

Amanda Whitbrook

# Programming Mobile Robots with Aria and Player

A Guide to C++ Object-Oriented Control

# Contents

<b>1</b>	<b>Introduction and Installations</b>	<b>1</b>
1.1	The Client-Server Paradigm	1
1.2	Software for Pioneer Robot Control	1
1.3	Pioneer Robot Devices	4
1.4	MobileRobots Software Installations	5
1.4.1	ARIA	6
1.4.2	Mapper3Basic	7
1.4.3	MobileSim	7
1.4.4	ACTS	7
1.5	Player and Stage Installations	8
1.5.1	Prerequisites	9
1.5.2	Player - Default Location	9
1.5.3	Player - Selected Location	10
1.5.4	Selecting Drivers	11
1.5.5	Stage - Default Location	11
1.5.6	Stage - Selected Location	12
<b>2</b>	<b>Programming with the ARIA API</b>	<b>13</b>
2.1	Getting Started	13
2.1.1	Compiling Programs	13
2.1.2	Connecting to a Robot	14
2.2	Instantiating and Adding Devices	16
2.2.1	Ranged Devices	16
2.2.2	Non-ranged Devices	18
2.3	Reading and Controlling the Devices	18
2.3.1	The Motors	18
2.3.2	The Sonar Sensors	21
2.3.3	The Laser Sensor	22
2.3.4	The Bumpers	24
2.3.5	The 5D Arm	27
2.3.6	The 2D Gripper	30
2.3.7	The Pan-tilt-zoom Camera	33

<b>3</b>	<b>Other MobileRobots Inc. Resources</b>	37
3.1	ACTS Software	37
3.1.1	Training the Channels	37
3.1.2	Programming ACTS Using ARIA	44
3.2	MobileSim	46
3.3	Mapper3Basic	49
<b>4</b>	<b>Using ARIA Subclasses</b>	53
4.1	Creating and Using ArAction Subclasses	53
4.2	Creating and Using ArActionGroup Subclasses	57
4.3	Creating and Using ArMode Subclasses	60
<b>5</b>	<b>Programming with Player</b>	63
5.1	Player Configuration Files	63
5.2	Using PlayerViewer	66
5.3	Programming with the Player C++ Client Library	68
5.3.1	Compiling Programs	68
5.3.2	Connecting to a Robot	70
5.4	Instantiating and Adding Devices	73
5.5	Reading and Controlling the Devices	73
5.5.1	The Motors	74
5.5.2	The Sonar Sensors	75
5.5.3	The Laser Sensor	75
5.5.4	The Bumpers	77
5.5.5	The 5D Arm	79
5.5.6	The 2D Gripper	83
5.5.7	The Pan-tilt-zoom Camera	85
5.5.8	The Virtual Blob Finder Device	86
5.5.9	Using the Blob Finder with ACTS	91
<b>6</b>	<b>Stage Simulations</b>	93
6.1	Introduction	93
6.2	Creating World Files	93
6.3	Creating Configuration Files	101
6.4	Running Stage	102
6.5	Accelerated Simulations	104
<b>A</b>	<b>Guide to the Extra Materials</b>	109
A.1	Folders	109
A.2	Testing the Programs	110
	<b>References</b>	111
	<b>Index</b>	113

# Chapter 2

## Programming with the ARIA API

### 2.1 Getting Started

The best source of information is the online help document that comes with the software installation [14]. It is located in `/usr/local/Aria` and has the name “Aria-Reference.html”. All the classes that form the ARIA library are listed and their attributes and methods are described there.

#### 2.1.1 *Compiling Programs*

ARIA programs are compiled under Linux by using `g++` on the command line. All programs must be linked to the ARIA library “`lAria`” and the additional libraries “`lpthread`” and “`ldl`”. The ARIA library is located in `/usr/local/Aria/lib` and the header files are located in `/usr/local/Aria/include`. You will need to add the path `/usr/local/Aria/lib` to the file `/etc/ld.so.conf` and run `ldconfig` in order to access the libraries. As an example, suppose you have a control program named “`test.cpp`” and you wish to create a binary called “`test`”. From the directory where “`test.cpp`” is located, you would type the following:

```
g++ -Wall -o test -lAria -ldl -lpthread -L/usr/local/
    Aria/lib -I/usr/local/Aria/include test.cpp.
```

Alternatively, a suitable bash script such as the example given below can be written to save typing:

```
#!/bin/sh

# Short script to compile an ARIA client
# Requires 2 arguments, (1) name of binary
```



```

ArRobot robot;                                //Instantiate robot      5

ArArgumentParser parser(&argc, argv);          //Instantiate argument parser 6
ArSimpleConnector connector(& parser);         //Instantiate connector      7

/* Connection to robot */

parser.loadDefaultArguments();                 //Load default values        8

if (!connector.parseArgs())                    //Parse connector arguments   9
{
    cout << "Unknown settings\n";           //Exit for errors            10
    Aria::exit(0);                          11
    exit(1);                                12
}

if (!connector.connectRobot(&robot))           //Connect to the robot       13
{
    cout << "Unable to connect\n";           //Exit for errors            14
    Aria::exit(0);                          15
    exit(1);                                16
}

robot.runAsync(true);                         //Run in asynchronous mode   17

robot.lock();                                //Lock robot during set up   18
robot.comInt(ArCommands::ENABLE, 1);          //Turn on the motors         19
robot.unlock();                               //Unlock the robot           20

Aria::exit(0);                               //Exit Aria                  21
}                                              //End main

