Amanda Whitbrook

Programming Mobile Robots with Aria and Player

A Guide to C++ Object-Oriented Control

Springer

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
```

```
# and (2) name of program to compile

if [ $# != 2 ]; then
   echo Require 2 arguments
   exit 1
fi

g++ -Wall -o $1 -lAria -ldl -lpthread -L/usr/local/Aria
/lib -I/usr/local/Aria/include $2
```

2.1.2 Connecting to a Robot

A method for sending and receiving data to and from the server must be specified. For real robots the server software for low level control runs on the micro-controller and communication between this and the robot PC is through a serial port. If you want to test your programs on a simulator first (on a remote PC) and then run them on a real Pioneer without changing the program, the best way to connect is to use the ArSimpleConnector and ArArgumentParser classes. The ArSimpleConnector class first tries to connect to a simulator if one is detected, otherwise it connects through the serial port of the real robot. For this to work you need to run the control program on the robot PC itself, i.e. connect to the robot first using `ssh` and then run the program. Unfortunately this involves copying the control program from the remote PC to the robot and recompiling. If you want to run the program directly from a remote PC you need to use the separate ArNetworking C++ library to create a server program that runs on the robot PC and a client program that runs on the remote PC. The server program sets up the services that the client program can then request. This involves writing a new control program and is beyond the scope of this guide, which assumes that you will run your program on the robot PC.

Below is an extract of a program that shows how to connect to a robot using ArSimpleConnector.

```
/* Include files */

#include "Aria.h"                                                    1
#include <stdio.h>                                                   2

/* Main method */

int main(int argc, char **argv)                                      3
{

/* The robot and its devices */

Aria::init();                         //Initialise ARIA library      4
```

```
5  ArRobot robot;                          //Instantiate robot

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

   /* Connection to robot */

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

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

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

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

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

21 Aria::exit(0);                          //Exit Aria

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

```
/* Connection to laser */

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

          .
          .
          .                               //Connect to robot
          .

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

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

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

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)
{
```

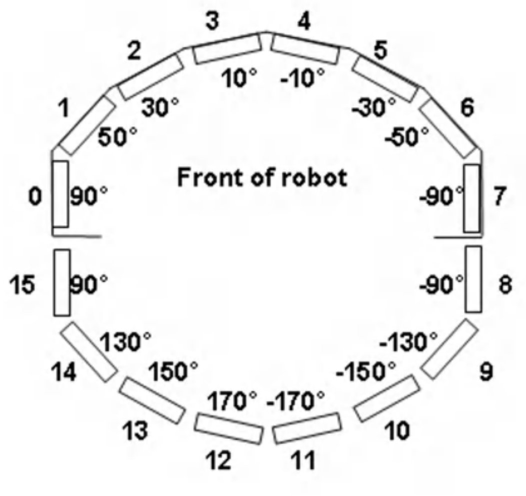

Fig. 2.1 The angular positions of the sonar sensors

```
int numSonar;                          //Number of sonar on the robot
int i;                                 //Counter for looping

numSonar = thisRobot->getNumSonar();  //Get number of sonar
ArSensorReading* sonarReading;         //To hold each reading

for (i = 0; i < numSonar; i++)         //Loop through sonar
  {
    sonarReading = thisRobot->getSonarReading(i);
                                       //Get each sonar reading
    cout << "Sonar reading " << i << " = " << sonarReading->getRange()
       << " Angle " << i << " = " <<
    sonarReading->getSensorTh() << "\n";
  }
}
```

The sonar can be simulated using MobileSim, see Section 3.2.

