

# PoE Lab 03 : Line-Following Robot

Yoonyoung Cho, Eric Miller

October 13 2016

## 1 Introduction

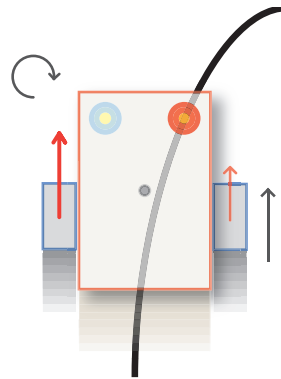


Figure 1: Our Line Follower architecture, a two-wheeled differential drive robot equipped with two IR reflectance sensors positioned at the front.

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

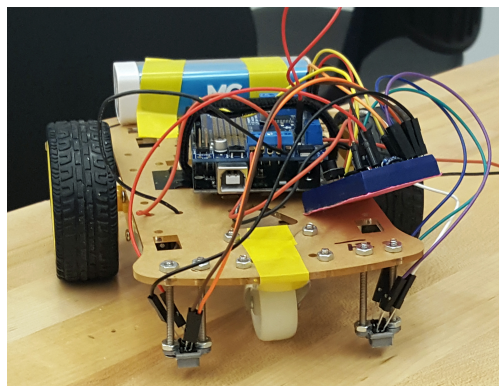


Figure 2: Final assembly of our robot.

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*.

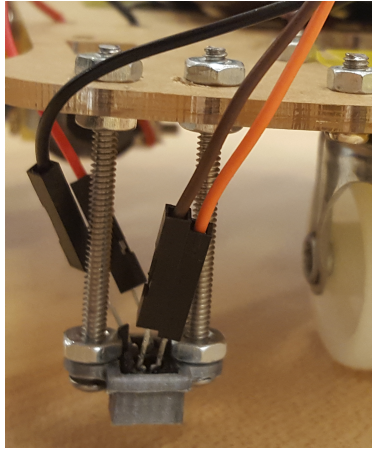


Figure 3: Mount for the IR reflectance sensor.

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

$$r = h \tan(\text{fov}/2)$$

where  $h$  is the height of the IR sensor,  $\text{fov}$  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

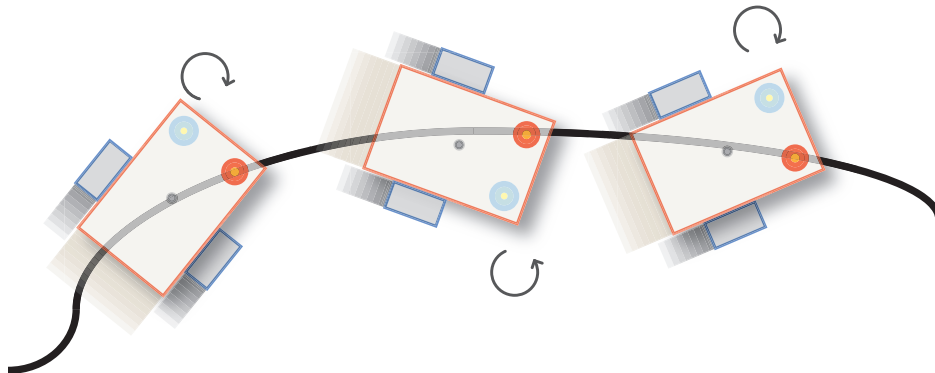


Figure 4: Simplified diagram that illustrates behavior of the *Bang-Bang* controller.