```

“Aria.h” must be included with all programs (line 1) and before the ARIA library can be used it must be initialised by using `Aria::init()` (line 4). The `ArRobot` class (instantiated here in line 5) is the base class for creating robot objects that you can then connect devices to. An instance of the class essentially represents the base of a robot with no sensors attached and only the motors for actuators [12]. However, MobileRobots describe the class as the “heart” of ARIA as it also functions as the client-server gateway, constructing and decoding packets and synchronising their exchange with the micro-controller [14]. Standard server information packets (SIPs) get sent by the server to the client every 100 milliseconds by default. The `ArRobot` class runs a loop (either in the current thread by using the `ArRobot::run()` method or in a background thread by using `ArRobot::runAsync()`), which is synchronised to the data updates sent from the robot micro-controller. In the above program the `ArRobot::runAsync()` method is used (line 17) after connection has been established. Running the robot asynchronously like this ensures that if the connection is lost the robot will stop.

An `ArArgumentParser` object is instantiated here in line 6. This is a standard argument parser for maintaining uniformity between ARIA-based programs. It ensures that all the configurable elements of an ARIA program (robot IP address etc.)

are passed to it in the same way [12]. The constructor for `ArSimpleConnector` takes a pointer to the `ArArgumentParser` object (line 7). The `loadDefaultArguments()` method of `ArArgumentParser` is called in line 8. This allocates the default arguments required to connect to a local host (either `MobileSim`, see Section 3.2 or the real robot). Once the default arguments are loaded they can be parsed to the `ArSimpleConnector` object by using its `parseArgs()` method (line 9). The `connectRobot()` method can then be used to make the actual connection. A pointer to the `ArRobot` object must be supplied as the argument (line 13).

Before running any commands the motors should be placed in an enabled state, (line 19). It is advisable to lock the robot (line 18) to ensure that the command is not interfered with by other users, and the robot should be unlocked afterwards (line 20). When the program ends ARIA must be exited using the syntax in line 21. If you get a segmentation fault when running the program it may be necessary to remake the files in `/usr/local/Aria` after installation.

## 2.2 Instantiating and Adding Devices

In ARIA devices fall into two categories, ranged devices (sonar, laser and bumpers), which inherit from the `ArRangeDevice` class and non-ranged devices, (anything else, e.g. a pan-tilt-zoom camera or a 2D gripper). There are differences in how these types of device are associated with a robot.

### 2.2.1 Ranged Devices

Ranged devices are instantiated and then added to the robot using `ArRobot`'s `addRangeDevice()` method, which takes a pointer to the device as its argument. Below are some extracts of programs that show how to instantiate a sonar device, a laser device and a set of bumpers, and also how to add them to an `ArRobot` object called "robot".

```
ArRobot robot;                //Instantiate the robot
ArSick laser;                 //Instantiate its laser
ArSonarDevice sonar;          //Instantiate its sonar
ArBumpers bumpers;            //Instantiate its bumpers

robot.addRangeDevice(&sonar);  //Add sonar to robot
robot.addRangeDevice(&laser);  //Add laser to robot
robot.addRangeDevice(&bumpers); //Add bumpers to robot
```

The laser device requires additional initialisation to other devices as it inherits from the `ArRangeDeviceThreaded` class (which inherits from the `ArRangeDevice` class). This means that it is a ranged device that can run in its own thread. It there-

fore requires additional connection to the robot using ArSimpleConnector's connectLaser() method, see line 8 of the program extract below.

```

/* Connection to laser */

Aria::init();                //Initialise ARIA library      1
ArRobot robot;               //Instantiate robot           2
ArSick laser;                //Instantiate laser            3
robot.addRangeDevice(&laser); //Add laser          4
ArArgumentParser parser(&argc, argv); //Instantiate argument parser 5
ArSimpleConnector connector(& parser); //Instantiate connector      6

    .
    .
    .                        //Connect to robot
    .

laser.runAsync();            //Asynchronous laser mode      7

if (!connector.connectLaser(&laser)) //Connect laser to robot      8
{
    cout << "Can't connect to laser\n"; //Exit if error          9
    Aria::exit(0);                10
    exit(1);                      11
}

laser.asyncConnect();        //Asynchronous laser mode      12

```

Lines 1 to 6 instantiate the various objects and lines 8 to 11 make and check the connection. Asynchronous connection is specified in lines 7 and 12 and ensures that the laser will stop if the connection fails. An alternative way of connecting to the laser is shown below.

```

connector.setupLaser(&laser);

laser.runAsync();

if (!laser.blockingConnect())
{
    cout << "Could not connect to SICK laser... exiting\n";
    Aria::exit(0);
    exit(1);
}

```



### 2.2.2 Non-ranged Devices

Non-ranged devices do not inherit from `ArRangeDevice` so are not associated with the `ArRobot` object in the same way. In fact, non-ranged devices may inherit from other base classes, for example an `ArVCC4` object (Canon VC-C4 pan-tilt-zoom camera) inherits from the `ArPTZ` class. In general, the robot is added to non-ranged devices instead of their being added to the robot. Sometimes this may be done as part of the initialisation, for example the program extract below shows how a 2D gripper and Canon VC-C4 pan-tilt-zoom camera are associated with the robot at the same time as they are instantiated:

```
ArGripper gripper(&robot);    //Instantiate gripper and add robot
ArVCC4 ptz(&robot);           //Instantiate Canon VCC4 camera and add robot
```

