CIS 32 Spring 2009, Project III

1 Description

This project involves writing another control program for the Pioneer robot. The aim of this lab is to get you thinking about how to have the robot work towards a specific goal.

2 Get ready.

I'm going to assume you know how to do this by now. If you don't, ask!

3 A new simulated world

1. You need to add four new files to your working directory:
   - `local-pioneer.cc`
   - `world3.cfg`
   - `world4.cfg`
   - `world4.world`
2. In one terminal, type:

   `player world3.cfg`

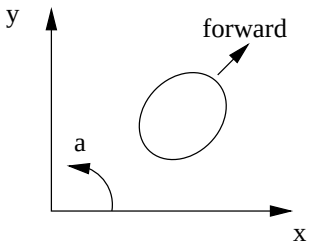
   This should pop up a square window labelled **Player/Stage: ./world2.world** which contains a red blob and a strangely shaped lump insides a jagged line. This is the simulated world in which your robot will operate.
3. The world is the same as you used in the previus exercise, but you now have more information about it.
4. As before, the red blob is the simulated Pioneer.

4 The challenge

1. The challenge is to write a controller that will move the robot from *any* initial position to *any* goal position.
2. To get you started, you have the skeleton program `local-pioneer.cc`, which you can also find at the end of these instructions.
3. `local-pioneer.cc` contains two new things (it has also undergone some re-arrangement since the `main` was getting too long, and one of the golden rules of programming that I try to follow is to not hve any function that is longer than a printed page).
4. The first new thing is the location the robot has to move to. This is hard-coded into the variables `goalX` and `goalY`.
5. The other new thing is the `LocalizeProxy`. This gives us information about where the robot is. It gives us three pieces of information:
   (a) the x coordinate of the robot;
   (b) the y coordinate of the robot;
   (c) the angle of the robot, in radians. If the robot has an angle of zero radians, then it is pointing directly along the x-axis in a positive direction, and the value of the angle increases towards the positive y-axis.

This information is bundled up in the variable `pose`.

Note that these measurements, unlike the ones you have been using up now, are *absolute*, they aren't relative to the robot.



6. Try compiling and running `local-pioneer`. This won't move the robot, but it should show you the new information you have to work with.
7. You can pick up the robot and move it around the simulated world with the mouse/trackpad.
8. Now, write that controller.