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

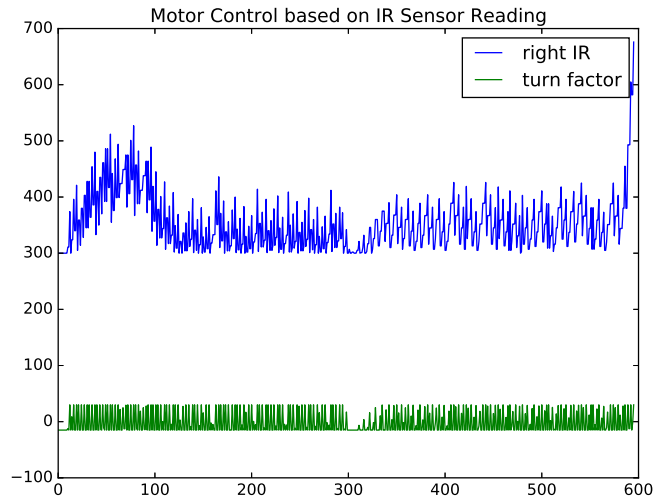


Figure 5: Sensor vs. Motor output graph (run in simulator)

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<sup>1</sup> 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.

<sup>1</sup>voltage readings that map the ranges 0-255 to 0-5V.

### 3.3 Memory Prediction

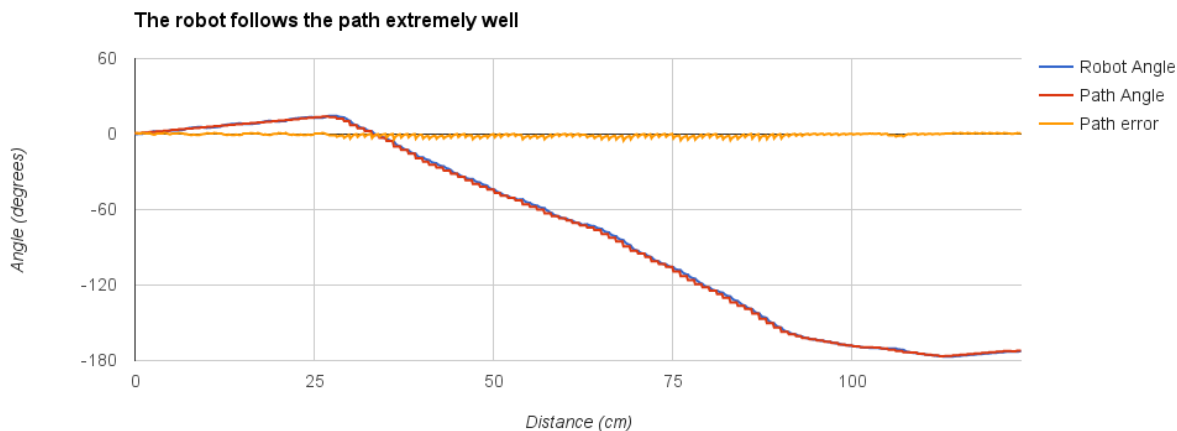


Figure 6: A snapshot of the pose of the robot along a segment in the path during one of the runs; the path is straight in the beginning (nearly constant angle) until a major 180 degree turn from 30cm until 90cm.

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.

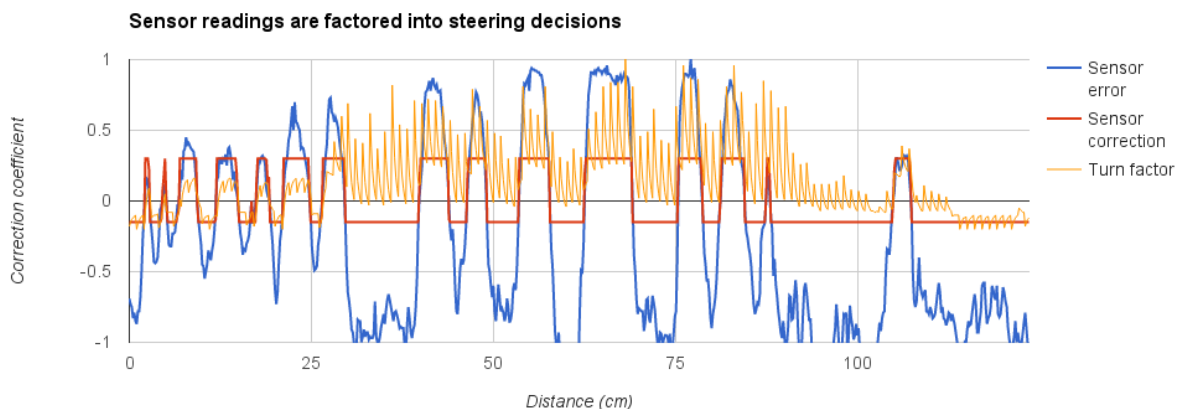


Figure 7: Sensor data is factored into the final steering corrections (shown in orange) but most of the input comes from error relative to the memorized path.