On the other hand, the robot is added to a 5D arm object by first instantiating the arm and then using its `setRobot()` method to add the robot, see Section 2.3.5 for further details.

```
ArP2Arm arm;                  //Instantiate a 5D arm
arm.setRobot(&robot);          //Add robot to arm
```

An ACTS object (virtual blob finding device) uses its `openPort()` method both to add the robot and to set up communication with the ACTS server running on the robot, see Section 3.1 for further details.

```
ArACTS_1_2 acts;              //Instantiate an ACTS object
acts.openPort(&robot);         //Add robot and set up communication
                               //with ACTS server running on that robot
```

## 2.3 Reading and Controlling the Devices

Once devices have been instantiated and added to the robot, they can be controlled. The rest of this chapter shows how this is achieved in ARIA for the Pioneer's motors, sonars, laser, bumpers, 5D arm, 2D gripper and camera. Programming of the ACTS blob finder is dealt with in Section 3.1.

### 2.3.1 The Motors

Motion commands can be issued explicitly by using the `setVel()`, `setVel2()` and `setRotVel()` methods of the `ArRobot` class; the `setVel()` method sets the desired translational velocity of the robot in millimetres per second, `setVel2()` sets the velocity of the wheels independently and `setRotVel()` sets the rotational velocity of the robot

in degrees per second. In addition there are the `setHeading()` and `setDeltaHeading()` methods, which change the robot's absolute and relative orientation (in degrees) respectively. There is also a method to move a prescribed distance (`move()`) and a method for stopping motion (`stop()`). If a positive double is supplied as the argument for `move()`, the robot moves forwards. If a negative double is supplied the robot moves backwards. Some examples of these methods are shown below. All these use a previously declared `ArRobot` object called "robot".

```
robot.setVel(200);           //Set translational velocity to 200 mm/s
robot.setRotVel(20);         //Set rotational velocity to 20 degrees/s
robot.setVel2(200,250);      //Set left wheel speed at 200 mm/s
                             //Set right wheel speed at 250 mm/s
robot.setHeading(30);        //30 degrees relative to start position
robot.setDeltaHeading(60);    //60 degrees relative to current orientation
robot.move(200);              //Move 200 mm forwards
```

Other methods of interest are `setAbsoluteMaxTransVel()` and `getAbsoluteMaxTransVel()`, which set and get the robot's maximum allowed translational speed. This is useful if you do not want your robot to exceed a given speed for safety reasons. The methods `setAbsoluteMaxRotVel()` and `getAbsoluteMaxRotVel()` do the same for rotational speed and the methods `getVel()` and `getRotVel()` return the robot's translational and rotational speeds respectively, as double values.

Note that more complex forms of motion can be achieved by creating action classes that inherit from ARIA's `ArAction` class and adding the actions to the robot. The actions then provide motion requests that can be evaluated and combined to produce a final desired motion. In this way complex behaviours can be achieved. However you can create actions that do not inherit from `ArAction` if you do not want to implement this particular behaviour architecture. Further details about `ArActions` are provided in Chapter 4. The program below shows user-written methods "wander()" and "obstacleAvoid()" that implement simple wandering and obstacle avoidance behaviours respectively. These methods do not inherit from `ArAction`.

```
/*
*-----
* Wandering mode
*-----
*/

void wander(double speed, ArRobot *thisRobot)
{
    int rand1;           //Whether to change direction
    int rand2;           //Used to decide angle of turn
    int rand3;           //Used to decide direction of turn
    int dir;             //Direction of turn

    srand(static_cast<unsigned>(time(0))); //Set seed
```

```

rand1 = (rand()%2); //Get random no. between 0 and 1

if (rand1 == 0) //1 in 2 chance of turning
{
    rand2 = (rand()%10); //Get random no. between 0 and 9
    rand3 = (rand()%2); //Get random no. between 0 and 1

    switch(rand3) //Get direction based on rand3
    {
        case 0:dir = -1;break; //Turn right
        case 1:dir = 1;break; //Turn left
    }
}
else
{
    dir = 0; //Don't turn
    rand2 = 0;
}

thisRobot->setRotVel(rand2*10*dir/2); //Set rotational speed
thisRobot->setVel(speed); //Set translational speed
}

/*
-----
* Obstacle avoidance mode
-----
*/

void obstacleAvoid(double minAng, double driveSpeed, ArRobot *thisRobot)
{
    double avoidAngle; //Angle to turn to avoid obstacle

    if (minAng ≥ 0 && minAng < 46 ) //If obstacle is to the left
    {
        cout << "TURNING RIGHT!\n";
        avoidAngle = -30.0; //Turn right
    }

    if (minAng > -46 && < 0) //If obstacle is to the right
    {
        cout << "TURNING LEFT!\n";
        avoidAngle = 30.0; //Turn left
    }

    thisRobot.setRotVel(avoidAngle); //Set rotational speed
    thisRobot.setVel(driveSpeed); //Set translational speed
}

