PoE Lab 03 : Line-Following Robot

Yoonyoung Cho, Eric Miller

October 13 2016

1 Introduction

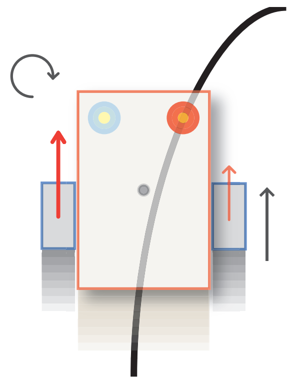

Figure 1: Our Line Follower architecture, a two-wheeled differential drive robot equipped with two IR reflectance sensors positioned at the front.

In implementing the line-follower robot, we adopted three different, yet complementary schemes for control algorithms: *Bang-Bang*, *PID*, and *Memory-Based* control. In parallel to development of these methods, we also developed a simulator that would visualize a virtual hardware interface with native arduino code as a test bench for later phases in development and debugging.

2 Hardware

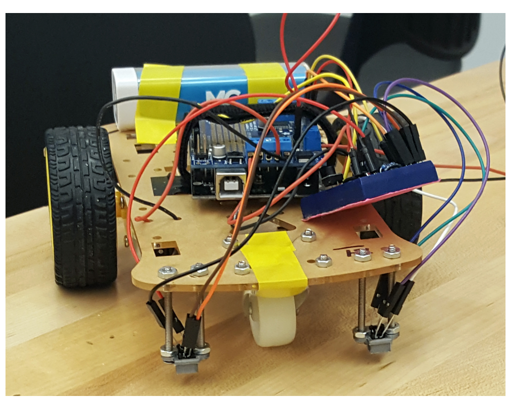

Figure 2: Final assembly of our robot.

In order to optimize for the reaction time from when the sensor detects a change in the environment, we reversed the orientation of the chassis so that what would otherwise be front would be the back side: i.e. we turned the *Tadpole* architecture to *Delta*.

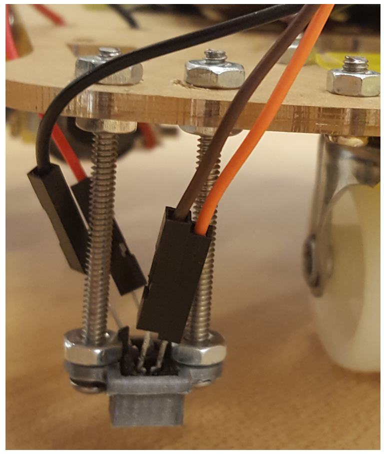

Figure 3: Mount for the IR reflectance sensor.

We have noticed that the sensor calibration was highly sensitive to the height at which the sensor was mounted; i.e.

$$r = htan(fov/2)$$

where h is the height of the IR sensor, fov is its generic field of view, and r is the radius of the conical section model from which the sensor will poll the values. In addition to this, the signal strength would depend heavily on h as well, as some of the reflected light gets dissipated to the atmosphere, etc. To this end, we designed a minimalistic adjustable-height mount that operates with nuts and bolts, which made it not only convenient to disassemble, but also possible to rapidly adjust the calibration parameters as needed.

3 Control

3.1 BangBang

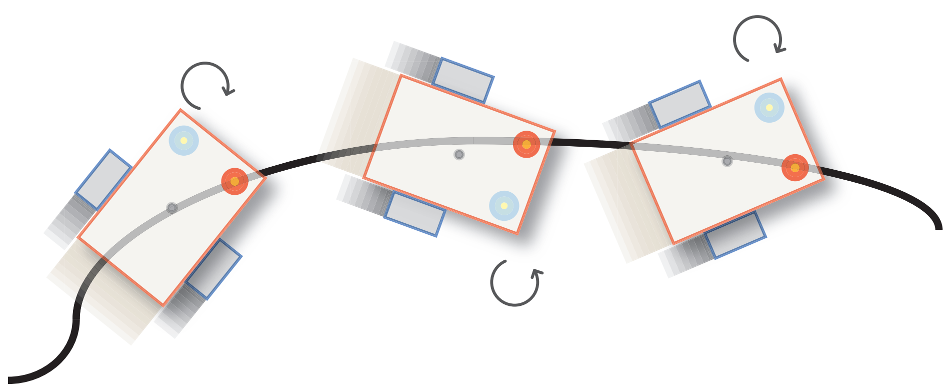

Figure 4: Simplified diagram that illustrates behavior of the *Bang-Bang* controller.

As an initial testing step, we employed the classical control algorithm for the Line-Follower Robot: *BangBang*.

With Bang-Bang control, the robot is, at any given moment, under either of the two states: turning left or turning right. Because the robot cannot incorporate past trajectory into its decision-making process, its decisions are momentary and quite extreme, and the robot is always turning. Whereas this is certainly not optimal, the robot will follow the line in an oscillatory trajectory as the line would always be contained within the perimeters of the two sensors.

3.2 PID Control

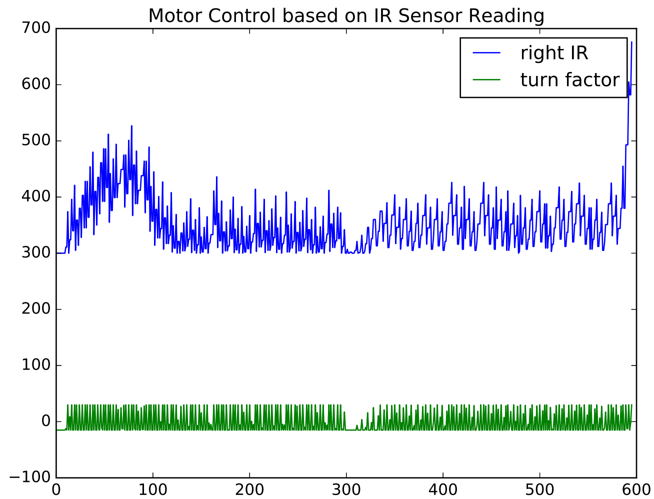

Figure 5: Sensor vs. Motor output graph (run in simulator)

Unlike *Bang-Bang* control, where there are only two possible states in which the car could be in, the PID control can evaluate the error function in continuous space, which tends to result in a lot smoother trajectory.

