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
```

```
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)
{
```

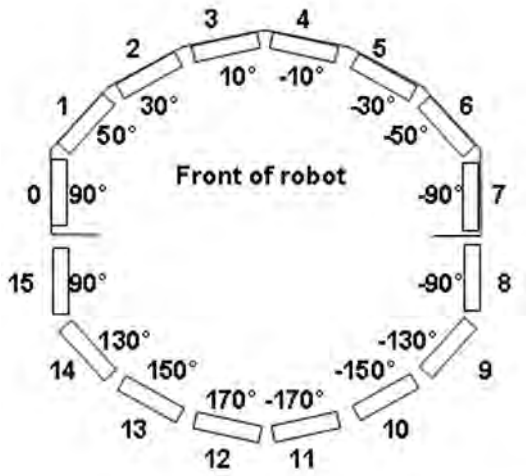

Fig. 2.1 The angular positions of the sonar sensors

```
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);
```

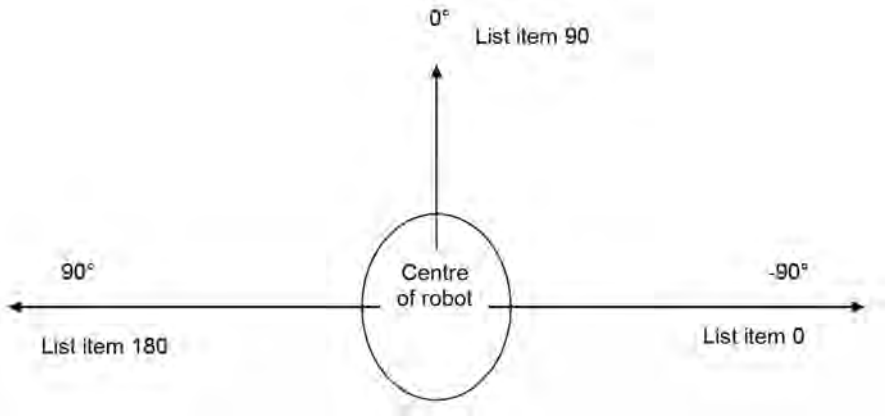

Fig. 2.2 Laser readings and their positions on the robot (181 readings)

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.

For example if 6 was returned this would be 0000000000000110. The interpretation of the integer 6 is that rear bumpers 1 and 2 were bumped. On the Pioneers bumper 1 is the right most rear bumper and bumper 5 is the left most rear bumper, see Figure 2.3. If an integer value of 32 was returned this would mean that bumper 5 was bumped. However, if the left wheel was stalled the integer value would be 1. If the right wheel was stalled it would be 256 and if both were stalled it would be 257.

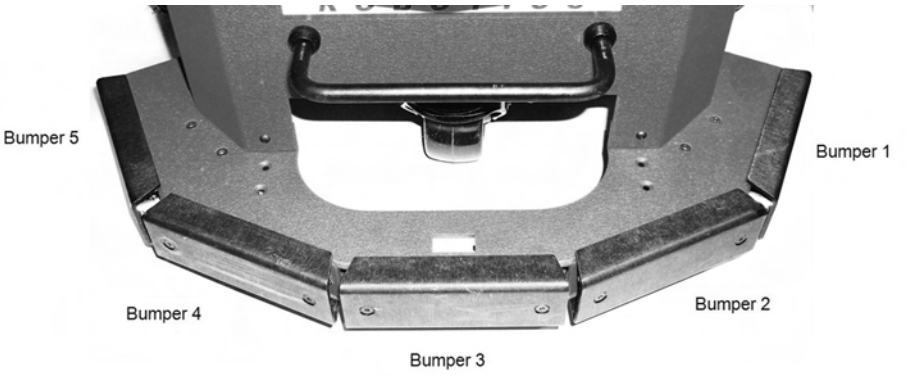

Fig. 2.3 Rear bumpers on the Pioneer robots

Table 2.1 Interpreting the stall integer

Example binary	0	0	0	0	0	0	1	1	0
Bumper number	Right wheel stall	-	-	5	4	3	2	1	Left wheel stall
Decimal component	256	128	64	32	16	8	4	2	1

If you need to check that the correct number of bumpers are being recognised (there are 5 rear bumpers on Pioneer P3-DX robots), then you can call the getNumRearBumpers() method of the ArRobot class, which returns an integer value. There are also methods for checking the number of front bumpers, getNumFrontBumpers(), and for checking whether front and rear bumpers are present, hasFrontBumpers() and hasRearBumpers(), which both return boolean values. The program below shows the implementation of a user-written method “escapeTraps()”, which uses the bumpers to determine where a bump has occurred and how to escape from it. The integer value “bumpVal” supplied to the method should be the result of calling ArRobot::getStallValue() . The double value “minRearReading” should be the smallest reading from the rear sonar, to determine whether the robot should reverse

out of the trap or move forwards. Notice that the method ArUtil::sleep() is called to allow the robot time to carry out the motion commands; the argument to this is in milliseconds.