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 build 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

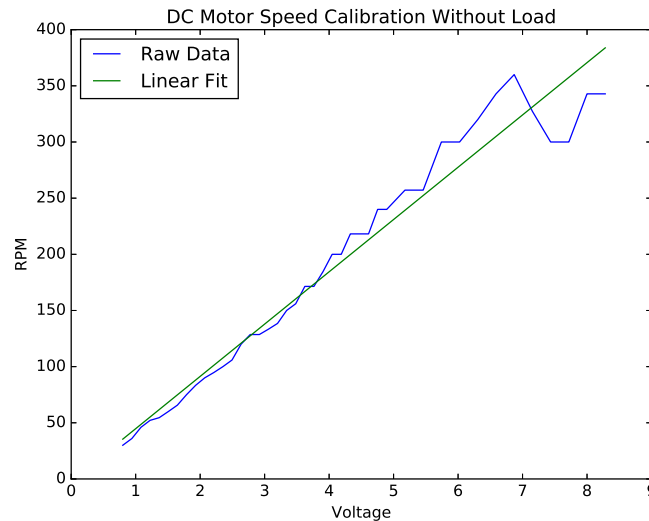


Figure 8: Calibration Curve

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

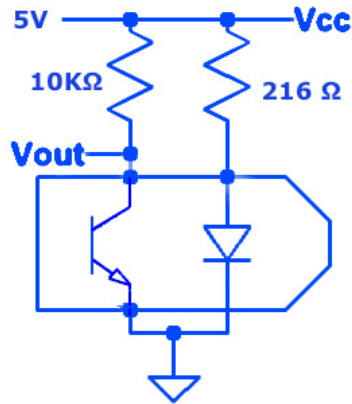


Figure 9: Final schematics for interfacing with the IR Reflectance Sensor

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

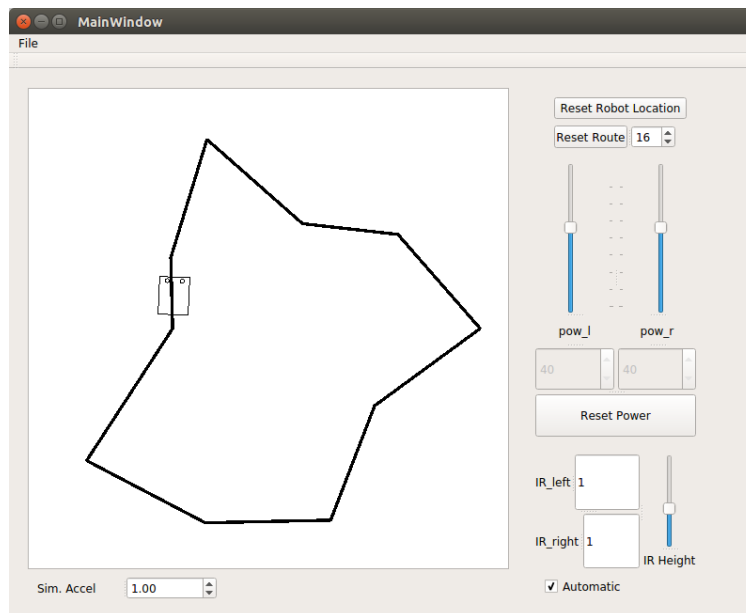


Figure 10: Snapshot of the simulation during one of its autonomous runs

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<sup>2</sup>:

$$ICC = (x - R\sin\theta, y + R\cos\theta)$$

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<sup>3</sup> 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.

---

<sup>2</sup>Refer to [this document] for more detailed derivation.