```

### 2.3.2 The Sonar Sensors

Sonar devices are instantiated and added to the robot as described in Section 2.2.1. To obtain the closest current sonar reading within a specified polar region, the `currentReadingPolar()` method of the `ArRangeDevice` class can be called. The polar region is specified by the `startAngle` and `endAngle` attributes (in degrees). This goes counterclockwise (negative degrees to positive). For example if you want the slice between -45 and 45 degrees, you must enter it as -45, 45. Figure 2.1 below shows the angular positions ARIA assigns to each of the sonar on the Pioneer robots. The closest reading is returned by the method, but is the distance from the object to the assumed centre of the robot. To obtain the absolute distance the robot radius should be subtracted. This can be done by calling `ArRobot`'s `getRobotRadius()` method. The angle at which the closest reading was taken is obtained by supplying a pointer to the double variable holding that value. An example program that implements the `currentReadingPolar()` method is shown below:

```
ArRobot robot;                //Instantiate the robot
ArSonarDevice sonar;          //Instantiate its sonar
robot.addRangeDevice(&sonar);  //Add sonar to robot
.
.                               //Connect to robot
.

double reading, readingAngle;  //To hold minimum reading and angle
reading = sonar.currentReadingPolar(-45,45,&readingAngle);
                               //Get minimum reading and angle
```

If raw sonar readings are required then the `getSonarReading()` method of the `ArRobot` class can be called. The index number of the particular sonar is used as the argument. The method returns a pointer to an `ArSensorReading` object. By calling the `getRange()` and `getSensorTh()` methods of this class you can obtain both the reading and its angle. If you need all the sonar readings then you should first determine the number of sonar present using the `getNumSonar()` method of the `ArRobot` class and then call the `getSonarReading()` method in a loop. An example user-written method “`getSonar()`”, which prints all the raw sonar readings and their angles is shown below:

```
/*
*-----
* Print raw sonar data
*-----
*/

void getSonar(ArRobot *thisRobot)
{
```



```

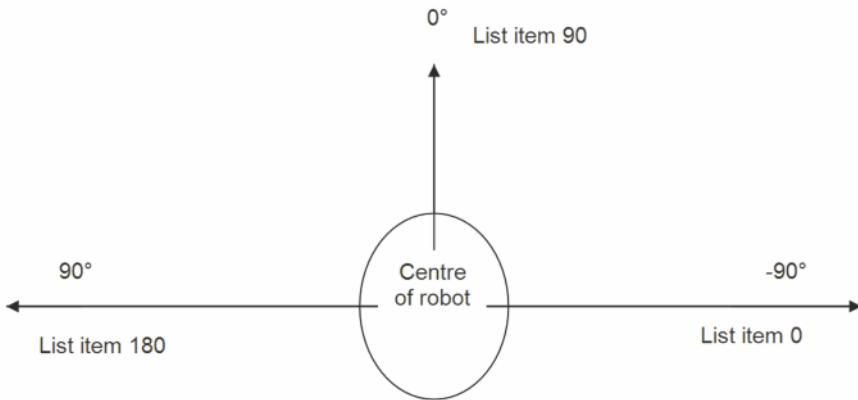
ArSick laser;                //Instantiate its laser
robot.addRangeDevice(&laser); //Add laser to robot
.
.
.                            //Connect to robot

double reading, readingAngle; //To hold minimum reading and angle
reading = laser.currentReadingPolar(-45,45,&readingAngle);
                                //Get minimum reading and angle

```

Another useful method to invoke is the `checkRangeDevicesCurrentPolar()` method of the `ArRobot` class. This checks all of the robot's ranged sensors in the specified range, returning the smallest value. An example using an `ArRobot` object called "robot" is shown below.

```
double reading = robot.checkRangeDevicesCurrentPolar(-45,45);
```



**Fig. 2.2** Laser readings and their positions on the robot (181 readings)

If raw laser readings are required then the procedure is slightly more complex than for sonar sensors as it involves using lists. The method to call is the `getRawReadings()` method of the `ArSick` class. This returns a pointer to a list of `ArSensorReading` object pointers. You will need to loop through this list to obtain the values and angles, so you will also need to declare an iterator object for the list as well as the list itself. You can then loop through each `ArSensorReading` pointer and obtain its reading and angle by calling its `getRange()` and `getSensorTh()` methods. An example user-written method "getLaser()", which prints all the raw laser readings and their angles is shown below:

```

/*
-----
* Print raw laser data
-----
*/

void getLaser(ArSick *thisLaser)
{
    /* Instantiate sensor reading list and iterator object */
    const std::list<ArSensorReading *> *readingsList;
    std::list<ArSensorReading *>::const_iterator it;
    int i = -1;                //Loop counter for readings

    readingsList = thisLaser->getRawReadings();
                                //Get list of readings
                                //Loop through readings
    for (it = readingsList->begin(); it != readingsList->end(); it++)
    {
        i++;
                                //Output distance and angle
        cout << "Laser reading " << i << " = " << (*it)->getRange()
              << " Angle " << i << " = " << (*it)->getSensorTh() << "\n";
    }
}

```

By default the laser should return 181 readings, see Figure 2.2 for the angular positions of each reading. If you require two readings for each degree then you should add the argument `-laserincrement half` when calling your control program. Further details about the SICK LMS200 laser and its operation can be found in [19]. Note that the laser can be simulated using MobileSim, see Section 3.2.