```
/*
*-------------------------------------------------------------
* Escape traps mode
*-------------------------------------------------------------
*/

void escapeTraps(int bumpVal, double speed, double minRearReading,
   ArRobot *thisRobot)
{

if (bumpVal == 0)
  {
    cout << "TRAPPED AT FRONT MOVING BACKWARDS!\n";
    thisRobot->setRotVel(20);
    thisRobot->setVel(-1*speed);                  //Reverse
    ArUtil::sleep(2000);
  }else
  if (bumpVal > 1 && bumpVal < 63)                //If any bumper registers
    {
      cout << "TRAPPED BEHIND MOVING FORWARDS!\n";
      thisRobot->setRotVel(20);
      thisRobot->setVel(speed);                   //Move forwards
      ArUtil::sleep(2000);
    }else
    if (bumpVal == 1 || bumpVal == 256 || bumpVal == 257)
                                                  //Either wheel has stalled
      {
        cout << "TRAPPED - MOVING EITHER FORWARD OR BACKWARDS!\n";

        if (minRearReading < 200)                 //Trapped at back
          {
            thisRobot->setRotVel(20);
            thisRobot->setVel(speed);             //Move forwards
            ArUtil::sleep(2000);
            cout << "GOING FORWARDS TO ESCAPE\n";
          }else                                   //Not trapped at back
          {
            thisRobot->setRotVel(20);
            thisRobot->setVel(-1*speed);          //Move backwards
            ArUtil::sleep(2000);
            cout << "GOING BACKWARDS TO ESCAPE\n";
          }
      }
    }
}
```

Note that the MobileSim simulator does not generate bump signals other than the right and left wheel stall signals, see Section 3.2.

2.3.5 The 5D Arm

ArP2Arm is the interface to the AROS/P2OS-based Pioneer arm servers. The arm is attached to the robot's micro-controller via an AUX serial port and the arm servers manage the serial communications with the arm controller [14]. The physical arm has 6 open-loop servo motors and 5 degrees of freedom, see [18] for more details. The end effector is a gripper with foam-lined fingers that can manipulate objects up to 150 g in weight. Table 2.2 lists the joints, which are illustrated in detail in Figure 2.4 and Figure 2.5.

Table 2.2 Joints list for the Pioneer 5D arm

Joint Number	Description
1	Rotating base
2	Pivoting shoulder
3	Pivoting elbow
4	Rotating wrist
5	Gripper mount (pivoting)
6	Gripper fingers

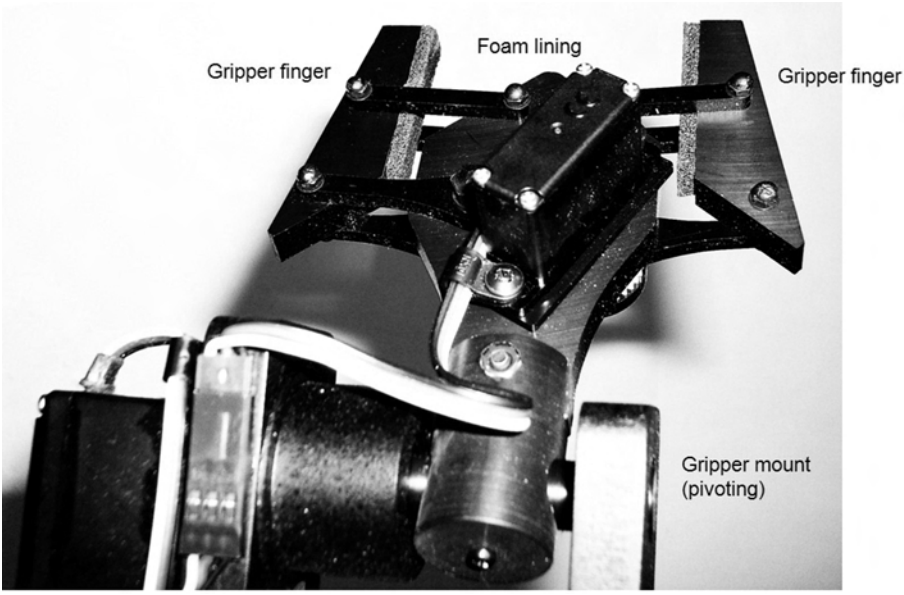

Fig. 2.4 5D arm gripper detail

Fig. 2.5 The joints on the Pioneer

An ArP2Arm object is instantiated and associated with the robot as described in Section 2.2.2, see also lines 1 to 3 of the program below. Note that the ArRobot object that attaches to it must be run in its own thread, i.e. you should use ArRobot::runAsync() if you are using the 5D arm. Following instantiation the arm must be initialised first (line 4). This process communicates with the robot, checking that an arm is present and in good condition [14]. The servos must also be powered on (line 8) before the arm can be used. The program below shows how to do this:

