FindNearestWall* to locate its next usable wall, if one exists. If there is a usable wall, the function calls *MoveToWall* to control the robots movement. If there is no wall available, then the robot moves directly to the WayPoint using the function *MoveToWayPoint* to control the movement.

Input Variables:

- WayPointSide – an integer variable designating the desired location of the waypoint. Possible values are POSITIVE, indicating that the robot should keep the endpoint on its right as it drives, and NEGATIVE, indicating that the robot should keep the endpoint on its left as it drives forward.

Possible Return Values:

- CompletedMove – move has been successfully completed.
- ProblemDetected – move cannot be completed.

3.4 Communication

3.4.1 PublishMessage

Function Prototype:

- `int PublishMessage(int robot, char *msg).`

Description:

- *PublishMessage* sends a message from the robot to the message server.

Input Variables:

- robot – an integer variable designating the robot publishing the message.
- msg – a string variable containing the text of the message.

Possible Return Values:

- 1 – meaningless.

3.4.2 GetMessage

Function Prototype:

- `GetMessage(int robot).`

Description:

- *GetMessage* gets a message from the server. It uses the variable, robot to determine which message to retrieve.

Input Variables:

- robot – an integer variable designating the robot that published the message.

Possible Return Values:

- BEARING – message contains bearing information.
- MOVING – message indicates the robot is in the MOVE state.
- WATCHING – message indicates the robot is in the WATCH state.
- STOPPED – the robot has terminated the mission.
- DANGER – the robot has detected movement.
- READY – the robot is ready to perform the mission.

3.4.3 ParseMessage

Function Prototype:

- int *ParseMessage*(char *msg).

Description:

- *ParseMessage* separates an incoming message into a series of words that can be interpreted by other functions in the program. It returns an integer constant, describing the type of message. This type is determined from the first word of the message.

Input Variables:

- msg – a string variable containing the original message from the server.

Possible Return Values:

- BEARING – message contains bearing information.
- MOVING – message indicates the robot is in the MOVE state.
- WATCHING – message indicates the robot is in the WATCH state.
- STOPPED – the robot has terminated the mission.
- DANGER – the robot has detected movement.
- READY – the robot is ready to perform the mission.

3.5 Miscellaneous

3.5.1 StartServers

Function Prototype:

- `int StartServers(int argc, char *argv[]).`

Description:

- *StartServers* links local variables to the servers necessary to run the behavior. There are seven servers used on the robot. The DriveCommand server sends commands to the driving system. The Odometry server provides position information. The Sonar server provides data from the sonar array. Two Camera servers provide images from the cameras. The Pan-Tilt server allows control of the camera gaze. The Compass server provides compass information.
- The two remaining servers, the Information server and the Map server are hosted by other computer systems on the local area network used by the robots. The Information servers allow messages to be passed between the robots. The Map server maintains a shared obstacle map used for debugging purposes.

Input Variables:

- `argv` – a string array containing the command line arguments.
- `argc` – the number of command line arguments.

Possible Return Values:

- None.

3.5.2 PrintImage

Function Prototype:

- `void PrintImage(int ImageNumber, int CameraNumber, int cx, int cy, int cbx, int cby, int DesiredLocation).`

Description:

- *PrintImage* writes an annotated image to an ascii portable pixmap file. In addition to the camera image, the saved image has a 20×20 grid, shown in white. Two pixels, one at (cx and cy) and the other at (cbx and cby) are highlighted in cyan and yellow, respectively. There is a vertical magenta line at `DesiredLocation` for reference. Images are tagged with the robot number, Camera number, and an Image number so that they can be easily organized for post-processing.

Input Variables:

- ImageNumber – integer variable giving the image number that is used to tag stored images.
- CameraNumber – an integer variable specifying the desired camera.
- cx and cy – image pixel to be highlighted in cyan.
- cbx and cby – image pixel to be highlighted in yellow.
- DesiredLocation – location in the image plane of a vertical reference line to be drawn in magenta.

Possible Return Values:

- None.

3.5.3 PrintYellowImage

Function Prototype:

- void PrintYellowImage(int ImageNumber, int CameraNumber, int WallLocation, int DesiredWallLocation, and int LoX, int HiX).

Description:

- *PrintYellowImage* writes the current processed image for the CameraNumber to an ascii portable pixmap file. The image shows yellow wall pixels, neutral pixels, and a grid. The image also shows vertical reference lines at WallLocation and DesiredWallLocation. Images are tagged with the robot number, Camera number, and an Image number so that they can be easily organized for post-processing.

Input Variables:

- ImageNumber – integer variable giving the image number that is used to tag stored images.
- CameraNumber – an integer variable specifying the desired camera.
- WallLocation – integer variable giving location of the vertical wall edge closest to the endpoint.
- DesiredWallLocation – integer variable giving desired location of the wall in the image plane.
- LoX,HiX – integer variables specifying the boundaries of the search region in the image plane. Typically, one boundary is set to the current location of the endpoint, and the other boundary is set to the appropriate edge of the image plane.

Possible Return Values:

- None.
-

4. Conclusions

This report has presented a guide to the software developed for the BoundingMovement behavior implemented on iRobots' ATRV-Jr platforms. It presents a short description of the behavior algorithm and a detailed description of the servers and functions used to implement the algorithm.

