Mobile Robots Lab Report

Noah Harvey

December 4, 2014

Abstract

The robot localization problem is a key problem faced in modern day mobile robots. This problem involves obtaining “clean” data about the robot’s environment, determining the current location of the robot, and deciding on an optimal travel path towards a desired location. One method of solving the robot localization problem is the dead reckoning method. This method uses received information from its environment via sensors and then uses the sensor data and previous motion state information to determine its current location.

Mobile robot simulation software is a useful tool for designing and testing localization algorithms. Aria is a C++ SDK used for developing and testing mobile robot software. MobileSim is a simulation software for testing mobile robot software which uses Aria. Both Aria and MobileSim were used to develop and test mobile robot software. This report aims to explore the method of dead reckoning using the Aria mobile robots software, wheel encoders and sonar range finders.

Contents

1 Introduction **2**
1.1 Dead Reckoning 2
1.2 Wheel Encoders 3
1.3 Sonar Range Finder 5

2 Example Robot Programs **7**
2.1 Aria Basics 7
2.2 Square Path Robot 8
2.3 Sonar Measure Robot 9
2.4 S-Curve Robot 10

Appendix A Robot Source Code **11**

List of Tables

1 Phases of Quadrature Encoder 5

List of Figures

1 2 Wheeled Robot (Side view) 2
2 Robot Motion Model from $t = 0-$ Δt 3
3 Quadrature Encoder 4
4 Quadrature Encoder Signals (90° phase shift) 4
5 Ultrasonic Range Finder 6
6 Sonar Finder Signals 6
7 Aria Control System Overview 8
8 Square Robot Path 9
9 Sonar Robot Simulation 10
10 S-Curve Robot Sinulation 11
11 Angles For Parallel Wall Heading 11

1 Introduction

1.1 Dead Reckoning

Robot localization is crucial to effective robot locomotion. Various methods are used to determine the current and future locations of robot; each unique to the available sensors, robot structure and environment. Dead reckoning is one of the simpler methods used. It requires as few sensors as possible and relies mostly on known motion models and states. Dead reckoning is a method of finding the location of an object based on a known motion model and current state information. Once the current state of an object is known then its motion model can be used to predict future motion states. For example, one can predict the future position of a robot if it is known that that the robot is moving in a straight line and at a constant speed and its current position.

A two wheeled robot as shown in Figure 1 will be used for our analysis of dead reckoning. Two wheeled odometry is used to create a motion model of the robot which is then used to predict future locations of the robot. In this study only the x and y coordinates are of interest although velocity can easily derived.

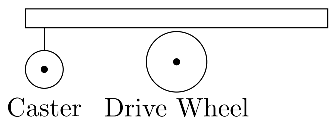

Figure 1: 2 Wheeled Robot (Side view)

We will begin our analysis with the odometry model of the two wheeled robot. The odometry model is as follows (its derivation is beyond the scope of this report):

$$P(t) = \begin{bmatrix} S \\ \theta \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{\mathrm{w}} & -\frac{1}{\mathrm{w}} \end{bmatrix} \begin{bmatrix} S_R(t) \\ S_L(t) \end{bmatrix} \tag{1}$$

Where:

- S is the robot distance travelled.
- θ is the heading of the robot.
- S_L and S_R are the distances traveled by each wheel (left and right respectively).
- w is the distance between the contact points of the wheels of the robot.

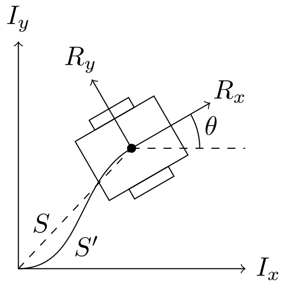

Figure 2: Robot Motion Model from $t = 0-$ Δt

From Equation 1 we can create a motion model relative to the inertial frame of reference I (as shown in Figure 2). To simplify our analysis the following is assumed:

- The robot's motion model is measured on a time differential Δt that is small with respect to the total time (T) the robot performs its motion.
- The robot's wheel encoders are error-less and no wheel slipping occurs.
- Δt is chosen to be small enough so that $S' - S \approx 0$ and thus $S' = S$.