```
ArRobot robot;                          //Instantiate a robot          1
ArP2Arm arm;                            //Instantiate a 5D arm         2
arm.setRobot(&robot);                   //Add arm to robot             3

if (arm.init() != ArP2Arm::SUCCESS)     //Initialize the arm           4
  {
    printf("Arm initialization failed.\n");                            5
    Aria::exit(0);                                                     6
    exit(1);                                                           7
  }
arm.powerOn();                          //Turn on the arm              8
ArUtil::sleep(4000);                    //Wait for arm to stop shaking 9
```

Note that the arm can shake for up to 2 seconds after powering on and if it is told to move before it stops shaking then it can shake even more violently. The powerOn() method of the ArP2Arm class waits 2 seconds by default but it is advisable to include an extra sleep statement as an added precaution (line 9).

The joints in the arm can be controlled by using the ArP2Arm::moveTo() method. This takes three arguments: an integer which specifies which joint is to be controlled, a float which specifies the position to move the joint to (in degrees) and an unsigned char, which specifies the speed of movement. If a velocity of 0 is specified then the current speed is used. Note that each joint has a -90 to 90 degree range approximately, but this can differ between designs. On the Pioneers all the joints rotate through at least 180 degrees, except the gripper fingers. The program below shows commands that move the arm joints (lines 1 to 5) and the fingers (line 6).

```
arm.moveTo(1,45,40);        //Set each joint                    1
arm.moveTo(2,50,40);                                            2
arm.moveTo(3,20,40);                                            3
arm.moveTo(4,10,40);                                            4
arm.moveTo(5,15,40);                                            5
arm.moveTo(6,30,40);        //Set gripper                       6
ArUtil::sleep(6000);                                            7
arm.park();                 //Home arm and power it off         8
arm.uninit();               //Uninitialize the arm              9
```

The gripper at the end of the Pioneer arm is treated like the joints, where the angle passed is proportional to the amount of closing, i.e. to move it you just send the moveTo() command to joint 6. There is a public attribute ArP2Arm::NumJoints that allows the number of joints to be determined. By declaring a P2ArmJoint object it is also possible to obtain information about the state of that joint. This is done by using the getJoint() method of the ArP2Arm class and by reference to the myVel, myHome, myCenter, myPos, myMin, myMax and myTicksPer90 attributes of the class. The program extract below illustrates this:

```
ArRobot robot; ArP2Arm arm;
arm.setRobot(& robot);
if (arm.init() != ArP2Arm::SUCCESS)
  {
    cout << "Arm did not initialise\n";
    exit(1);
  }
P2ArmJoint *joint;
for (i=1, i<ArP2Arm::NumJoints; i++)
  {
    joint = arm.getJoint();
    cout << "Joint " << i << "velocity " << myVel << "home "
       << myHome << "\n";
  }
```

After use the arm should be set to its home position, powered off and uninitialised. Lines 8 and 9 of the previous program show how this is achieved. The park() method both homes the arm and powers it off. This can also be accomplished with the separate methods home() and powerOff(). The home() method takes an integer value as its argument. If -1 is specified all joints are homed at a safe speed. If a single joint is specified only that joint is homed at the current speed. The powerOff() method should only be called when the arm is in a good position to power off as it will go limp. It is safer to use park() as this homes the arm first before it is powered off.

There are a number of other joint controlling methods that can be used. These include moveToTicks(), moveStep(), moveStepTicks(), moveVel() and stop(). The moveToTicks() method works in a similar way to the moveTo() method except the position is specified in ticks instead of degrees. A tick is the arbitrary position value that the arm controller uses. It uses a single unsigned byte to represent all the possible positions in the range of the servo for each joint, so the range is 0 to 255 and this is mapped to the physical range of the servo. This is a lower level of arm control than using moveTo(). The moveStep() method also works in a similar way to moveTo() except that it moves a joint through the specified number of degrees rather than to a fixed position. The moveStepTo() method moves a joint through a specified number of ticks. The moveVel() method sets a particular joint to move at a specified velocity. It takes two integers, the first specifies the joint and the second specifies the velocity. The desired velocity is actually achieved by varying the time between each tick movement. Thus, the attribute value supplied is actually the number of milliseconds to wait between each point in the path; 0 is the fastest, 255 is the slowest and a reasonable range is between 10 and 40. Calling the stop() method simply stops the arm from moving. This overrides all other actions except for initialisation. The 5D arm cannot be simulated using MobileSim as a 3D simulator is required for robot arms.

2.3.6 The 2D Gripper

The 2D gripper is instantiated and added to the robot as described in Section 2.2.2. Physically it is a two degree of freedom manipulation accessory that attaches to the front of the robot, see Figure 1.3 and Figure 2.6. It has paddles and a lift mechanism driven by reversible DC motors, with embedded limit switches that sense the paddle and lift positions. The paddles contain a grip sensor and front and rear infrared break beam switches that close horizontally until they grasp an object or close on themselves. Further details about the gripper device can be found in [14].

Table 2.3 shows the commands (i.e., all the methods of the ArGripper class) that can be used to determine the state of the gripper, the integer values that they return and how these are interpreted. Note that the getGripState() method returns a value of 2 (closed) both when the grippers are closed around an object and when they are fully closed on themselves. The integer value 0 (between open and closed) refers to their being in a moving state, not to their semi-closure. Note also that the paddles

are always triggered when the gripper is closed. When the gripper is open they are triggered only when they are touched with sufficient pressure. Table 2.4 shows the ArGripper methods that can be used to control the gripper.

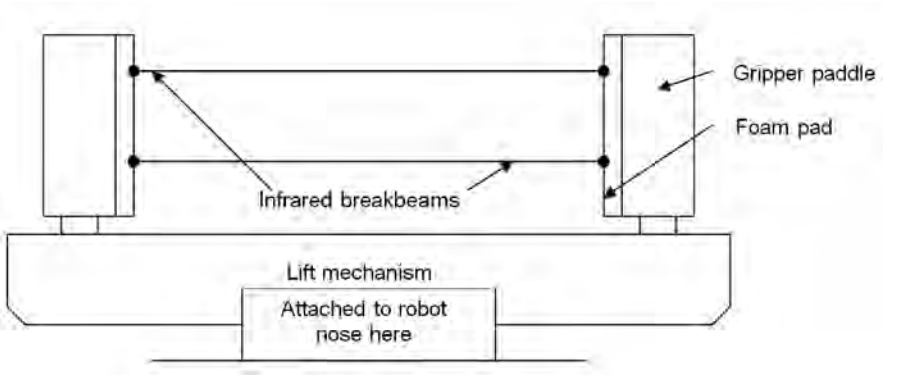

Fig. 2.6 The Pioneer 2D gripper

The program extract below shows use of the gripper's getType() method to check that a gripper is present before it is deployed for action.