Our PID controller governed the turning factor of the robot: fixing the forward velocity of the robot essentially constrains the problem such that the control target and the error term collapses into a one-dimensional PID loop for determining sideways motion, i.e.

$$\begin{aligned}
e_{p^t} &= (1+1)\frac{ir_l - 300}{800 - 300} - 1 \\
e_{i^t} &= e_{i^{t-1}} + e_{p^t} dt \\
e_{d^t} &= \frac{e_{p^t} - e_{p^{t-1}}}{dt} \\
e_t &= k_p e_{p^t} + k_i e_{i^t} + k_d e_{d^t} \\
p_l &= p_{fw} + e_t p_{tn} \\
p_r &= p_{fw} - e_t p_{tn}
\end{aligned}$$

Where e, p, l, r, fw, tn, and t represents error, power, left, right, forward-factor, turn-factor, and time, respectively. 300 and 800 are the empirical response readings[1] we have obtained from the Infrared sensor, distinguishing the tape and floor parts.

With untuned PID control trial to roughly verify that the code was working as intended, we obtained a time of 1:13 around the track. It should be noted that during the memorization phase of our final control algorithm, the PID controller (despite frequent stops in order to record the sectional path information) completed the track in less than a minute.

[1] voltage readings that map the ranges 0-255 to 0-5V.

3.3 Memory Prediction

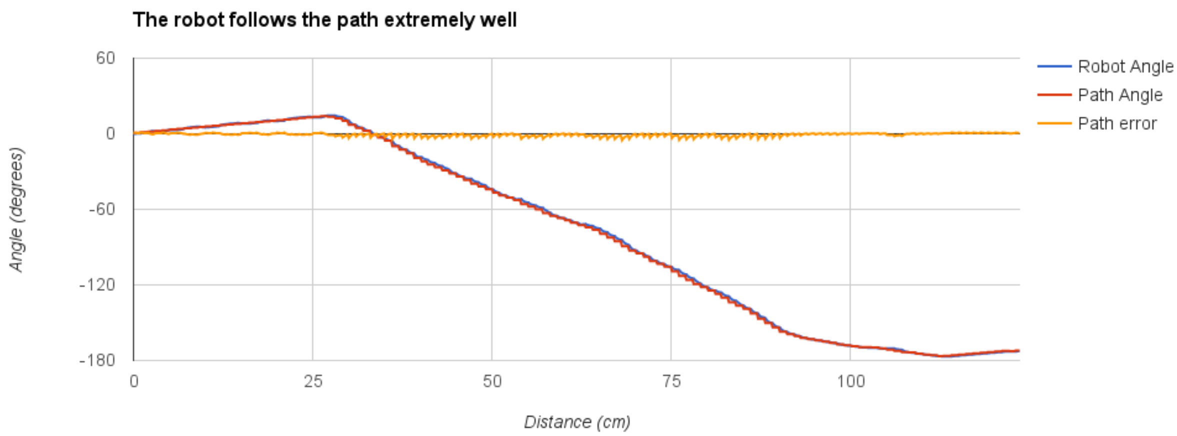

Figure 6: A snapshot of the pose of the robot along a segment in the path during one of the runs; the path is straight in the beginning (nearly constant angle) until a major 180 degree turn from 30cm until 90cm.

While PID is an incredibly powerful tool, its benefits primarily shine in situations where the behavior of the system varies unpredictably with time. At first glance, a line follower seems like a prime example of this because the shape of the upcoming line is unknown, but in reality, the nature of the course as a closed loop that is permanently affixed to the floor means that the path of the line is actually predictable, and that predictability can be harnessed to enhance the performance of the robot significantly. This insight gave rise to our third-generation algorithm, the *memory-prediction follower*, which circumnavigates the course twice, using the first run to collect readings on the precise curvature of the course and the second run to use those readings for improving the speed of the navigation.

Intuitively, this task would be easy if the behavior of the motors was perfectly repeatable from run to run because it would be possible to simply record the outputs on the first run and replay them faster for the second run. In reality, the low-quality DC motors we were provided are not anywhere near perfectly repeatable, as demonstrated in Figure 8, and exhibit variations of up to 10% from predictable behavior. We likely could have decreased that error substantially by incorporating true encoders onto the robot, but we wanted to stay within the letter of the lab description by using only the two IR sensors as input devices.

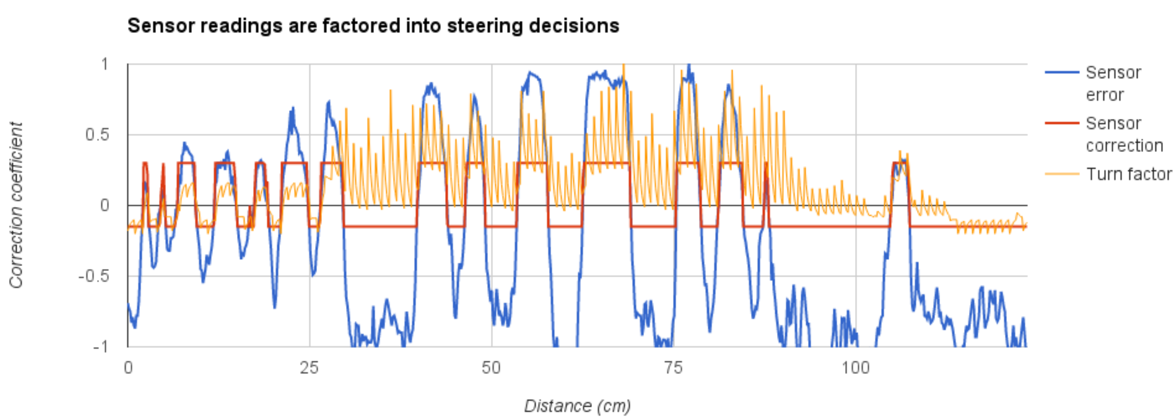

Figure 7: Sensor data is factored into the final steering corrections (shown in orange) but most of the input comes from error relative to the memorized path.

Because the encoder data we are generating from the motor commands is highly inaccurate, we needed to fuse it with traditional line sensor readings for it to be useful. We did this in two ways. One line sensor (after calibration scaling) was used as a source of a PID loop to update the robot's estimated orientation in a way akin to following the line. We made the rates on this loop substantially lower than they would be for a pure sensor-based following solution because we could take advantage of the map data collected on the previous run. We placed the other sensor several inches away, much farther than a normal two-sensor follower, and used it exclusively to detect abrupt changes in curvature direction. These changes were used to reset the "distance travelled" component of the robot's position estimate. Every time the curvature changed, these two sensors switch roles, guaranteeing that the "following" sensor will always be on the interior (concave) portion of the curving line.

By running this procedure without prior map data (and at low speed) we were able to built a map composed of seven segments that together build up the course. We stored these segments as lists of floating point values representing the estimated angle of the line at each point in relation to the angle at the start of that segment, with one value stored per centimeter of course. By using separate segments, we allowed ourselves to allocate appropriate amounts of memory and store configuration values associated with each piece of the course. Storing absolute angles instead of rates of curvature was intended to allow efficient computation of drive instructions from any point to any other, although either decision would likely have fallen within our performance requirements.