In future work, we can use this behavior to study aspects of the bounding overwatch behavior. One aspect we intend to explore is the response to danger. In the current system, the robots respond to danger by increasing speed to get to the next concealed spot quickly. Other behaviors we intend to explore are the use of other assets such as small (<5 lb) robotic assets and aerial robotic platforms that can be used to monitor or destroy the danger. Another aspect we want to explore is the effect of communication delays on system performance. The experimental area is too small to actually affect communications but messages can be delayed to simulate communication delays.

In future work, we will also incorporate more realistic sensor algorithms. In particular, we plan to use more realistic hiding locations in the future. We will modify the *FindWall* routine so that it uses vertical edges, shape, and color information to identify possible hiding locations.

Right now the robots do very little planning to determine their next course of action. By incorporating a world map, from the Map server, the robots could plan their moves more effectively. We will also address this issue in our future research.

Appendix. Defined Constants for the BoundingMovement Code

Table A-1. List of the defined constants used in the BoundingMovement software.

Constant Name	Value	Meaning
YES	1	Successful completion of function.
NO	0	Unsuccessful completion of function.
ProblemDetected	102	Servers did not activate properly.
NotEnoughPixels	103	Image does not contain enough candidate pixels for the analysis.
EnoughPixels	104	Image contains enough candidate pixels for the analysis.
FoundWayPoint	105	Image contains enough candidate pixels to find the endpoint.
ReachedDestination	106	Robot has reached the next wall or endpoint.
StartingMove	107	Robot has begun its move.
StillMoving	108	Robot has nonzero velocity.
SafeDistance	15.0	Maximum safe distance from obstacles in inches.
SAFE	300	Robot is not too close to an obstacle.
TooCloseFront	301	Obstacle near the front of the robot.
TooCloseRear	302	Obstacle near the rear of the robot.
TooCloseLeft	303	Obstacle near the right of the robot.
TooCloseRight	304	Obstacle near the left of the robot.
NormalForwardSpeed	0.3	Normal driving speed in meters per second.
CompletedMove	501	Robot has successfully completed its move.
SomethingInTheWay	502	Robot cannot complete its move because there is an obstacle near the robot.
POSITIVE	1	Endpoint is on the left of the robot.
NEGATIVE	-1	Endpoint is on the right of the robot.
BEARING	1001	Robot sent a Bearing message.
MOVING	1002	Robot sent a Moving message.
STOPPED	1003	Robot sent a Stopped message.
WATCHING	1004	Robot sent a Watching message.
DANGER	1004	Robot sent a Danger message.
READY	1006	Robot sent a Ready message.
UNKNOWN	2000	Robot sent an indecipherable message.
RED	200	Color is red.
BLUE	201	Color is blue.
YELLOW	202	Color is Yellow.
NEUTRAL	203	Color is not blue or red.
Odometry	167	Robot is wheel encoder readings to the map server.
GPS	267	Robot is sending global positioning system readings to the map server.
INU	367	Robot is sending inertial navigation unit readings to the map server.
Compass	467	Robot is sending compass readings to the map server.

NO. OF
COPIES ORGANIZATION

1
(PDF
Only) DEFENSE TECHNICAL
INFORMATION CTR
DTIC OCA
8725 JOHN J KINGMAN RD
STE 0944
FT BELVOIR VA 22060-6218

1 COMMANDING GENERAL
US ARMY MATERIEL CMD
AMCRDA TF
5001 EISENHOWER AVE
ALEXANDRIA VA 22333-0001

1 INST FOR ADVNCD TCHNLGY
THE UNIV OF TEXAS
AT AUSTIN
3925 W BRAKER LN STE 400
AUSTIN TX 78759-5316

1 US MILITARY ACADEMY
MATH SCI CTR EXCELLENCE
MADN MATH
THAYER HALL
WEST POINT NY 10996-1786

1 DIRECTOR
US ARMY RESEARCH LAB
AMSRD ARL CS IS R
2800 POWDER MILL RD
ADELPHI MD 20783-1197

3 DIRECTOR
US ARMY RESEARCH LAB
AMSRD ARL CI OK TL
2800 POWDER MILL RD
ADELPHI MD 20783-1197

3 DIRECTOR
US ARMY RESEARCH LAB
AMSRD ARL CS IS T
2800 POWDER MILL RD
ADELPHI MD 20783-1197

NO. OF
COPIES ORGANIZATION

ABERDEEN PROVING GROUND

1 DIR USARL
AMSRD ARL CI OK TP (BLDG 4600)

NO. OF
COPIES ORGANIZATION

- 1 UNIVERSITY OF NE
ENGINEERING CTR
N118 WALTER SCOTT
S FARRITOR
LINCOLN NE 68588-0656

- 1 UNIVERSITY OF SOUTH FL
COMPUTER SCIENCE AND
ENGINEERING
R MURPHY
4202 EAST FOWLER AVE
ENB342
TAMPA FL 33620-5399

- 1 US MILITARY ACADEMY
D EECS
A SAYLES
WEST POINT NY 10996

- 1 APPLIED PHYSICS LAB
T NEIGHOFF
11100 JOHNS HOPKINS RD
LAUREL MD 20723-6099

ABERDEEN PROVING GROUND

- 24 DIR USARL
AMSRD ARL WM BF
M BARONOSKI
R DEPONTBRIAND
H EDGE
M FIELDS (15 CPS)
G HAAS
T HAUG
T KELLEY
W OBERLE
R VON WAHLDE
S YOUNG