2.3.3 The Laser Sensor

Laser devices are instantiated, added to the robot and connected as described in section 2.2.1. As both the sonar and laser devices inherit from the ArRangeDevice class, the currentReadingPolar() method can also be used with the laser, see Section 2.3.2. An example program is shown below:

```
ArRobot robot;                     //Instantiate the robot
```

```
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);
```

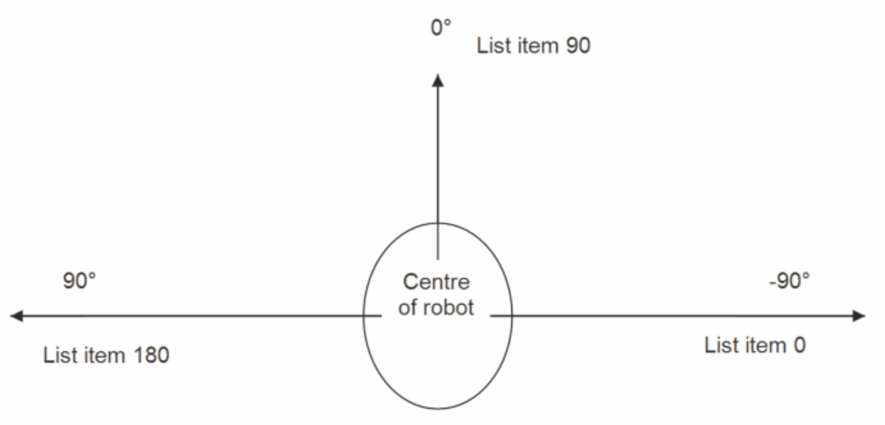

Fig. 2.2 Laser readings and their positions on the robot (181 readings)

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
          {                                  //Move in required direction
            thisRobot->setDeltaHeading(-xRel * 10);
          }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.

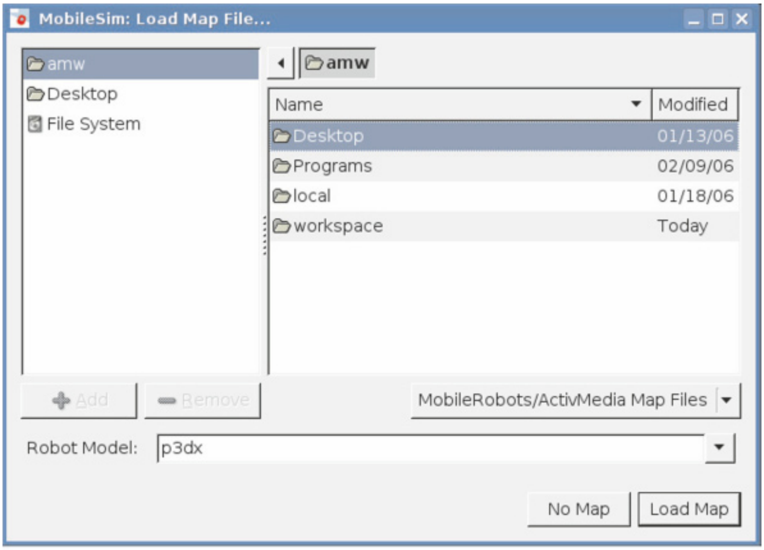

Fig. 3.8 The initial dialogue box for MobileSim

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.