4 Calibration

4.1 Motor Characterization

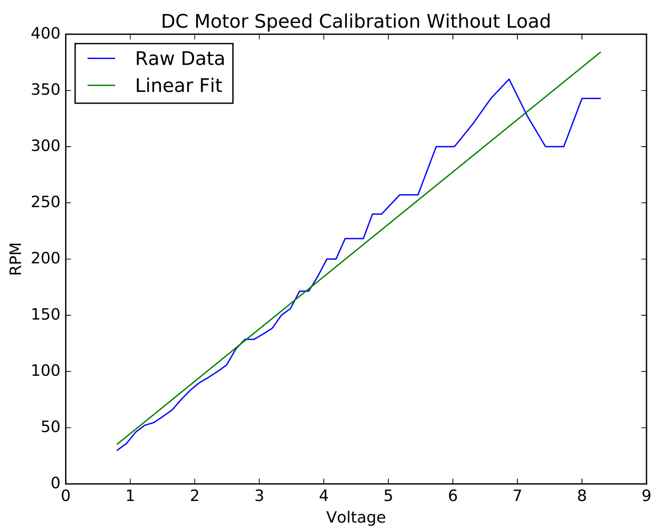

Figure 8: Calibration Curve

In order to obtain better data with dead-reckoning, it was essential to have a reasonably precise estimate of the correlation between the power input to the motors and the output velocity. In order to characterize the motors, Below are the thus obtained relations:

$$\begin{aligned} r &= 3.25cm \\ \omega &= 2\pi(46.55p - 1.80)^{rd}/_{mn} \\ \vec{v} &= \omega r \\ &= 6.5\pi(46.55p - 1.80)^{cm}/_{mn} \end{aligned}$$

This data was obtained by attaching the tape on the shaft of the motors and counting the number of times that a tape would pass through a certain point in space within a defined amount of time; in practice, this was done by simply placing a finger in the trajectory of the tape and counting the number of collisions over a minute.

4.2 IR Reflectance Sensor Calibration

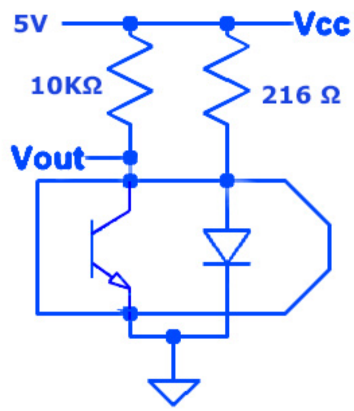

Figure 9: Final schematics for interfacing with the IR Reflectance Sensor

The calibration process for the IR reflectance sensor was more of a guess-and-check binary search for the right resistor value that can distinguish between the reflectance responses from the floor and the tape. In order to smooth out the possible noise, we polled the readings for a set duration of the during the loop and updated the values at 10 milliseconds, which was significant enough to cancel the noise but small enough to facilitate rapid reaction.

5 Simulation

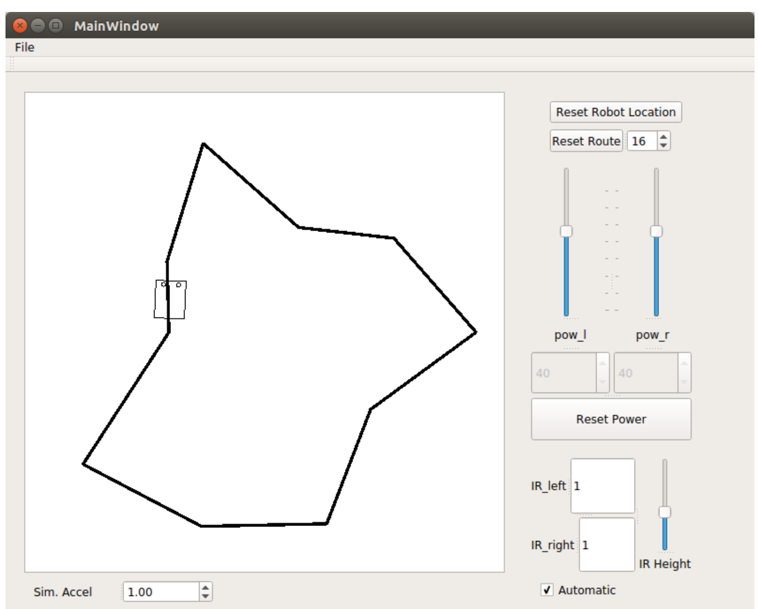

Figure 10: Snapshot of the simulation during one of its autonomous runs

Initially, we developed the simulation as test bench for the software code before the chassis arrived for hardware integration testing. However, the process was delayed such that it was completed around the same time as the robot. At this point, we sought to use the simulator to fine-tune the k_p, k_i, and k_d parameters, as well as the offset and the position of the IR sensors, but it quickly turned out that adjusting those parameters didn't take much effort.

What we ultimately ended up doing is to use the simulator as a relatively idealized environment, in parallel to the physical interaction of the robot with the actual course, where the inherent environmental noise and hardware limitations do not affect the logics of the code. Whereas we did not have time to actually utilize its powers as an optimization platform, it would be a reasonable next step to do so.

5.1 Kinematics

In the differential drive robot, the velocities of the two motor are responsible for dictating the motion of the robot; i.e. the two signals can be mixed to form the next-state estimate as follows:

The instantaneous center of curvature, denoted ICC, is the imaginary center of rotation along which the robot will rotate. it is found by[2]:

$$ICC = (x - Rsin\theta, y + Rcons\theta)$$

thus, at $t = t + \delta t$,

$$\begin{bmatrix} x' \\ y' \\ \theta' \end{bmatrix} = \begin{bmatrix} cos(w\delta t) & -sin(w\delta t) & 0 \\ sin(w\delta t) & cos(w\delta t) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x - ICC_x \\ y - ICC_y \\ \theta \end{bmatrix} + \begin{bmatrix} ICC_x \\ ICC_y \\ w\delta t \end{bmatrix}$$

6 Results & Reflection

Our robot, on its memorized run, was able to go through the course in under 30 seconds. [Demo Video]

When we were electing to broaden the scope of this project such that the robot would take advantage of accumulated knowledge of the course from history, we knew we were taking on a task that was potentially impossible, and certainly very hard. Although we probably over-scoped the problem some, particularly by refusing to use encoders, we are proud we managed to make it work at all, let alone with the speed we eventually demonstrated.

Our robot's coding strategy is more prone to external disturbances as a result of high reliance on odometry, but it has the potential of a far higher maximum speed: its anticipation of future positions and low reliance[3] on IR sensor readings on its memorized run removes the dependency on the rate of the control loop, and could potentially allow it to perform at or near the limit of motor speed. With a few more sensors, we believe a map-based approach like ours would demonstrate clear superiority, and with one sensor, PID-like (stateless) approaches are probably best, but with the middle-ground of two line sensors, it isn't clear which approach is superior.