From these assumptions we have a motion model for the robot:

$$\xi_i = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & 1 \end{bmatrix} P(t_i + \Delta\mathrm{t}) - P(t_i) \tag{2}$$

ξ_i is the position of the robot after some time lapse Δt . Thus to get the current position of the robot after some time T simply sum the interval position changes[1]:

$$\xi(T) = \xi_0 + \sum_{i=0}^{n} \xi_i : n = \frac{T}{\Delta\mathrm{t}} \tag{3}$$

Equation 3 simply means that given the robots initial position (ξ_0) and a sequence of wheel distance measurements for $t = \{\Delta\mathrm{t}, 2\Delta\mathrm{t}, \cdots, T\}$ the position of the robot at T can be predicted. This of course is only valid for the above assumptions.

Dead reckoning is not practical when used alone. Sensor noise, wheel slippage, and distance estimation error ($S - S'$) all affect the outcome of Equation 3. However dead reckoning is not completely useless. In fact it forms the base motion model for other filter and motion estimation algorithms (such as the Kalman filter[2]).

1.2 Wheel Encoders

As seen in Equation 1 the robot's motion model is dependent on wheel distance travelled. Sensors that can translate rotational wheel motion to linear distance traveled are called wheel encoders. Different types of encoders are available each with their advantages and disadvantages. In this study we will examine a quadrature wheel encoder.

[1]A continuous form of Equation 3 is beyond ths scope of this lab and is not needed computationally.
[2]`https://en.wikipedia.org/wiki/Kalman_filter`

A quadrature encoder uses a disk (as shown in Figure 3) and two optical sensors to track the position of a wheel. The disk is patterned with alternating opaque and transparent wedges. As the disk spins the optical sensors produce a logic signal (0 or 1) respective to the wedge type it is over. The two sensors are placed in such a manner that their outputs are 90° out of phase (see Figure 4). The encoder is coupled with the wheel in such a manner that the disk spins with an angular velocity that is proportional to that of the wheel.

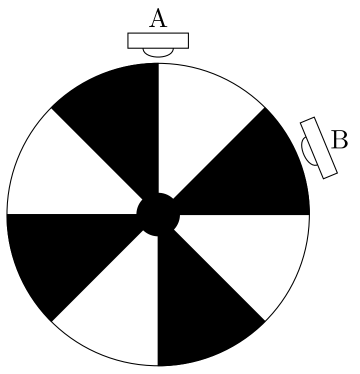

Figure 3: Quadrature Encoder

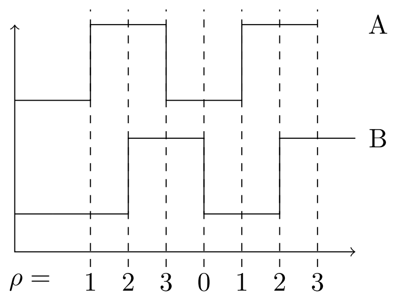

Figure 4: Quadrature Encoder Signals (90° phase shift)

The output from the encoder can be viewed as a 4 phase signal (see Table 1). Each phase change is called a “pulse” and is detected by an observing hardware (typically a MCU). Software can then be used to determine the rotation state of the wheel by analyzing the encoder phase after every pulse it produces.

The direction of the encoder can be determined by two successive phase changes. For example if the current phase is 2 and on the next encoder reading the phase is now 1 the encoder is rotating clockwise. If determining the encoder phase is not performed within a certain amount of time then the software will miss some encoder ticks resulting in erroneous calculations.

Because the encoder uses wedges to indicate rotation the encoder’s resolution is restricted to

Phase(ρ)	A Signal	B signal
0	0	0
1	1	0
2	1	1
3	0	1

Table 1: Phases of Quadrature Encoder

a certain number of pulses per rotation angle. This means that their is a minimum distance the wheel must travel in order for the encoders to detect motion.

The lateral displacement at some time t for each wheel is:

$$S_{L,R}(t) = R\mu \sum_{i=1}^{n} (-1)^{f(\rho_i,\rho_{i-1})} \tag{4}$$

Where:

- μ is the encoder resolution in $[\frac{radians}{pulse}]$
- n is the number of pulses from $t = 0$ to t
- $f(\rho_i, \rho_{i-1})$ is a function that takes the "current" phase and the previous phase and returns 2 or 1 if the direction between the two phases is forward or reverse respectively.