### 2.3.4 The Bumpers

Bumpers are instantiated and added to the robot as described in Section 2.2.1. Once bumpers have been declared you can obtain their state by calling the `getStallValue()` method of the `ArRobot` class. An example program using an `ArRobot` object called “robot” is shown below:

```

int rearBump=0;                //State of bumpers and wheels
int numBumpers;                //Number of bumpers

numBumpers = robot.getNumRearBumpers(); //Find number of bumpers
rearBump = robot.getStallValue();       //Get stall status

```

Table 2.1 below shows how to interpret the integer value returned by the `getStallValue()` method. First convert the integer to a binary number and store it in two bits.

```

        }else
        {
            cout << "Tilting camera down toward blob\n";
            thisPTZ->tiltRel(-1);
        }
    }
}

// Set the heading for the robot

if (ArMath::fabs(xRel) < .10)                //If blob central don't adjust
{
    thisRobot->setDeltaHeading(0);            //XRel should be > 0.1
}else
{
    if (ArMath::fabs(-xRel * 10) <= 10) //If blob central
    {
        thisRobot->setDeltaHeading(-xRel * 10); //Move in required direction
    }else if (-xRel > 0) //If blob is not central
    {
        thisRobot->setDeltaHeading(10); //Move in required direction
    }else
    {
        thisRobot->setDeltaHeading(-10);
    }
}
thisRobot->setVel(speed);                    //Set speed for travel to blob
}
return largestBlob.getArea();                //Return value of largest blob
}

```

## 3.2 MobileSim

MobileSim simulates MobileRobots platforms and their environments, which is useful for debugging and testing ARIA clients. It is a modification of the Stage simulator (see Chapter 6) created by Richard Vaughan, Andrew Howard and others as part of the Player/Stage project, converting Mapper3Basic .map files (see Section 3.3) to the Stage environment and placing a simulated robot model there. Control is provided via TCP port 8101.

The binary is run from the command line by typing `MobileSim`. If no additional parameters are specified a dialogue box is opened, see Figure 3.8. This allows you to select your robot type from the *Robot Model* list box (p3dx is the default) and load a map by clicking the *Load Map* button and selecting a saved Mapper3Basic map. Alternatively, the *No Map* button can be clicked. If no map is specified the usable universe (indicated by a grey colour) is limited to 200 metres by 200 metres.

You can also open a map and specify a robot type from the command line by typing:

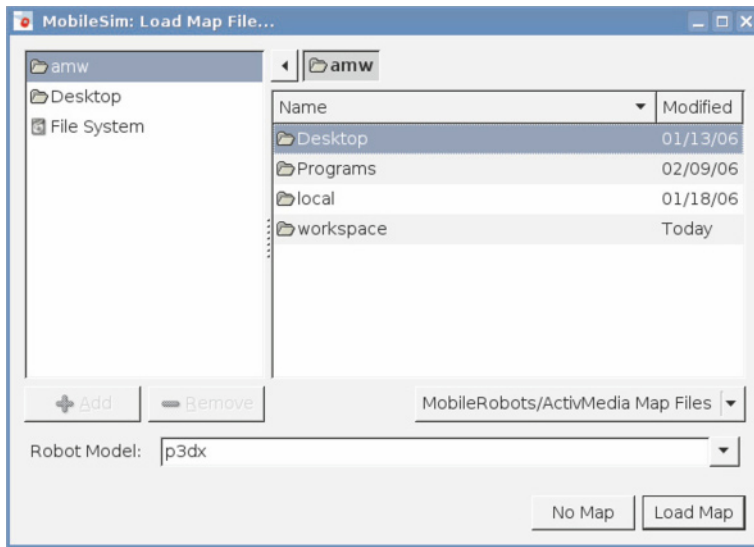


```
MobileSim -m <map file> -r <robot model>,
```

for example,

```
MobileSim -m mymap.map -r p3dx.
```

If you launch the application in this way no initial dialogue box is displayed.



**Fig. 3.8** The initial dialogue box for MobileSim

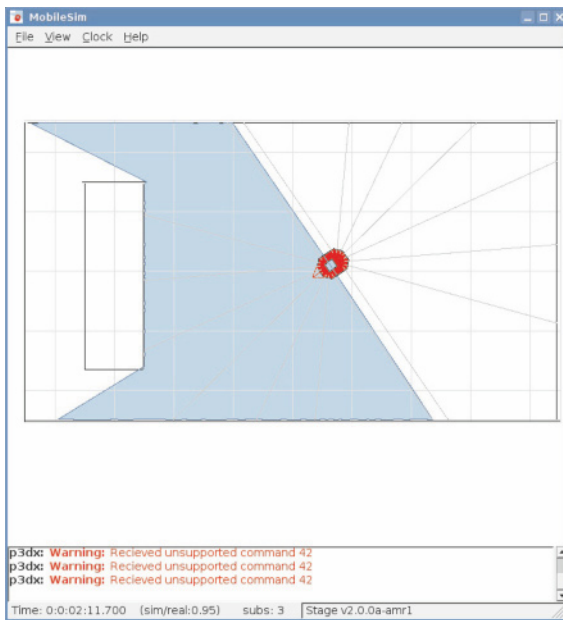
The MobileSim window is opened once the robot type and map have been specified, see Figure 3.9. The map environment and robot are displayed in the centre of the window with the robot at a home position (if this was specified when the map was created) or at the centre. You can pan the window by holding down the right mouse button and dragging and can zoom it with the mouse scroll wheel or by holding down the middle mouse button and dragging towards or away from the centre of the circle that appears. The robot can be moved by dragging it with the left mouse button and can be rotated by dragging with the right mouse button. Both of these actions update the robot's odometry. Grid lines may be added by checking *View* → *Grid* from the menu.