<sup>3</sup>around 10%

## 7 Appendix: Code

### 7.1 Memory-Prediction Implementation

Main code (.ino)

```
1 /*
2 * Version 2 Arduino code for Lab 3 of POE Fall 2016 as taught at Olin College
3 * This code attempts to follow the line while recording (and eventually being able to play
4 * back) a motion path.
5 *
6 * Authors: Eric Miller (eric@legoaces.org) and Jamie Cho (yoonyoung.cho@students.olin.edu)
7 */
8 #include <stdlib.h>
9
10 // Imports for Motor Shield, taken from https://learn.adafruit.com/adafruit-motor-shield-v2-
11 // for-arduino/using-dc-motors
12 #include <Wire.h>
13 #include <Adafruit_MotorShield.h>
14 #include "utility/Adafruit_MS_PWM_ServoDriver.h"
15
16 // Import of PID library for training run, see libraries.md for download instructions
17 #include <PID_v1.h>
18
19 // Setup state machine for robot
20 byte state = 1;
21 const byte STATE_STOP = 0;
22 const byte STATE_MEMORIZE = 1;
23 const byte STATE_REPLAY = 2;
24
25 // Include path management code
26 #include "odometry.h"
27 #include "paths.h"
28
29 // Controlling constants
30 const int LOOP_DURATION = 10; // (ms) This is the inverse of the main loop frequency
31
32 const int FORWARD_POWER_INITIAL = 30; // 0...255
33 const int TURN_POWER_INITIAL = 30; // 0...255
34
35 const float OUTER_TURN_LIMIT = 0.05;
36
37 const int POWER_REPLAY = 40; // 0...255
38
39 const double PATH_STEERING_RATE = .20; // Measured in fraction / degree, path-based replay
40 // steering constant.
41
42 const byte SMOOTHING_LENGTH = 4;
43
44 const double LINE_ANGLE_ADJUSTMENT_RATE_AWAY = 300.0/1000; // Measured in degrees per ms,
45 // maximum line-based odometry adjustment factor right.
46 const double LINE_ANGLE_ADJUSTMENT_RATE_TOWARD = 150.0/1000; // Measured in degrees per ms,
47 // maximum line-based odometry adjustment factor right.
48
49 const int MIN_SENSOR_LEFT = 360;
50 const int MAX_SENSOR_LEFT = 780;
51 const int MIN_SENSOR_RIGHT = 600;
52 const int MAX_SENSOR_RIGHT = 880;
53
54 // Pin setup (must match hardware)
55 const byte leftSensorPin = A0;
56 const byte rightSensorPin = A1;
57
58 Adafruit_MotorShield AFMS = Adafruit_MotorShield();
59 Adafruit_DCMotor *leftMotor = AFMS.getMotor(1);
60 Adafruit_DCMotor *rightMotor = AFMS.getMotor(2);
```



```

58
59 // Global variable setup (things that change each loop)
60 long lastActionTime;
61
62
63 int leftPower = 0, rightPower = 0; // range -255...255
64
65 // Declare and allocate robot pose and paths
66
67 Pose robotPose;
68
69 Path path1(200, false);
70 Path path2(100, true);
71 Path path3(600, false);
72 Path path4(150, true);
73 Path path5(200, false);
74 Path path6(100, true);
75 Path path7(100, false);
76
77 Path *paths[] = {&path1, &path2, &path3, &path4, &path5, &path6, &path7};
78 const byte numPaths = 7;
79 byte currentPathId = 0;
80
81 Path *currentPath = paths[currentPathId];
82
83 // Setup PID controller
84 double PIDerror=0, PIDsetpoint=0, PIDoutput;
85 double kp=1,ki=0,kd=0;
86
87 PID pid(&PIDerror, &PIDoutput, &PIDsetpoint, kp, ki, kd, DIRECT);
88
89 void setup()
90 {
91   AFMS.begin();
92
93   // Wait one second before running to allow the user to get their hand
94   // out of the robot.
95   stop();
96   delay(1000);
97
98   // Note the high baud rate to allow Serial.print() statements to run non-blocking within
99   // 10ms
100  Serial.begin(57600);
101
102  lastActionTime = millis();
103
104  // Initialize PID controller parameters
105  pid.SetMode(AUTOMATIC);
106  pid.SetSampleTime(LOOP_DURATION - 2);
107  pid.SetOutputLimits(-1, 1);
108
109  // Reset robot pose for beginning of training run
110  robotPose.reset();
111 }
112
113 // Tracking variables for IR averaging to allow maximally smooth data collection
114 // As many readings as possible over <10ms are averaged before any other processing is done
115 long totalLeft = 0;
116 long totalRight = 0;
117 int count = 0;
118
119 // Used to moderate debugging printouts with modulo.
120 int loopCount = 0;
121
122 void loop()
123 {
124   // Check for newly-set PID parameters on serial
125   handleIncomingSerial();