The translational velocity and acceleration of the wheels can be found from Equation 4:

$$\dot{S}_{L,R}(t) = \frac{S_{L,R}(t) - S_{L,R}(t - \Delta\mathrm{t})}{\Delta\mathrm{t}} \tag{5}$$

$$\ddot{S}_{L,R}(t) = \frac{\dot{S}_{L,R}(t) - \dot{S}_{L,R}(t - \Delta\mathrm{t})}{\Delta\mathrm{t}} \tag{6}$$

Note that Δt is the same as defined in section §1.1. Also that $\dot{S}_{L,R}(-\Delta\mathrm{t}) = 0$ and $\ddot{S}_{L,R}(-\Delta\mathrm{t}) = 0$ because any movement previous to $t = 0$ does not exist and to allow $S_{L,R}(t)$ and $\dot{S}_{L,R}(t)$ to be the initial velocity and acceleration.

Wheel encoders are not perfect and are subjected to error. Encoder errors can occur from many sources including: electric noise, slippages, temperature, and wheel velocity. As stated above if the wheel being measured is moving faster than the decoding algorithm ($f(\rho_i, \rho_{i-1})$) can run then encoder missing will happen causing the robot to possibly become disoriented. Thus encoders perform best when the robot is being driven on a smooth, flat surface and their is little to no slippage of the wheels.

1.3 Sonar Range Finder

Wheel encoders are not the only sensors used to determine a robot's position. Often circumstances require that a robot knows its position relative to objects in its environment. Wheel encoders cannot provide this form of information. Instead range finding sensors are used to give the robot information about how far away objects are from it's current position. One such device is the sonar range finder.

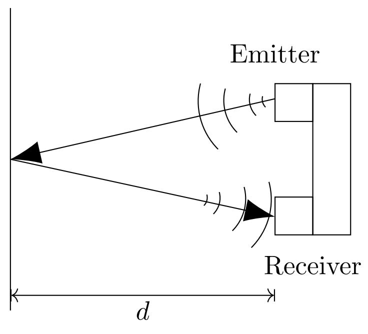

Figure 5: Ultrasonic Range Finder

A sonar range finder uses ultrasonic sound waves to detect nearby objects. The finder is made of two main components: an emitter and receiver. The emitter sends out a high frequency sound wave which is then reflected back to the sensor. The receiver then detects when the reflected sound wave (see Figure 5) returns to the sensor. The time it takes for the sound wave to travel from the sensor to the object and back is called the time of flight (Δt_f) and is used to calculate the distance from the object.

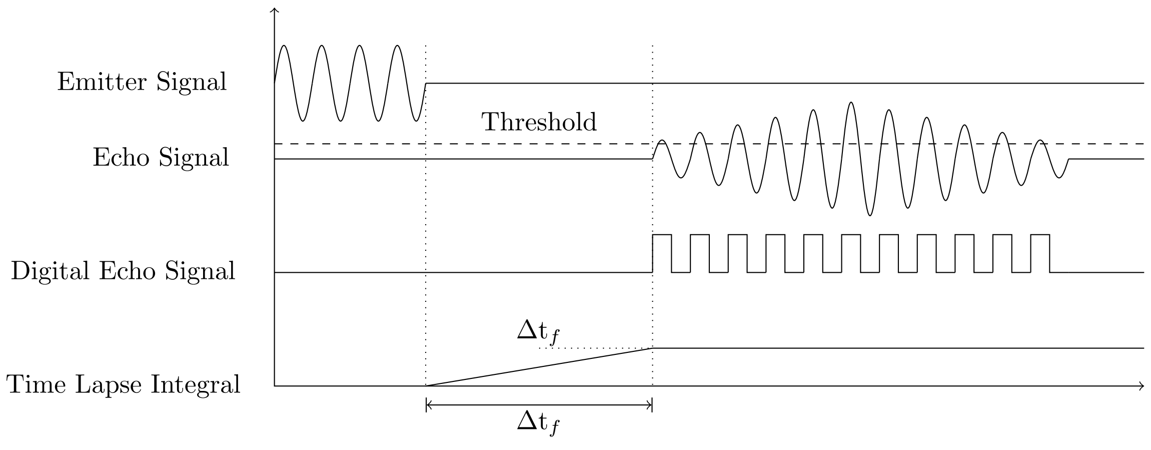

Figure 6: Sonar Finder Signals

Figure 6 shows a sample pulse and receive sequence by the sonar range finder. After the signal is emitted the sensor counts the time (shown as the "Time Lapse Integral") it takes before it receives an echo that meets a set threshold. Once Δt_f is known the distance from the object can be found:

$$d = \frac{c_{air}\Delta \mathrm{t}_f}{2} \tag{7}$$

Where:

- c_{air} is the speed of sound in air
- $\Delta \mathrm{t}_f$ is the time of flight

Figure 5 is slightly misleading. It implies that a single pulse from the sensor can be used to determine the distance towards an object. In reality the sound waves do not reflect off of a single point of an object, nor do the sound waves travel along a single line. Instead the sound waves emit within a range of angles from the sensor. Thus multiple pulses from a stationary sonar range finder will give different experimental values for d. Statistical analysis is typically used to determine an estimate for object location. This analysis is also used to filter out noise.

Sonar range finders do not come without their flaws. One of the largest obstacles a range finder faces is object surface type and sonar noise. Every object surface affects sound reflection differently. If an object's surface tends to scatters sound waves then the receiver will not detect an echo (and thus the robot will not know the object exists). Sonar noise can be created when other range finders are present in the environment, or due to wave scattering. Thus an ideal environment to use a range finder in is one in which few (if any) other range finders are present, and the object surfaces are smooth (so as to minimize scattering).

2 Example Robot Programs

The following example robot programs make use of the Aria and MobileSim softwares. The source code can be found in Appendix A.

2.1 Aria Basics

Aria[3] is an SDK framework written in C++ for controlling mobile robots. It aims to free the programmer from low level tasks (such as signal processing, motor code, and interrupts) to allow for higher level control methods. Figure 7 shows an overview of the control system that is implemented. Essentially the robot runs its own firmware that is in charge of low level commands (gathering sensor data, direct motor control, etc.) which communicates with a server. The server performs higher level data analysis and control methods.

The server handles control methods via control commands. That is there is a series of low level commands that the server communicates to the robot. These low level commands can then be abstracted into higher level motion commands. Motion commands are used to command the robot to perform motion actions such as driving a certain distance or rotating at a certain speed. These commands are converted into low level commands at run time.

Motion commands are also abstracted into what Aria calls Actions. Actions are thought of as modules composed of motion commands that can be connected together to achieve more complex motion. Each action has a set of motion commands it uses to perform a desired action (such as avoiding a wall, or moving forward at a constant velocity). The Actions are added to a list and are evaluated upon runtime to produce a series of low level commands. Each Action is given a weight when added to the action list. This allows for prioritizing of actions and so the actions with higher priority have more influence on the outcome of the robot commands.

[3] `http://robots.mobilerobots.com/Aria/docs/main.html`

Each action can be thought of as running in parallel to the main server code and other actions. This becomes important in the S-curve program where the robot is commanded to go to a position once it has reached a certain location. Because actions are used these commands are independent of one another and the robot can handle internally when to obey the given commands.

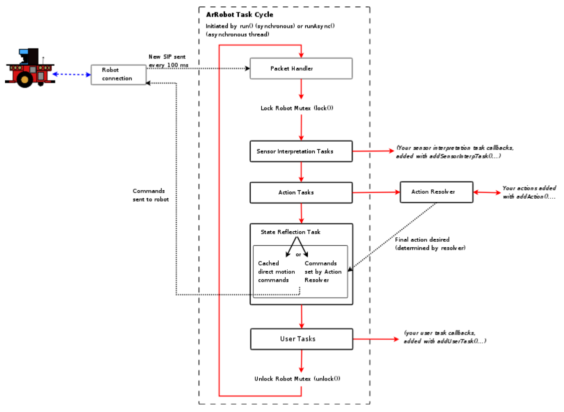
ArRobot Task Cycle
Initiated by run() (synchronous) or runAsync() (asynchronous thread)
Robot connection
New SIP sent every 100 ms
Packet Handler
Lock Robot Mutex (lock())
Sensor Interpretation Tasks
(Your sensor interpretation task callbacks, added with addSensorInterpTask()...)
Action Tasks
Action Resolver
Your actions added with addAction()....
Commands sent to robot
State Reflection Task
or
Cached direct motion commands
Commands set by Action Resolver
Final action desired (determined by resolver)
User Tasks
(your user task callbacks, added with addUserTask()...)
Unlock Robot Mutex (unlock())

Figure 7: Aria Control System Overview

2.2 Square Path Robot

The square robot program is a basic program that drives a robot in a square path. Each side of the square is two meters long and the goal was simply to drive this path as accurately as possible. The algorithm used to travel the square path is show in Algorithm 1.