A control program that uses the *ArSimpleConnector* class to connect to a robot will work on the MobileSim simulator without requiring any modification. This is because the class first tries to connect to MobileSim and only tries to connect to a real robot on a serial connection if MobileSim is not running. A program that uses *ArTcpConnection* should also work on the simulator with no modification. To run these programs on the simulator you need only run MobileSim and then type

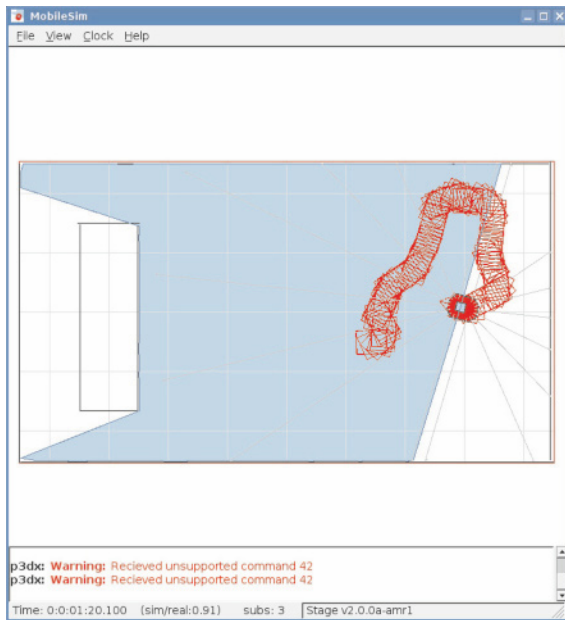
the name of the program's binary into the command line, for example `./test`. Figure 3.9 shows the execution of a wandering and obstacle avoidance program that uses the laser and sonar devices. The area shaded blue represents the laser output and the sonar rays are shown in grey coming from the edge of the robot.

The *File* menu allows the user to load a fresh map (*Load File*), reset the robot to its original position on the map (*Reset*) and export frames or sequences of frames (*Export*). The format for frame export and the duration of the export can also be set. The *View* menu allows various display features to be turned off and on. These include shading the laser range area, showing grid lines, showing the trails that the robot makes, turning off display of the laser and sonar rays and showing position data. Position data gives the odometric pose (x, y and theta values), velocity and true pose. The *Clock* menu allows the user to pause the robot. A display showing the robot's trail and the position data are illustrated in Figure 3.10 and Figure 3.11 respectively.

Note that several devices cannot be simulated by MobileSim. These include grippers, 5D arms, pan-tilt-zoom units, cameras, and blob finding devices, see Table 1.2 for a full list. MobileRobots does not have any immediate plans to update MobileSim to include these devices, but it is likely that a version that includes the gripper will be released before any version that includes the blob finder.



**Fig. 3.9** The MobileSim GUI window



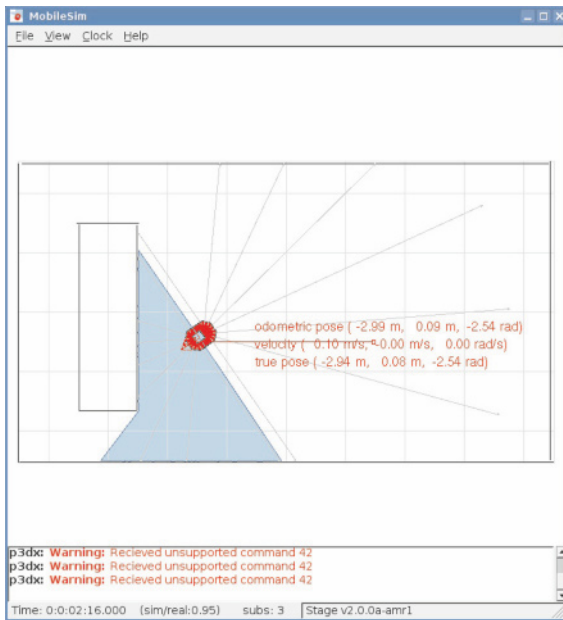
**Fig. 3.10** The simulated Pioneer's trail

### 3.3 Mapper3Basic

Mapper3Basic can be used to create and edit maps for MobileSim (see Section 3.2) so that walls and other obstacles can be simulated. This can be done by drawing map lines, goals, forbidden lines and areas, home points and areas and dock points.

The binary is run from the command line by typing `Mapper3Basic`, which opens a graphical window shown in Figure 3.12. To start a new map select *File* → *New* from the menu and a blank sheet will be loaded. To open an existing map select the *Open* icon or *File* → *Open* from the menu. If you require grid lines you can select *View* → *Grid Lines* from the menu.

Lines, goals and other map objects are placed on the sheet by selecting the appropriate button from the second row and then clicking and dragging the mouse to draw the object. The example above shows four lines drawn to form a rectangle and another four drawn to form an inner rectangle (unshaded). If placed outside the inner rectangle but inside the outer rectangle the robot would be able to move within the outer but would not be able to enter the inner rectangle. However, this is not a truly forbidden area as the robot could be placed within the inner rectangle and would still be free to move around. Forbidden areas are created by selecting the *Forbidden Area* icon and clicking and dragging the mouse over the area that the robot must not enter. These areas are shown shaded orange. In addition, forbidden lines can also be created using the *Forbidden Line* icon. These could be used to prevent the robot from