[2] Refer to [this document] for more detailed derivation.
[3] around 10%

7 Appendix: Code

7.1 Memory-Prediction Implementation

Main code (.ino)

```
/*
* Version 2 Arduino code for Lab 3 of POE Fall 2016 as taught at Olin College
* This code attempts to follow the line while recording (and eventually being able to play back) a motion path.
*
* Authors: Eric Miller (eric@legoaces.org) and Jamie Cho (yoonyoung.cho@students.olin.edu)
*/

#include <stdlib.h>

// Imports for Motor Shield, taken from https://learn.adafruit.com/adafruit-motor-shield-v2-for-arduino/using-dc-motors
#include <Wire.h>
#include <Adafruit_MotorShield.h>
#include "utility/Adafruit_MS_PWMServoDriver.h"

// Import of PID library for training run, see libraries.md for download instructions
#include <PID_v1.h>

// Setup state machine for robot
byte state = 1;
const byte STATE_STOP = 0;
const byte STATE_MEMORIZE = 1;
const byte STATE_REPLAY = 2;

// Include path management code
#include "odometry.h"
#include "paths.h"

// Controlling constants

const int LOOP_DURATION = 10; //(ms) This is the inverse of the main loop frequency

const int FORWARD_POWER_INITIAL = 30; // 0...255
const int TURN_POWER_INITIAL = 30; // 0...255

const float OUTER_TURN_LIMIT = 0.05;

const int POWER_REPLAY = 40; // 0...255

const double PATH_STEERING_RATE = .20; // Measured in fraction / degree, path-based replay steering constant.

const byte SMOOTHING_LENGTH = 4;

const double LINE_ANGLE_ADJUSTMENT_RATE_AWAY = 300.0/1000; // Measured in degrees per ms, maximum line-based odometry adjustment factor right.
const double LINE_ANGLE_ADJUSTMENT_RATE_TOWARD = 150.0/1000; // Measured in degrees per ms, maximum line-based odometry adjustment factor right.

const int MIN_SENSOR_LEFT   = 360;
const int MAX_SENSOR_LEFT   = 780;
const int MIN_SENSOR_RIGHT  = 600;
const int MAX_SENSOR_RIGHT  = 880;

// Pin setup (must match hardware)
const byte leftSensorPin  = A0;
const byte rightSensorPin = A1;

Adafruit_MotorShield AFMS = Adafruit_MotorShield();
Adafruit_DCMotor *leftMotor  = AFMS.getMotor(1);
Adafruit_DCMotor *rightMotor = AFMS.getMotor(2);
```

```

// Global variable setup (things that change each loop)
long lastActionTime;


int leftPower = 0, rightPower = 0; // range -255...255

// Declare and allocate robot pose and paths

Pose robotPose;

Path path1(200, false);
Path path2(100, true);
Path path3(600, false);
Path path4(150, true);
Path path5(200, false);
Path path6(100, true);
Path path7(100, false);

Path *paths[] = {&path1, &path2, &path3, &path4, &path5, &path6, &path7};
const byte numPaths = 7;
byte currentPathId = 0;

Path *currentPath = paths[currentPathId];

// Setup PID controller
double PIDerror=0, PIDsetpoint=0, PIDoutput;
double kp=1,ki=0,kd=0;

PID pid(&PIDerror, &PIDoutput, &PIDsetpoint, kp, ki, kd, DIRECT);

void setup()
{
  AFMS.begin();

  // Wait one second before running to allow the user to get their hand
  // out of the robot.
  stop();
  delay(1000);

  // Note the high baud rate to allow Serial.print() statements to run non-blocking within
      10ms
  Serial.begin(57600);

  lastActionTime = millis();

  // Initialize PID controller parameters
  pid.SetMode(AUTOMATIC);
  pid.SetSampleTime(LOOP_DURATION - 2);
  pid.SetOutputLimits(-1, 1);

  // Reset robot pose for beginning of training run
  robotPose.reset();
}

// Tracking variables for IR averaging to allow maximally smooth data collection
// As many readings as possible over <10ms are averaged before any other processing is done
long totalLeft = 0;
long totalRight = 0;
int count = 0;

// Used to moderate debugging printouts with modulo.
int loopCount = 0;

void loop()
{
  // Check for newly-set PID parameters on serial
  handleIncomingSerial();
```

```
  // Update tracking variables of IR sensor
  int leftRead = 0;
  int rightRead = 0;
  getMeasurements(&leftRead, &rightRead);

  totalLeft += leftRead;
  totalRight += rightRead;
  count++;

  // Every (configurable) milliseconds, average together the readings recieved and handle them
  int dt = millis() - lastActionTime;

  if (dt > LOOP_DURATION) {
    float leftAvg = float(totalLeft) / count;
    float rightAvg = float(totalRight) / count;

    // Update the estimated position of the robot based on currently commanded powers
    // In the future, this could be stateful and involve other sensors.
    robotPose.odometryUpdate(leftPower, rightPower, dt);


    // Main state machine
    if(state == STATE_MEMORIZE){
      //Serial.println(lineOffset(leftAvg, rightAvg));
      memorizeLine(leftAvg, rightAvg);

      if(state == STATE_REPLAY){
        // The memorizeLine function decided to change to STATE_REPLAY
        // Transition from STATE_MEMORIZE to STATE_REPLAY

        Serial.println("finished course, replaying.");

        stop();

        for(int i=0; i<numPaths; i++){
          paths[i]->smooth(SMOOTHING_LENGTH);

          Serial.println(i);
          paths[i]->writeOut();
        }

        // This is blocking code, but the Serial printing above is inherently blocking,
        // so there is nothing we can do to prevent it. This adds an aesthetically nice pause,
        // and makes any bugs caused by blocking behavior more obvious so they can be fixed.
        delay(3000);

        // Reset to the first path
        currentPathId = 0;
        currentPath = paths[0];

        // Reset the estimated robot position
        robotPose.reset();
      }

    }else if(state == STATE_REPLAY){

      replayLine(leftAvg, rightAvg, dt);

      if(loopCount % 50 == 0){
        writePoseSerial();
      }

    }else{
      stop();
    }
```

```

    // Functions above write requested powers to global variables,
    // driveMotors() reads, processes, and outputs those.
    driveMotors();


    // Check to see wheter the loop is running as fast as desired, and throw warnings if it isn't.
    // This code intentionally refuses to run every 10th loop to prevent feeding badly where
    // printing the warning causes excessive loop time in a self-perpetuating loop.
    if(dt > LOOP_DURATION * 2 && loopCount % 10){
        Serial.print("WARNING: Main loop running too slow: ");
        Serial.println(dt);
    }

    // Reset sensor tracking variables
    totalRight = totalLeft = 0;
    count = 0;
    loopCount++;

    // This formulation attempts to ensure average loop duration is LOOP_DURATION,
    // without causing hyperactive behavior if the code is running slower than expected.

    // In particular, this won't drift at all unless a loop takes more than 10ms,
    // and won't ever run multiple loops in a row with less than 10ms spacing.
    lastActionTime = lastActionTime + LOOP_DURATION*int(dt / LOOP_DURATION);

    // If something messed up such that this loop took ridiculously long, prevent
    // massive values of dt next time through the loop.
    // This happens during state transitions sometimes, especially if lots of serial data is being printed
    if (millis() - lastActionTime > 500) {