An Aria action class is used to drive the robot towards the desired goal. It continuously monitors its position and rotates the robot to face the goal so as to make it drive in a straight line (the robot will not drive straight until it is fully rotated towards the desired location). Algorithm 1 and Algorithm 2 run concurrently in software thus the robot will go towards the desired goal once it has been commanded.

Algorithm 1 Procedure for Navigating a Square Path

```
for $i = 0$ to 3 do
    set the next robot position
    while robot has not reached desired goal do
        nothing
    end while
end for
```

Algorithm 2 Drive Straight Towards Goal

```
**if** $\theta - \theta_{Goal} > 0$ **then**
    rotate towards goal
    set translational velocity to 0
**else if** robot is not within goal threshold **then**
    drive robot towards goal
**end if**
```

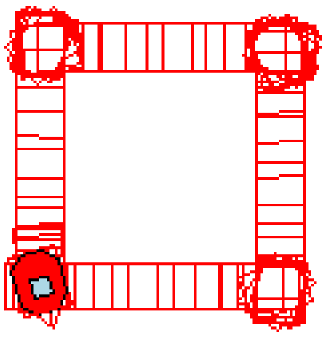

Figure 8: Square Robot Path

Figure 8 shows the result of the square robot simulation. The red boxes are the locations of the robot at certain time frames; it can be viewed as the path the robot moved. Note that at the corners of the square the boxes form circles because of the robot's rotation.

2.3 Sonar Measure Robot

The sonar robot program is the simplest out of those presented here. That is the robot does not move during the program. Instead the robot simply uses sonar range finders to detect the distance and angle towards the nearest object (relative to the robot of course). No Action classes were needed for this program. Instead the function checkRangeDevicesCurrentPolar() is used. This function uses Equation **??** to compute the distance from the object to each sensor. Once each sensor reading has been established it then uses triangulation to determine the objects position. Figure 9 shows the program in action. Each line from the robot represents a sonar sensor's output.

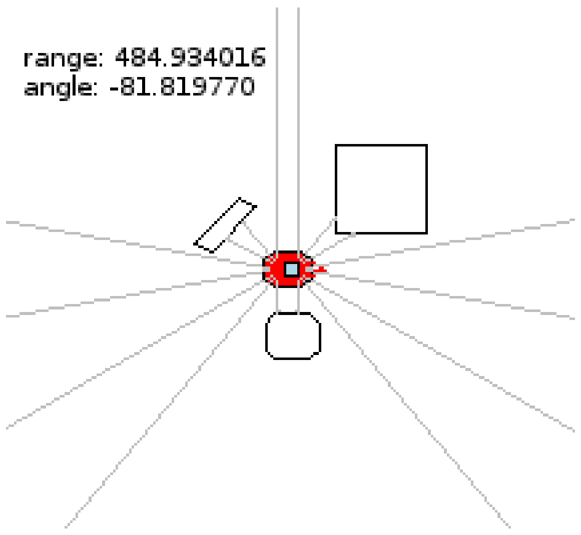

Figure 9: Sonar Robot Simulation

2.4 S-Curve Robot

The S-Curve robot program was designed to move a robot through a curved "canyon" (see Figure 10). An Action class was written to control the heading of the robot so as to move parallel to the nearest object (a wall in this case). At the start of the program the robot is commanded to move with a constant forwards velocity and avoid hitting objects in front of it. This is done internally by the Aria library which uses the wheel encoders to track its velocity and its sonar range finders to detect object distances.