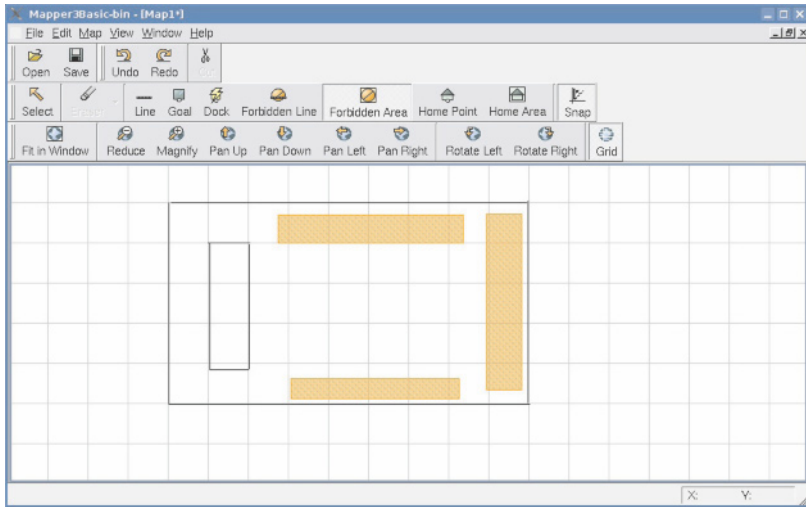


**Fig. 3.11** The simulated Pioneer's position data

getting too close to hazards that cannot be detected with range sensors, for example staircases and holes. If you require your robot to avoid forbidden areas you will also need to create an instance of a virtual ranged device `ArForbiddenRangeDevice` in your ARIA program and add it to the robot. This is used to measure the distances from forbidden areas.

If you require your robot to begin in a particular location on the map then select the *Home Point* icon and click on the point where the robot must begin. Maps are saved as bitmap images in the form of `.map` files by selecting the *Save* icon or choosing *Save* or *Save As* from the file menu. Once saved the maps can be loaded into MobileSim.

Goals, home areas and dock points can also be created. However, these features are for use when creating maps for MobileEyes, MobileRobots' GUI navigation system for remote robot control and monitoring. MobileEyes can connect to ARIA, ArNetworking and ARNL (ARIA's Navigation Library) servers over a wireless network to display the map of the robot's environment. It provides controls to send the robot to goal points or any other point on the map, and also allows the robot to be driven directly with the keyboard or joystick. However, further details about MobileEyes and the navigation library are not included in this guide as details about MobileEyes are available with the online documentation that comes with the software.



**Fig. 3.12** The interface for Mapper3Basic

The next chapter examines the use of subclasses within ARIA and covers each of `ArAction`, `ArActionGroup`, and `ArMode` subclasses.

## Chapter 4

# Using ARIA Subclasses

### 4.1 Creating and Using ArAction Subclasses

Another way of controlling a robot with ARIA is to create action subclasses that inherit from the base ArAction class. When instances of these classes are added to an ArRobot object the robot's resulting behaviour is determined through an action resolver. This invokes each ArAction object (via its fire() method), and the actions request what kind of motion they want by returning a pointer to an ArActionDesired object. The action resolver determines what the resulting combination of those requested motions should be, then commands the robot accordingly. The idea behind this is to have several behaviours acting simultaneously, which combine to drive the robot.

When using the ArAction class direct commands can still be used (for example ArRobot::setVel()), but if you mix direct motion commands with ArAction objects you must fix ArRobot's state by calling ArRobot::clearDirectMotion() before actions will work again.

The program below shows how to create an action that inherits from the ArAction class. This is an adaptation of the actsSimple.cpp program that appears in /usr/local/Aria/examples. Note that it is similar to the blob finding method shown in Section 3.1.2. The difference is that this is a class inheriting from ArAction, whereas the program in Section 3.1.2 was just a method.

```
#include "Aria.h"
#include <iostream>

/* This class moves a robot toward the largest blob seen */

class Blobfind : public ArAction
{
public:

    enum State
    {
        NO_TARGET,
    }
    //State of action
    //No target in view
```

```

    TARGET,                                     //Target in view
};

Blobfind(ArACTS_1_2 *acts, ArVCC4 *camera); //Constructor      2
~Blobfind(void);                             //Destructor
ArActionDesired *fire(ArActionDesired currentDesired);         3
State getState(void) return myState;          //Return state of action

protected:                                     4

    ArActionDesired myDesired;                  5
    ArACTS_1_2 *myActs;
    ArVCC4 *myCamera;
    State myState;
    int myChannel;
};

                                     // Constructor
Blobfind::Blobfind(ArACTS_1_2 *acts, ArVCC4 *camera) :          6
    ArAction("Blobfind", "Moves towards the largest blob.")

{
    myActs = acts;
    myCamera = camera;
    myChannel = 1;
    myState = NO_TARGET;
}

Blobfind::~Blobfind(void)                             //Destructor

// The fire method
ArActionDesired *Blobfind::fire(ArActionDesired currentDesired) 7
{
    ArACTSBlob blob;
    ArACTSBlob largestBlob;

    bool flag = false;
    int numberOfBlobs;
    int blobArea = 10;
    double xRel, yRel;

    myDesired.reset();                                     //Reset desired action      8

    numberOfBlobs = myActs->getNumBlobs(myChannel);

    if(numberOfBlobs != 0)                                //If there are blobs
    {
        myState = TARGET;
        for(int i = 0; i < numberOfBlobs; i++)
        {
            myActs->getBlob(myChannel, i + 1, &blob);
            if(blob.getArea() > blobArea)
            {
                flag = true;
            }
        }
    }
}