      lastActionTime = millis();
    }
  }
}

// Using a basic clipped PID controller, memorizes the shape of the course.
void memorizeLine(float leftAvg, float rightAvg)
{
  // Whether the current path curves right or left (preconfigured)
  bool useLeftSensor = currentPath->useLeftSensor;

  // Step 1: compute the error from a simple PID line follower
  float error = lineOffset(leftAvg, rightAvg, useLeftSensor);
  PIDerror = error;

  pid.Compute();

  // whether the robot will turn right or left (positive is right)
  float turnFactor = PIDoutput;

  if (!useLeftSensor){
    turnFactor *= -1;
  }

  // Step 2: constrain that error to make sure the robot doesn't turn (much) toward the line
  if(useLeftSensor){
      turnFactor = min(turnFactor, OUTER_TURN_LIMIT);
  } else {
      turnFactor = max(turnFactor, -OUTER_TURN_LIMIT);
  }

  leftPower = FORWARD_POWER_INITIAL + turnFactor * TURN_POWER_INITIAL;
  rightPower  = FORWARD_POWER_INITIAL - turnFactor * TURN_POWER_INITIAL;


  // Step 3: Record the current motion into the path.
```

```
  currentPath->attemptUpdate( &robotPose );


  // Step 4: determine whether the robot has ended the current segment

  float offReading = lineOffset(leftAvg, rightAvg, !useLeftSensor);

  // Note guards to prevent segments from finishing immediately (less then 5cm)
  // or outlasting their allocated memory.
  if ((offReading > 0.5 && robotPose.distAlong > 5) || robotPose.distAlong > currentPath->allocatedPoints){
    // The robot's off-line sensor has seen a line
    Serial.println("segment end detected");


    stop();
    // delay(500);
    robotPose.reset();

    if(currentPathId < numPaths - 1){
        currentPathId++;
        currentPath = paths[currentPathId];
    }else{
      // trigger state transition
      state = STATE_REPLAY;
    }

  }

  // Print debug information
  if(loopCount % 50 == 0){
    writePoseSerial();
  }
}

// Sets motor values for replaying a recorded path with input from sensor readings leftAvg and rightAvg
void replayLine(float leftAvg, float rightAvg, int dt) {

  // Step 0: Handle exception for control parameters on the 5th segment (id=4)
  double awayAdjustAmount = LINE_ANGLE_ADJUSTMENT_RATE_AWAY;
  if(currentPathId == 4){
      awayAdjustAmount *= 1.8;
  }

  bool useLeftSensor = currentPath->useLeftSensor;

  // Step 1: adjust current odometry estimate on the basis of sensor readings.
  // Note that this is clipped such that it will adjust dramatically to avoid
  // a line and gradually to find it again.
  // This prevents errors caused by the asymmetry of one-sensor following.

  double lineError = lineOffset(leftAvg, rightAvg, useLeftSensor);

  double lineCorrection = max(awayAdjustAmount, LINE_ANGLE_ADJUSTMENT_RATE_TOWARD)
              * dt * lineError;

  lineCorrection = constrain(lineCorrection, -LINE_ANGLE_ADJUSTMENT_RATE_TOWARD, awayAdjustAmount);

  robotPose.angleFrom += lineCorrection;


  // Step 2: Drive the robot to follow the recorded path

  PathPoint *target = currentPath->getPoint(robotPose.distAlong);

  // error is positive if the path is left of the robot
```

```
  double pathError = target->wrappedAngle - robotPose.angleFrom;

  // whether the robot will turn right or left (positive is right)
  double turnFactor = -pathError * PATH_STEERING_RATE;

  leftPower = POWER_REPLAY * (1 + turnFactor);
  rightPower  = POWER_REPLAY * (1 - turnFactor);

  // Step 3: Determine whether this segment of path is finished

  float offReading = lineOffset(leftAvg, rightAvg, !useLeftSensor);

  if (offReading > 0.5 && robotPose.distAlong > currentPath->usedPoints*0.7){
    // The robot's off-line sensor has seen a line
    Serial.println("segment end detected");

    stop();
    // delay(200);
    robotPose.reset();

    if(currentPathId < numPaths - 1){
        currentPathId++;
        currentPath = paths[currentPathId];
    }else{
      state = STATE_STOP;
    }

  }

  // Print debug information
  if(loopCount % 50 == 0){
      Serial.print("pathError = ");
      Serial.print(pathError);
      Serial.print("\t:\t");
      Serial.println(turnFactor);
  }
}

// normalizePowers ensures that
// abs(*left) < limit and abs(*right) < limit
// while maintaining their ratio.
// Useful for constraining desired speeds to be
// achievable by the motors.
void normalizePowers(int *left, int *right, int limit){
  int maxabs = max(abs(*left), abs(*right));
  if(maxabs > limit)
  {
      *left = (*left * limit) / maxabs;
      *right = (*right * limit) / maxabs;
  }
}

// Returns how much the selected sensor is on the line, with
// -1 reflecting "completely off" and 1 meaning "completely  on"
// Note that the output is not strictly gaurunteed to be in this range.
float lineOffset(float leftAvg, float rightAvg, bool useLeftSensor)
{
  if(useLeftSensor){
    return map(leftAvg, MIN_SENSOR_LEFT, MAX_SENSOR_LEFT, -100, 100) / 100.0;
  } else {
    return map(rightAvg, MIN_SENSOR_RIGHT, MAX_SENSOR_RIGHT, -100, 100) / 100.0;
  }
}

// If there is any data in the serial buffer, attempts to read in that data as new P, I, and D values.
void handleIncomingSerial()
{
```

```
  if(Serial.available() > 0){

    Serial.setTimeout(100);
    // Read first number from serial stream
    kp = Serial.parseFloat();
    Serial.read();
    // Read second number from serial stream
    ki = Serial.parseFloat();
    Serial.read();
    // Read third number form serial stream
    kd = Serial.parseFloat();
    Serial.read();

    // Ingest remainder of serial buffer in case something went wrong
    while(Serial.available()){
        Serial.read();
    }

    pid.SetTunings(kp, ki, kd);
    writeTuningsSerial();
  }
}

void writePoseSerial(){
  Serial.print("Pose is: \t");
  robotPose.writeOut();
  Serial.println();
}

void writeTuningsSerial()
{
  Serial.println("Tunings set to (kp, ki, kd) = ");
  Serial.print("\t(");
  Serial.print(kp);
  Serial.print(", ");
  Serial.print(ki);
  Serial.print(", ");
  Serial.print(kd);
  Serial.print(")\n");
}

void stop(){
  leftPower = 0;
  rightPower = 0;
  driveMotors();
}

void driveMotors(){
  // Inputs leftPower and rightPower vary from -255...255
  // Code in this function is based on https://learn.adafruit.com/adafruit-motor-shield-v2-for-arduino/using-dc-motors

  normalizePowers(&leftPower, &rightPower, 255);

  // For each motor, decide whether to run it FORWARD, BACKWARD, (or RELEASE)
  // These are ternary operators, returning FORWARD if power > 0 and backward otherwise.
  byte leftDirection  = (leftPower  > 0) ? FORWARD : BACKWARD;
  byte rightDirection = (rightPower > 0) ? FORWARD : BACKWARD;

  // Set motor speeds
  leftMotor ->  setSpeed(abs(leftPower));
  rightMotor  ->  setSpeed(abs(rightPower));

  // Set motor directions
  leftMotor ->  run(leftDirection);
  rightMotor  ->  run(rightDirection);

}
```

```
void getMeasurements(int *leftRead, int *rightRead)
{
  *leftRead = analogRead(leftSensorPin);
  *rightRead = analogRead(rightSensorPin);
}
```