```
ArRobot (robot);                          //Instantiate robot
ArGripper gripper(&robot);                //Instantiate gripper and add robot

int gripType;                             //Type of gripper
gripType = gripper.getType();             //Get gripper type
if (gripType != ArGripper::NOGRIPPER)     //If gripper is present
  {
    gripper.gripperDeploy();              //Open and raise gripper
    ArUtil::sleep(4000);                  //Wait while this completes
  }
```

The program sample below shows a user-written method to test the state of the break beams and the paddles and to close the grippers if they are broken by a can. After closure, the state of the break beams is tested again to make sure the can was successfully grabbed. The method returns a boolean value, which indicates whether the grab was successful or not.

```
/*
*-------------------------------------------------------------
* Can gripping routine
*-------------------------------------------------------------
*/

bool canGrip(ArGripper *thisGripper, ArRobot *thisRobot)
{
```

Table 2.3 Methods to obtain the gripper states

Method	Description	Returns	Interpretation
getGripState()	The collective state of the paddles	0 1 2	Between open and closed Open Closed
getPaddleState()	The individual state of the paddles	0 1 2 3	Not triggered Left triggered Right triggered Both triggered
getBreakBeamState()	The state of the break beams	0 1 2 3	None broken Inner beam broken Outer beam broken Both beams broken
getType()	Type of gripper The returned integer maps to an ARIA -defined enumeration value shown in brackets	0 1 2 3 4	Query type (QUERYTYPE) General input output (GENIO) User input output (USERIO) Packet requested from robot (GRIPPAC) No gripper present (NOGRIPPER)

Table 2.4 Methods to control the gripper

Command	Interpretation
gripOpen()	Opens the gripper paddles
gripClose()	Closes the gripper paddles
liftUp()	Raises the lift to the top
liftDown()	Lowers the lift to the bottom
gripperDeploy()	Puts the gripper in a deployed position, ready for use
gripStop()	Stops the gripper paddles
liftStop()	Stops the lift
gripperHalt()	Halts the lift and the gripper paddles

```
int beamState;                                    //State of break beams
int paddleState;                                  //State of the paddles
bool grippedCan = false;                          //Whether can gripped

beamState = thisGripper->getBreakBeamState();     //Get state of beams
paddleState = thisGripper->getPaddleState();      //Get paddle state

cout << "Gripper state is " << gripState << " \n";
cout << "Beam state is " << beamState << " \n";

/* If any beam is broken or paddles are triggered */
if (beamState == 1 || beamState == 2 || beamState == 3 ||
      paddleState == 1 || paddleState == 2 || paddleState == 3)
  {
```

```
    thisRobot->setVel(0);
    thisGripper->gripClose();                    //Grasp can
    ArUtil::sleep(4000);
    beamState = thisGripper->getBreakBeamState(); //Get beam state

    if (beamState == 1 || beamState == 2 || beamState == 3)
      {
        grippedCan = true;
        thisGripper->liftUp();                   //Lift can
        ArUtil::sleep(4000);
      }else
      {
        thisGripper->gripOpen();                 //Re-open as no can
        ArUtil::sleep(2000);
        grippedCan = false;
      }
  }

return grippedCan;                               //Whether grab succeeded
}
```

The user-written method below releases a gripped can, reverses and then turns the robot. Note that the 2D gripper cannot be simulated using MobileSim, see Section 3.2.

```
/*
*-------------------------------------------------------------
* Can dropping routine
*-------------------------------------------------------------
*/

void canDrop(double speed, ArGripper *thisGripper, ArRobot *thisRobot)
{
thisRobot->setVel(0);                        //Stop moving
thisGripper->liftDown();                     //Lower gripper
ArUtil::sleep(4000);
thisGripper->gripOpen();                     //Open gripper
ArUtil::sleep(4000);
thisRobot->setVel(-1*speed);                 //Reverse slowly
ArUtil::sleep(4000);
thisRobot->setRotVel(90);                    //Turn away
ArUtil::sleep(2000);
}
```

2.3.7 The Pan-tilt-zoom Camera

A pan-tilt-zoom camera is instantiated and added to the robot as described in Section 2.2.2. Most Pioneers come with a Canon VC-C4 camera (see [21] and [20] for more details), so the ArVCC4 class, which inherits from the ArPTZ class, must be

used. Once instantiated the camera must first be initialised using the init() method of the ArVCC4 class. The program extract below shows how to do this:

```
ArRobot(robot);                          //Instantiate robot
ArVCC4 ptz(&robot);                      //Instantiate ptz and add robot

bool ptzInitialized;                     //Whether ptz initialized

ptzInitialized = ptz.init();             //Initialize ptz
ArUtil::sleep(4000);
```

Once initialised, the camera can be controlled using the pan(), tilt() and zoom() methods of the ArVCC4 class. These move the camera to a specified angle in degrees, which must be an integer value. In addition, the tiltRel() and panRel() methods can also be used to tilt or pan the camera relative to its present position. Other useful methods include, panTiltRel() and panTilt() which perform the pan and tilt operations together. Here, two integers representing the degrees of pan and tilt respectively are taken as arguments. The current angles of the camera can be obtained by calling getPan(), getTilt() and getZoom(). The methods getMaxPosPan() and getMaxNegPan() retrieve the highest positive and lowest negative values that the camera can pan to (in degrees). The methods getMaxPosTilt() and getMaxNegTilt() do the same for the tilt angles.

The user-written method below performs a simple camera movement exercise, panning and tilting the camera through its full range continuously. Note that a pan-tilt-zoom camera cannot be simulated using MobileSim, see Section 3.2.

```
/*
*-------------------------------------------------------------
* PTZ exercise mode
*-------------------------------------------------------------
*/

void ptzExercise(int inc, bool initPTZ, ArVCC4 *thisPTZ)
{

typedef enum                             //Tilt up or down
{
    up_U,
    down_D,
} VertDirection;

typedef enum                             //Pan left or right
{
    left_L,
    right_R,
} HorizDirection;

int panAngle;                            //Current pan angle
int tiltAngle;                           //Current tilt angle
int lowPan;                              //Lowest pan angle
int lowTilt;                             //Lowest tilt angle
```

```
int highPan;                                //Highest pan angle
int highTilt;                               //Highest tilt angle
HorizDirection hDir;                        //Horizontal direction
VertDirection vDir;                         //Vertical direction

if (initPTZ == true)                        //Camera initialization success
  {
    panAngle = thisPTZ->getPan();           //Get current pan
    tiltAngle = thisPTZ->getTilt();         //Get current tilt
    highPan = thisPTZ->getMaxPosPan();      //Get max pan
    lowPan = thisPTZ->getMaxNegPan();       //Get min pan
    highTilt = thisPTZ->getMaxPosTilt();    //Get max tilt
    lowTilt = thisPTZ->getMaxNegTilt();     //Get min tilt

    cout << "Pan = " << panAngle << " Tilt = " << tiltAngle << "\n";

    if (panAngle == highPan && tiltAngle == highTilt)
      {
        cout << "Changing direction to L and D\n";
        hDir = left_L;                      //Change directions
        vDir = down_D;
      }
    if (panAngle == lowPan && tiltAngle == lowTilt)
      {
        cout << "Changing direction to R and U\n";
        hDir = right_R;                     //Change directions
        vDir = up_U;
      }
    if (hDir == right_R)                    //If going right
      {
        thisPTZ->panRel(inc);               //Increment pan right
      }else
      {
        thisPTZ->panRel(-1*inc);            //Increment pan left
      }
    if (vDir == up_U)                       //If going up
      {
        thisPTZ->tiltRel(inc);              //Increment pan up
      }else
      {
        thisPTZ->tiltRel(-1*inc);           //Increment pan down
      }
  }else
  {
    cout << "Cannot initialize camera\n";   //Error message
  }
}
```

The next chapter looks at some of the other software packages offered by MobileRobots including ACTS, Mapper3Basic and the simulator MobileSim.

http://www.springer.com/978-1-84882-863-6

Programming Mobile Robots with Aria and Player
A Guide to C++ Object-Oriented Control
Whitbrook, A.
2010, XII, 117 p. 33 illus., 5 illus. in color., Softcover
ISBN: 978-1-84882-863-6