Algorithm 3 Set Heading to Follow Wall

```
get sonar reading (r,$\theta$)
side = $\theta < 0$ ? 1 : 0
angle += $\theta + (2 * side - 1) * 90$
if angle > $\Delta\theta$ then
    newHeading + = $angle$
    set Rotational Veloctiy = angle(.125)
else
    set Rotational Velocity = 0
end if
```

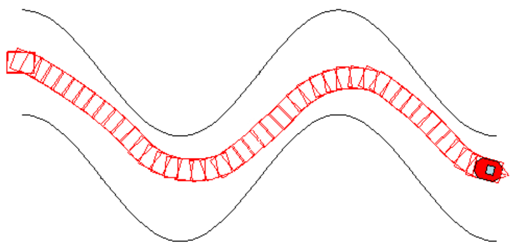

Figure 10: S-Curve Robot Sinulation

The robot monitors whether an object is in front of the robot at any distance. If there is no object then the robot will not move. For this instance it stops the robot once it reaches the end of the canyon. This is necessary for running the robot in real time as to avoid having undesired behavior once it finished following the curve.

Algorithm 3 describes the method use to keep the robot parallel to the walls. The robot needs to know which side the nearest object is on, and then determine the angle to rotate so that it is normal to the wall (θ in Figure 11). Once the angle is known and it is outside of our threshold value ($\Delta\theta$) then set the new heading and rotational velocity.

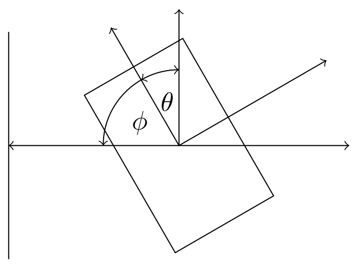

Figure 11: Angles For Parallel Wall Heading

Appendix A Robot Source Code

Below are the source code files for the various robot programs. Each program is implemented as its own function which is called from the main function of the program. Below is the main function. The main function is called with the RUNASNx macro. This macro is replaced with the desired program function call during compile time (this is done through a makefile system which is beyond the support of this report[4]).

main.cpp

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

extern void ASNxHEADER;

int main(int argc, char *argv[])
{
        Aria::init(); // initialize Aria

        ArArgumentParser argParser(&argc, argv); // deal with the
            command line parameters
        argParser.loadDefaultArguments();

        ArRobot robot("blarg"); // create the robot object
        ArRobotConnector robotConnector(&argParser, &robot); //
            create the connector
        if(!robotConnector.connectRobot()) // connect to the robot
        {
          ArLog::log(ArLog::Terse, "Could not connect to the robot.")
              ;
        }

        ArSonarDevice sonar; //create the sonar object and add it to
            the robot
        robot.addRangeDevice(&sonar);

        /** wait for devices to configure */
        //ArUtil::sleep(1000);

        /** run robot control loop in a seperate thread */
        robot.runAsync(true);

  //run asssignment
        RUNASNx;

        /* Shutdown Aria and exit */
        Aria::shutdown();
        Aria::exit(0);
        return 0;
```

[4]https://www.gnu.org/software/make/

```
}
```

Square Robot Code

SquareBot.cpp

```
/*
 * tst.cpp is part of mecharia.
 *
 * mecharia is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published
    by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * mecharia is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with mecharia.  If not, see <http://www.gnu.org/licenses/>.
 */

/**
 * @file tst.cpp
 *
 * Aria test/learning file
 *
 * @author Dr. Matthew Marshall (mmarshall@spsu.edu)
 * @author Noah Harvey (nharvey@spsu.edu)
 *
 * @copyright GNU Public License 2
 */
#include <Aria.h>
#include "ariaTypedefs.h"
#include "ariaUtil.h"
#include "ArAction.h"
/**< square shapre distance is 2 meters */
const int SQRDIST = 2000;

const double ps[4] = {0,SQRDIST,SQRDIST,0};
typedef struct
{
        unsigned char xcnt:2;
        unsigned char ycnt:2;
}cntr_t;

class ArActionGotoCust : public ArActionGotoStraight
```

```
{
        public:
                ArActionGotoCust() : ArActionGotoStraight("goto",10000){};
                virtual ~ArActionGotoCust() {};
                virtual ArActionDesired* fire(ArActionDesired);
};

/** @brief custom fire action to have robot rotote before it moves linearly
 *
 * code copied from ArActionGotoStraight.h
 */
ArActionDesired *ArActionGotoCust::fire(ArActionDesired currentDesired)
{
  double angle;
  double dist;
  double distToGo;
  double vel;

  // if we're there we don't do anything
  if (myState == STATE_ACHIEVED_GOAL || myState == STATE_NO_GOAL)
    return NULL;