Path library (.h)

```
// PathPoint represents the data associated with each point
// along a recorded segment of path.
// Currently this is just an angle because the speed adjustment flags
// needed to be removed before they could be fully implemented.
class PathPoint
{
public:
  double wrappedAngle;

  PathPoint();
};

PathPoint::PathPoint( void ){
  wrappedAngle = 0;
}

// A Path is a collection of PathPoints that represents a segment of the course.
class Path
{
public:
  Path(int length, bool useLeft);

  PathPoint *points;
  int allocatedPoints; // = 0;
  int usedPoints; // = 0;

  bool useLeftSensor; // = false;

  void writeOut( void );
  bool attemptUpdate( Pose *pose );
  void smooth( byte smoothingLength );
  PathPoint *getPoint( double distAlong );
};

// Note that the points array is dynamically allocated to save space.
Path::Path(int length, bool useLeft){
  useLeftSensor = useLeft;

  allocatedPoints = length;

  points = (PathPoint*) malloc(sizeof(PathPoint) * length);

  for (int i = 0; i < allocatedPoints; ++i)
  {
    points[i] = PathPoint();
  }

  points[0].wrappedAngle = 0;
}

PathPoint *Path::getPoint( double distAlong ){
  int index = min(distAlong, usedPoints - 1);
  if (index < 0)
  {
    return NULL;
  }
  return &points[index];
}

```

```
// Updates the path with new information if the information is not currently contained in the path.
bool Path::attemptUpdate( Pose *pose ) {
  if (int(pose->distAlong) > usedPoints && usedPoints < allocatedPoints){
    usedPoints++;
    points[usedPoints].wrappedAngle = pose->angleFrom;
    return true;
  } else {
    return false;
  }
}

// Prints Path data to serial for debugging and analysis.
void Path::writeOut(){
  Serial.println("id\tspeed\tisOffLine\tangle");
  for (int i = 0; i < usedPoints; ++i)
  {
    Serial.print(i);
    Serial.print("\t");
    Serial.print(points[i].wrappedAngle);
    Serial.println();
  }
}

// Runs a n-point leading average to enable both forward-looking controls predictions
// and to smooth out irregular behavior in the training follow.
void Path::smooth(byte smoothingLength){
  for (int i = 0; i < usedPoints - smoothingLength; ++i)
  {
    double total = 0;
    for (int j = 0; j < smoothingLength; ++j)
    {
      double point = points[i+j].wrappedAngle;

      total += point;
    }
    double average = total / smoothingLength;

    points[i].wrappedAngle = average;
  }
}
```

Odometry code (.h)

```
#include <Arduino.h>

// This file attempts to estimate the ego-motion of the robot
// using the commanded powers of the motors. Expect accuracy of += 15%

class Pose
{
public:
  Pose();
  void odometryUpdate(int leftPower, int rightPower, int timestep);
  void writeOut( void );
  void reset();
  // Measured in cm, with positive being forward of the starting position
  double distAlong;
  // Measured in degrees, with positive being nose left.
  double angleFrom;
};

Pose::Pose() {
  reset();
}

void Pose::writeOut( void ) {
  Serial.print("(d, theta) = (");
```

```
  Serial.print(distAlong);
  Serial.print(", ");
  Serial.print(angleFrom);
  Serial.print(")");
}

void Pose::reset(){
  distAlong = 0;
  angleFrom = 0;
}

// Neither of these constants matter much as long as they do not
// change between recording and playback.
// MAX_FORWARD_SPEED is scaled to approximately 1 = 1cm on our robot,
// while MAX_TURN_SPEED is approximately 1 = 1/10 degree.
const double MAX_FORWARD_SPEED = 100;
const double MAX_TURN_SPEED = 1000;

double adjustPower( int power ){
  double powerfrac = power / 255.0;
  return powerfrac - .05 * powerfrac * powerfrac;
}

// Update the estimated pose of the robot based on wheel (estimated) odometry.
// Currently uses an almost linear approximation, with slight falloff at high powers.
// timestep is in ms
void Pose::odometryUpdate( int leftPower, int rightPower, int timestep) {
  if (timestep > 100)
  {
    // Prevent large timesteps from causing jumps in odometry readings (for example, on
        first boot)
    return;
  }

  double leftPowerFrac  = adjustPower(leftPower);
  double rightPowerFrac = adjustPower(rightPower);
  double forwardSpeed = (leftPowerFrac + rightPowerFrac) / 2 * MAX_FORWARD_SPEED;
  double turnSpeed = (rightPowerFrac - leftPowerFrac) / 2 * MAX_TURN_SPEED;

  distAlong += forwardSpeed * timestep / 1000.0;
  angleFrom += turnSpeed * timestep / 1000.0;
}
```

7.2 Simulator Implementation

The simulator is a relatively large project with a lot of split definitions, so including all of them here would be impractical. Below are several more relevant parts that are directly related to either the computations or the functions of the simulator. For the full documentation, please visit the github link.