```

```

        blobArea = blob.getArea();
        largestBlob = blob;
    }
}
} else
{
    myState = NO_TARGET;
}

if(flag == true)
{
    //Determine where the largest blob's center of gravity
    //is relative to the center of the camera
    xRel = (double)(largestBlob.getXCG() - 160.0/2.0) / 160.0;
    yRel = (double)(largestBlob.getYCG() - 120.0/2.0) / 120.0;

    if(!(ArMath::fabs(yRel) < .20))          //Tilt camera toward blob
    {
        if (-yRel > 0)
            myCamera->tiltRel(1);
        else
            myCamera->tiltRel(-1);
    }

    if (ArMath::fabs(xRel) < .10)             //Set heading and velocity
    {
        myDesired.setDeltaHeading(0);          9
    }
    else
    {
        if (ArMath::fabs(-xRel * 10) <= 10)
            myDesired.setDeltaHeading(-xRel * 10);          10
        else if (-xRel > 0)
            myDesired.setDeltaHeading(10);                  11
        else
            myDesired.setDeltaHeading(-10);                  12
    }

    myDesired.setVel(200);                      13
    return &myDesired;                          14
}
else
{
    myDesired.setVel(0);                      15
    myDesired.setDeltaHeading(0);              16
    return &myDesired;                          17
}
}

```

The important lines in the program are numbered. Line 1 declares the class as a subclass of ArAction. Line 2 declares the constructor, which takes pointers to an ACTS device and a camera as its arguments. Line 3 declares the fire() method, which is the important one to override for subclasses of ArArction. It must return a



pointer to an `ArActionDesired` object to indicate what the action wants to do and can be `NULL` if the action does not want to change what the robot is currently doing. It must also have `ArActionDesired` `currentDesired` as its parameter. This enables the action to determine what the resolver currently wants to do as `currentDesired` refers to the resolver's current desired action. It is used solely for the purpose of giving information to the action.

Line 4 begins declaration of the protected attributes of the class; only subclasses have access to these. Line 5 declares the `ArActionDesired` object called "myDesired". Line 6 begins the constructor method and the right hand side part, for example : `ArAction("Blobfind", "Moves towards the largest blob")` must be included. Line 7 begins the `fire()` method. This method sets the action request by returning the pointer to "myDesired". Line 8 resets "myDesired" and lines 9 to 17 set "myDesired" under different conditions. Note that "myDesired" is used with `ArRobot` direct motion commands like `setDeltaHeading()`.

A main method that uses the above action is given below. This assumes that the above class was saved as "BlobFind.cpp".

```
#include "Aria.h"
#include "BlobFind.cpp" 1
#include <stdio.h>
#include <iostream>

using namespace std;

int main(int argc, char** argv)
{
    ArRobot robot;                //Instantiate robot
    ArSonarDevice sonar;           //Instantiate sonar
    ArVCC4 vcc4 (&robot);          //Instantiate camera
    ArACTS_1.2 acts;               //Instantiate acts device
    ArSimpleConnector simpleConnector(&argc, argv);

    if (!simpleConnector.parseArgs() || argc > 1)
    {
        simpleConnector.logOptions();
        exit(1);
    }

    /* Instantiate actions */
    ArActionLimiterForwards limiter("speed limiter near", 300,600,250); 2
    ArActionLimiterForwards limiterFar("speed limiter far", 300,1100,400); 3
    ArActionLimiterBackwards backwardsLimiter; 4
    ArActionConstantVelocity stop("stop", 0); 5
    ArActionConstantVelocity backup("backup", -200); 6
    Blobfind blobFind(&acts, &vcc4); //Blob finding action 7

    Aria::init();
    robot.addRangeDevice(&sonar); //Add sonar to robot

    /* Connect to the robot */
    if (!simpleConnector.connectRobot(&robot))
```

```

{
    cout << "Could not connect to robot... exiting\n";
    Aria::shutdown();
    return 1;
}

acts.openPort(&robot);           //Connect to acts
vcc4.init();                     //Initialise camera
ArUtil::sleep(1000);             //Wait a second.....
robot.setAbsoluteMaxTransVel(400);
robot.comInt(ArCommands::ENABLE, 1); //Enable motors
ArUtil::sleep(200);

/* Add actions to robot */
robot.addAction(&limiter, 100);      8
robot.addAction(&limiterFar, 99);    9
robot.addAction(&backwardsLimiter, 98); 10
robot.addAction(&blobFind, 77);      11
robot.addAction(&backup, 50);         12
robot.addAction(&stop, 30);           13
robot.run(true);                    //Run the program      14

Aria::shutdown();
return 0;
}

```

Here, line 1 includes the file containing the ArAction subclass “Blobfind”. Lines 2 to 7 declare instances of the action classes that the robot will use; line 7 is the “Blobfind” action created earlier, the others are all standard ArAction subclasses that form part of the ARIA library. ArActionLimiterForwards and ArActionLimiterBackwards limit the forwards and backwards motion of the robot respectively based on range sensor readings, and ArActionConstantVelocity simply sets the robot at a constant velocity. Lines 8 to 14 add the actions to the robot using ArRobot’s `addAction()` method. This method takes a pointer to an ArAction object and an integer value representing the action’s priority as its arguments. The priority values are used by the action resolver to determine the final desired action of the robot. Line 14 runs the program.

## 4.2 Creating and Using ArActionGroup Subclasses

ArActionGroup subclasses are used to wrap a group of ArAction subclasses together to form an action group. This is useful if you have a number of actions that implement a behaviour collectively but you want to be able to activate the behaviour with one call to the group’s `activate()` method. The program below shows how to group actions using a subclass of the ArActionGroup base class. It does the same job as the previous example, i.e. carries out blob tracking at the same time as limiting the forward and backward robot motions. The difference is that here all the actions are