  ArPose goal;
  if (!myUseEncoderGoal)
  {
    goal = myGoal;
    myDistTravelled += myRobot->getPose().findDistanceTo(myLastPose);
    myLastPose = myRobot->getPose();
  }
  else
  {
    goal = myRobot->getEncoderTransform().doTransform(myEncoderGoal);
    myDistTravelled += myRobot->getEncoderPose().findDistanceTo(myLastPose);
    myLastPose = myRobot->getEncoderPose();
  }

  if (myJustDist)
  {
    distToGo = myDist - myDistTravelled;
    dist = fabs(distToGo);
  }
  else
  {
    dist = myRobot->getPose().findDistanceTo(goal);
  }

  if (((myJustDist && distToGo <= 0) ||
        (!myJustDist && dist < myCloseDist))
      && ArMath::fabs(myRobot->getVel() < 5))
  {
    if (myPrinting)
      ArLog::log(ArLog::Normal, "Achieved goal");
    myState = STATE_ACHIEVED_GOAL;
    myDesired.setVel(0);
    myDesired.setDeltaHeading(0);
    return &myDesired;
  }

  // see where we want to point
  angle = myRobot->getPose().findAngleTo(goal);
  if (myBacking)
    angle = ArMath::subAngle(angle, 180);

  myDesired.setHeading(angle);

          /**
           * added by Noah Harvey
           * rotate towards our target before we start moving
           */
          if(myRobot->getRotVel() == 0)
          {
                  // if we're close, stop
            if ((myJustDist && distToGo <= 0) ||
                (!myJustDist && dist < myCloseDist))
            {
              myDesired.setVel(0);
              vel = 0;
            }
            else
            {
              vel = sqrt(dist * 200 * 2);
              if (vel > mySpeed)
                vel = mySpeed;
              if (myBacking)
                vel *= -1;
              myDesired.setVel(vel);
            }
          }
          else
                  myDesired.setVel(0);

  if (myPrinting)
    ArLog::log(ArLog::Normal, "dist %.0f angle %.0f vel %.0f",
                 dist, angle, vel);

  return &myDesired;
}

void asn1(ArRobot* robot)
{
          char i;
          cntr_t cntr;
          cntr.xcnt = 1;
          cntr.ycnt = 0;

          if(!robot)
                  return;

          ArLog::log(ArLog::Normal,"Starting Assigment 1");

          /** set up robot actions */
          ArActionGotoCust gotopnt;
          gotopnt.setCloseDist(0);
          gotopnt.setRobot(robot); //uneeded
          robot->addAction(&gotopnt,100);

          /** turn on the motors */
          robot->enableMotors();

          /** set the target points in turn */
          for(i = 0; i < 4; i++)
          {
                  gotopnt.setGoal(ArPose(ps[cntr.xcnt++],ps[cntr.ycnt
                      ++]),false,false);
                  while(!gotopnt.haveAchievedGoal());
          }

          robot->disableMotors();
          ArLog::log(ArLog::Normal,"End of Assigment 1");

          return;
}
```

Sonar Measure Robot

SonarBot.cpp

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

/**
 * @brief control func for asn2
 *
 * call this function in main
 *
 * @return void
 *
 */
void asn2(ArRobot* robot)
{
        if(!robot)
                return;
        double range,angle;

        ArUtil::sleep(1000);

        while(1)
        {
                /** get sonar range infront of robot */
                range = robot->checkRangeDevicesCurrentPolar(-90,90,&
                    angle);
                ArLog::log(ArLog::Normal,"range:␣%f␣angle:␣%f",range,
                    angle);
                ArUtil::sleep(100);
        }
}
```

S-Curve Robot

SCurveBot.cpp

```
#include <Aria.h>
#include <ArAction.h>

#include "ArActionWallFollow.h"
#include "ArActionStopAct.h"

void asn4(ArRobot* robot)
{
        if(!robot)
                return;

        /** add robot actions */
        ArActionWallFollow wallFollow(ArActionWallFollow::leftSide
            ,0.2);
        ArActionConstantVelocity constantVelocity("Constant␣Velocity"
            , 400);

        robot->addAction(&wallFollow,50);
        robot->addAction(&constantVelocity,49);

        robot->enableMotors();
```

```

            while(wallFollow.nowall == false);
}
```