7.2.1 Arduino Interface

Adafruit_MotorShield.h (declaration)

```
#ifndef ADAFRUIT_MOTORSHIELD_H
#define ADAFRUIT_MOTORSHIELD_H

// This class emulates the behavior of the Adafruit Motor Shield V2 for the Arduino.

#include "utils.h"
#include "arduino.h"
#include "robot.h"

#define FORWARD 1

#define BACKWARD 2
```

```cpp

class Adafruit_DCMotor{
private:
    int pin;
    int speed;
public:
    Adafruit_DCMotor(int pin);
    void setSpeed(int);
    void run(byte&);
};

class Adafruit_MotorShield
{
private:
    Adafruit_DCMotor *motor_left, *motor_right;
public:
    Adafruit_MotorShield();
    ~Adafruit_MotorShield();
    void begin();
    Adafruit_DCMotor* getMotor(int);
};

#endif // ADAFRUIT_MOTORSHIELD_H
```

Adafruit_MotorShield.cpp (definition)

```cpp
#include "Adafruit_MotorShield.h"

// function definitions for the emulation of the Adafruit Motor Shield V2
// just to have it interface with the "virtual arduino"

Adafruit_DCMotor::Adafruit_DCMotor(int pin):pin(pin){

}

void Adafruit_DCMotor::run(byte & direction){
    float dir;
    if(direction == FORWARD){
        dir = 1;
    }else if (direction == BACKWARD){
        dir = -1;
    }

    if(robot){
        switch(pin){
        case 1: //left
            robot->setPowerL(dir * speed);
            break;
        case 2:
            robot->setPowerR(dir * speed);
            break;
        }
    }

}

void Adafruit_DCMotor::setSpeed(int speed){
    this->speed = speed;
}

Adafruit_MotorShield::Adafruit_MotorShield()
{
    this->motor_left = new Adafruit_DCMotor(1);
    this->motor_right = new Adafruit_DCMotor(2);
}

Adafruit_MotorShield::~Adafruit_MotorShield(){
    if(motor_left){
```

```
        delete motor_left;
        motor_left = nullptr;
    }
    if(motor_right){
        delete motor_right;
        motor_right = nullptr;
    }
}

void Adafruit_MotorShield::begin(){
    // don't need to do anything here
}

Adafruit_DCMotor* Adafruit_MotorShield::getMotor(int id){
    switch(id){
    case 1:
        return this->motor_left;
        break;
    case 2:
        return this->motor_right;
        break;
    default:
        return nullptr;
    }
}
```

arduino.h (declaration)

```
#ifndef ARDUINO_H
#define ARDUINO_H

// arduino.h essentially enables native arduino code to run seamlessly with the application.

#define BIN (2)

const int A0 = 0xA0;
const int A1 = 0xA1;

#include <iostream>
#include <bitset>

#include "utils.h"
#include "mainwindow.h"
#include "robot.h"

class _Serial{

public:
    void begin(int){
        // ignore baud rate
    }

    // all prints are redirected to stdout
    template<typename T>
    void print(T val){
        std::cout << val;
    }

    template<typename T>
    void print(T val, int flag){
        if(flag == BIN){
            std::cout << std::bitset<8>(val);
        }
        // ... not handling other flags yet
    }

    template<typename T>
    void println(T val){
```

```
        std::cout << val << std::endl;
    }

    void println(){
        std::cout << std::endl;
    }
};

extern _Serial Serial;
typedef unsigned char byte;

// Arduino APIs

extern long long millis();
extern int analogRead(const int pin);
extern void setup();
extern void loop();
extern void delay(int);

#endif // ARDUINO_H
```

arduino.cpp (definition)

```
#ifndef ARDUINO_H
#define ARDUINO_H

// arduino.h essentially enables native arduino code to run seamlessly with the application.

#define BIN (2)

const int A0 = 0xA0;
const int A1 = 0xA1;

#include <iostream>
#include <bitset>

#include "utils.h"
#include "mainwindow.h"
#include "robot.h"

class _Serial{

public:
    void begin(int){
        // ignore baud rate
    }

    // all prints are redirected to stdout
    template<typename T>
    void print(T val){
        std::cout << val;
    }

    template<typename T>
    void print(T val, int flag){
        if(flag == BIN){
            std::cout << std::bitset<8>(val);
        }
        // ... not handling other flags yet
    }

    template<typename T>
    void println(T val){
        std::cout << val << std::endl;
    }

    void println(){
        std::cout << std::endl;
```

```
    }
};

extern _Serial Serial;
typedef unsigned char byte;

// Arduino APIs

extern long long millis();
extern int analogRead(const int pin);
extern void setup();
extern void loop();
extern void delay(int);

#endif // ARDUINO_H
```

7.2.2 System Definition

Route.h (declaration)

```
#ifndef ROUTE_H
#define ROUTE_H

#include <QVector>
#include <QPointF>
#include <QPolygonF>
#include <QGraphicsPolygonItem>
#include <QPen>
#include <QString>
#include <QFile>
#include <QTextStream>

#include "utils.h"

struct Route
{
    //QVector<QPointF> route;
    QPolygonF poly;
    QGraphicsPolygonItem poly_item;
    QPen routePen;
public:
    Route();

    void draw();
    void reset(int n); //randomize to length
    void reset(const QVector<QPointF>& route);

    void save(const QString& filename);
    void load(const QString& filename);
};

extern Route* route;

#endif // ROUTE_H
```

Route.cpp (definition)

```
#include "route.h"
#include <random>
#include <iostream>

const double R_MIN = PXL_DIMS/8;
const double R_MAX = PXL_DIMS/2;

Route* route;

Route::Route()
```

```
{
    QBrush routeBrush = QBrush(QColor::fromRgb(0,0,0),Qt::SolidPattern);
    routePen = QPen(routeBrush, c2p(1.7));
}

void Route::reset(int n){
    // reinitialize to a route of n points

    QVector<QPointF> route;

    float x = R_MAX;
    float y = R_MAX; // at center

    for(int i=0; i<n; ++i){
        float t = (2 * M_PI) * i / n;
        float r = R_MIN + (R_MAX - R_MIN) * float(rand())/RAND_MAX;
        route.push_back(QPointF(x + r*cos(t), y + r*sin(t)));
    }
    std::cout << "N : " << n << " ROUTE SIZE : " << route.size() << std::endl;
    reset(route);
}

void Route::reset(const QVector<QPointF> &route){
    // reset from vector
    poly = QPolygonF(route);
    poly_item.setPolygon(poly);
    poly_item.setPen(routePen);
}

void Route::save(const QString& filename){
    // save route to file
    QFile file(filename);
    if (file.open(QIODevice::ReadWrite)) {
        QTextStream stream(&file);

        for(auto& p : poly){
            stream << p.x() << ',' << p.y() << '\n';
        }
    }
    file.close();
}

void Route::load(const QString& filename){
    // load route from file
    QFile file(filename);
    if(file.open(QIODevice::ReadOnly)) {
        QTextStream stream(&file);
        QVector<QPointF> route;

        while(!stream.atEnd()){
            QString line = stream.readLine();
            QStringList pt_s = line.split(",");
            QPointF pt(pt_s[0].toFloat(),pt_s[1].toFloat());
            route.push_back(pt);
        }

        file.close();
        Route::reset(route);
    }
}
```