5 The rest of the challenge

1. Once you have your controller working in `world3`, make sure that it works in `world4`. All you have to do is to run:

   ```
   player world4.cfg
   ```

2. Check that the robot can get to the goal, when the goal is at:

   (a) $(5, 5)$
   (b) $(0, 7)$
   (c) $(6, 3)$
   (d) $(6, -3)$

```
//-----------------------------------------------------------------------
//
// local-pioneer.cc
//
// Example player libplayerc++ controller that uses sonar to handle a
// pioneer robot.
//
// The robot is just a simple wall-follower, but it is using the
// fake-localise feature of player/stage to know where the robot is.
//
// Written by: Simon Parsons
// Modified  : 2nd March 2009
//
// Usage:
//
// wall-follow [-h <host>] [-p <port>] [-i <index>] [-m]
//
//  -h <host> : connect to Player on this host
//  -p <port> : connect to Player on this TCP port
//  -i <index>: connect to this device (default 0)
//  -m        : turn on motors

//-----------------------------------------------------------------------

#include <iostream>
#include <cmath>
#include <libplayerc++/playerc++.h>

#include "args.h" // Code that allows us to handle standard arguments

using namespace PlayerCc;

//-----------------------------------------------------------------------

player_pose2d_t readPosition(LocalizeProxy&);
void printRobotData(player_pose2d_t, SonarProxy&);
double distanceToGoal(player_pose2d_t, double, double);

//-----------------------------------------------------------------------

int main(int argc, char *argv[])
{
  //-----------------------------------------------------------------------
  //
  // Variables

  // For handling localization data

  player_pose2d_t          pose;

  // For setting speed

  player_pose2d speed;
```

```
const  double stdSpeed = 0.4;
double xSpeed;
double ySpeed;
double tSpeed;

// Where we are going to:

double goalX = -3.3;
double goalY = -3.3;
double distance;

//-----------------------------------------------------------------

// Parse the arguments and set up the relevant variables.
parse_args(argc,argv);

// Use the arguments to connect to the specified device.
PlayerClient robot (gHostname, gPort);
Position2dProxy pp (&robot, gIndex);
SonarProxy sp (&robot, gIndex);
LocalizeProxy lp (&robot, gIndex);

// Control loop starts

while(true)
{
  // Read values from the simulated robot. This reads new sensor
  // values into the sonar proxy sp, and new position data into the
  // localization proxy lp.
  robot.Read();

  // Load the position of the robot into the variable pose, which is
  // a struct with three parts --- x, y and angle.
  pose = readPosition(lp);

  // Print data on the robot to the terminal
  printRobotData(pose, sp);

  // How far are we from the goal?
  distance = distanceToGoal(pose, goalX, goalY);
  std::cout << "Distance to goal" << std::endl;
  std::cout << distance << std::endl;

  // Start with speed zero
  xSpeed = 0;
  ySpeed = 0;
  tSpeed = 0;

  //
  //
  // Write your controller in here
  //
  //
```

```
    // Setup speed values to send them to the robot.
    // speed.px is speed in the x direction
    // speed.py is speed in the y direction
    // speed.pa is rotational speed.
    speed.px = xSpeed;
    speed.py = ySpeed;
    speed.pa = tSpeed;

    // Now use those values to drive the robot
    //
    // This time around, setting ySpeed won't have any effect since the robot
    // is differential drive.
    pp.SetSpeed(speed);
  }

  // Control loop ends
}

//-------------------------------------------------------------------

// readPosition
//
// Read the position of the robot from the localization proxy.
//
// The localization proxy gives us a hypothesis, and from that we extract
// the mean, which is a pose.

player_pose2d_t readPosition(LocalizeProxy& lp)
{

  player_localize_hypoth_t hypothesis;
  player_pose2d_t          pose;
  uint32_t                 hCount;

  // Need some messing around to avoid a crash when the proxy is
  // starting up.

  hCount = lp.GetHypothCount();

  if(hCount > 0){
    hypothesis = lp.GetHypoth(0);
    pose       = hypothesis.mean;
  }

  return pose;
}

// printRobotData
//
// Print out data on the state of the sonar and the current location
// of the robot.

void printRobotData(player_pose2d_t pose, SonarProxy& sp)
{
```

```
  // Print out the sonar values. We can treat the sonar readings as
    // if they are an array of 16 values
    //
    // Left to right across the front.
    std::cout << "Sonar front" << std::endl;
    std::cout << sp[0] << std::endl << sp[1] << std::endl;
    std::cout << sp[2] << std::endl << sp[3] << std::endl;
    std::cout << sp[4] << std::endl << sp[5] << std::endl;
    std::cout << sp[6] << std::endl << sp[7] << std::endl << std::endl;
    //
    // Right to left across the back
    std::cout << "Sonar rear" << std::endl;
    std::cout << sp[8] << std::endl << sp[9] << std::endl;
    std::cout << sp[10] << std::endl << sp[11] << std::endl;
    std::cout << sp[12] << std::endl << sp[13] << std::endl;
    std::cout << sp[14] << std::endl << sp[15] << std::endl << std::endl;

    std::cout << "Current position" << std::endl;
    std::cout << "X: " << pose.px << std::endl;
    std::cout << "Y: " << pose.py << std::endl;
    std::cout << "A: " << pose.pa << std::endl;
}

// distanceToGoal
//
// Given the current pose and the goal coordinates, report the
// distance between the two.

double distanceToGoal(player_pose2d_t pose, double goalX, double goalY)
{
  double poseX = pose.px;
  double poseY = pose.py;
  double distance;

  distance = sqrt(pow((poseX - goalX), 2) + pow((poseY - goalY), 2));

  return distance;
}
```