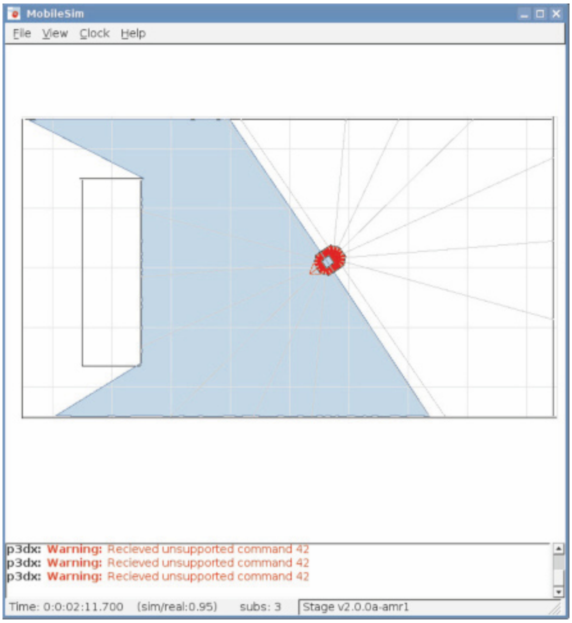

Fig. 3.9 The MobileSim GUI window

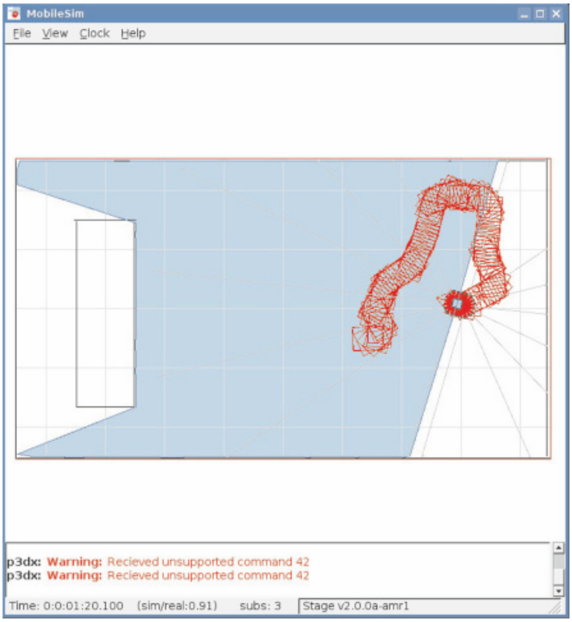

Fig. 3.10 The simulated Pioneer's trail

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

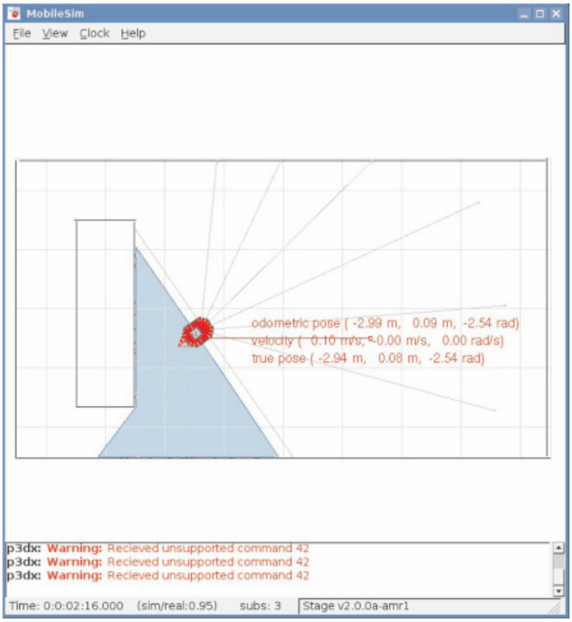

Fig. 3.11 The simulated Pioneer's position data

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.

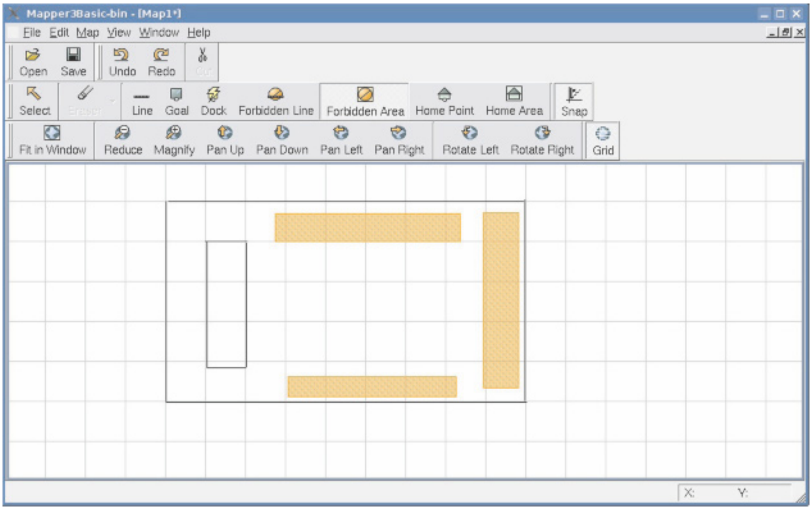

Fig. 3.12 The interface for Mapper3Basic

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