Robot.h (declaration)

```
#ifndef ROBOT_H
#define ROBOT_H

#include <QObject>
#include <QPointF>
```

```

#include <QPolygonF>

#include <QGraphicsRectItem>
#include <QGraphicsEllipseItem>

#include <QGraphicsScene>

#include "utils.h"
#include "robotitem.h"


class Robot
{

public:
    QPointF pos; // current position
    float theta; // heading, measured from horz.  radians

    QPointF irOffset; // position of ir from center

    float vel_l, vel_r; //left-right velocity of motors

    float h; // height of IR sensor, inches
    float fov; //field of view of IR sensor, radians

    float ir_val_l; //value of ir reflectance sensors
    float ir_val_r;

    // frequently computed values
    float cr; //cone radius

    RobotItem* body;

public:
    Robot(QGraphicsScene& scene,
          QPointF pos, float theta,
          QPointF irOffset, float h, float fov);
    ~Robot();

    void reset(QPointF pos, float theta);
    void reset(QPolygonF route);

    void update();
    void move(float delta, float dtheta);
    void setVelocity(float left, float right);

    void setPowerR(int r);
    void setPowerL(int l);

    void setVelocityR(float r);
    void setVelocityL(float l);

    void setIRHeight(float h);

    void sense(QImage& img);
    void setVisible(bool visible);
};

extern Robot* robot;

#endif // ROBOT_H
```

Robot.cpp (definition)

```
#include "utils.h"
#include "robot.h"

```

```cpp
#include <iostream>

// 1 pxl = .5 cm
// 270 x 270 cm world

// robot = 20x16 cm

Robot* robot = nullptr;

float coneRadius(float h, float fov){
    return h*tan(fov/2);
}

// RobotBody
Robot::Robot(QGraphicsScene& scene, QPointF pos, float theta, QPointF irOffset, float h,
    float fov):
    pos(pos),
    theta(theta),
    irOffset(irOffset),
    h(h),
    fov(fov),
    cr(coneRadius(h,fov))
{
    vel_l = vel_r = 0.;

    body = new RobotItem(pos,QPointF(ROBOT_LENGTH,ROBOT_WIDTH),irOffset, theta, cr);
    // RobotItem will take care of graphics
    scene.addItem(body);
}

Robot::~Robot(){
    if(body){
        delete body;
        body = nullptr;
    }
}

void Robot::reset(QPointF p, float t){
    pos=p; theta=t;
    body->setPos(pos,theta);
}

void Robot::reset(QPolygonF route){
    pos = route.front();
    float dst_x = route[1].x();
    float dst_y = route[1].y();

    theta = atan2(pos.y() - dst_y,dst_x - pos.x());

}

void Robot::move(float delta, float dtheta){
    theta += dtheta * DT;
    pos += delta * QPointF(cos(theta), -sin(theta)) * DT;
    update();
}

void Robot::update(){
    float w = (vel_r - vel_l) / WHEEL_DISTANCE;
    if(w != 0){
        float R = (vel_l + vel_r) / (2*w);

        QPointF ICC = pos + R * QPointF(-sin(theta), -cos(theta));
        // ICC = virtual center of rotation
        // update position based on obtained kinematics prediction

        float x = pos.x();
        float y = pos.y();
```

```
        float iccx = ICC.x();
        float iccy = ICC.y();

        pos.setX(
                    cos(-w*DT) * (x - iccx) +
                    -sin(-w*DT) * (y - iccy) +
                    iccx
                    );
        pos.setY(
                    sin(-w*DT) * (x - iccx) +
                    cos(-w*DT) * (y - iccy) +
                    iccy
                    );
        theta += w * DT;
    }else{
        // if w == 0, then division by zero would be bad..
        // since it's a special (and well-defined) case, we should handle this
        pos += vel_l * QPointF(cos(theta), -sin(theta)) * DT;
    }


    body->setPos(pos, theta);
}
void Robot::setVelocityL(float v){
    vel_l = v;
}

void Robot::setVelocityR(float v){
    vel_r = v;
}

void Robot::setPowerL(int pow){
    // convert power to velocity
    // then convert it back to pixel units
    setVelocityL(c2p(pow2vel(pow)));
}


void Robot::setPowerR(int pow){
    // convert power to velocity
    // then convert it back to pixel units
    setVelocityR(c2p(pow2vel(pow)));
}


void Robot::setVelocity(float l, float r){
    setVelocityL(l);
    setVelocityR(r);
}

void Robot::setIRHeight(float h){
    this->h = h;
    this->cr = coneRadius(h,fov);
    body->setCR(this->cr);
}

void Robot::sense(QImage& image){
    // poll a conical section of the ground-plane beneath the IR sensors
    // and average out the readings of pixels

    float l_1 = irOffset.x();
    float l_2 = irOffset.y();

    QPointF ir_root = pos + QPointF(l_1 * cos(theta), -l_1 * sin(theta));
    QPointF ir_l = ir_root - QPointF(l_2 * sin(theta), l_2 * cos(theta));
    QPointF ir_r = ir_root + QPointF(l_2 * sin(theta), l_2 * cos(theta));

    float cR = coneRadius(h,fov);
```

```
    int i_cR = round(cR);

    int n = 0;
    float sum_l=0;
    float sum_r = 0;

    for(int offsetX = -i_cR; offsetX <= i_cR; ++offsetX){
        for(int offsetY = -i_cR; offsetY <= i_cR; ++offsetY){
            if((offsetX * offsetX + offsetY * offsetY) < (cR * cR)){
                QPointF offset(offsetX, offsetY);

                QRgb col_l = image.pixel((ir_l + offset).toPoint());
                QRgb col_r = image.pixel((ir_r + offset).toPoint());

                sum_l += qGray(col_l) / 255.0;
                sum_r += qGray(col_r) / 255.0;
                ++n;


            }
        }
    }

    // set ir values
    if(n){
        ir_val_l = sum_l / n;
        ir_val_r = sum_r / n;
    }
}

void Robot::setVisible(bool visible){
    body->setVisible(visible);
}
```