```

```

125
126 // Update tracking variables of IR sensor
127 int leftRead = 0;
128 int rightRead = 0;
129 getMeasurements(&leftRead , &rightRead);
130
131 totalLeft += leftRead;
132 totalRight += rightRead;
133 count++;
134
135 // Every (configurable) milliseconds, average together the readings recieved and handle
    them
136 int dt = millis() - lastActionTime;
137
138 if (dt > LOOP_DURATION) {
139     float leftAvg = float(totalLeft) / count;
140     float rightAvg = float(totalRight) / count;
141
142     // Update the estimated position of the robot based on currently commanded powers
143     // In the future, this could be stateful and involve other sensors.
144     robotPose.odometryUpdate(leftPower, rightPower, dt);
145
146
147 // Main state machine
148 if(state == STATE_MEMORIZE){
149     //Serial.println(lineOffset(leftAvg, rightAvg));
150     memorizeLine(leftAvg, rightAvg);
151
152     if(state == STATE_REPLAY){
153         // The memorizeLine function decided to change to STATE_REPLAY
154         // Transition from STATE_MEMORIZE to STATE_REPLAY
155
156         Serial.println("finished course, replaying.");
157
158         stop();
159
160         for(int i=0; i<numPaths; i++){
161             paths[i]->smooth(SMOOTHING_LENGTH);
162
163             Serial.println(i);
164             paths[i]->writeOut();
165         }
166
167         // This is blocking code, but the Serial printing above is inherently blocking,
168         // so there is nothing we can do to prevent it. This adds an aesthetically nice
            pause,
169         // and makes any bugs caused by blocking behavior more obvious so they can be fixed.
170         delay(3000);
171
172         // Reset to the first path
173         currentPathId = 0;
174         currentPath = paths[0];
175
176         // Reset the estimated robot position
177         robotPose.reset();
178     }
179
180 }else if(state == STATE_REPLAY){
181
182     replayLine(leftAvg, rightAvg, dt);
183
184     if(loopCount % 50 == 0){
185         writePoseSerial();
186     }
187
188 }else{
189     stop();
190 }

```

```

191 // Functions above write requested powers to global variables ,
192 // driveMotors() reads, processes, and outputs those.
193 driveMotors();
194
195
196
197 // Check to see wheter the loop is running as fast as desired, and throw warnings if it
198 // isn't.
199 // This code intentionally refuses to run every 10th loop to prevent feeding badly where
200 // printing the warning causes excessive loop time in a self-perpetuating loop.
201 if(dt > LOOP_DURATION * 2 && loopCount % 10){
202     Serial.print("WARNING: Main loop running too slow: ");
203     Serial.println(dt);
204 }
205
206 // Reset sensor tracking variables
207 totalRight = totalLeft = 0;
208 count = 0;
209 loopCount++;
210
211 // This formulation attempts to ensure average loop duration is LOOP_DURATION,
212 // without causing hyperactive behavior if the code is running slower than expected.
213
214 // In particular, this won't drift at all unless a loop takes more than 10ms,
215 // and won't ever run multiple loops in a row with less than 10ms spacing.
216 lastActionTime = lastActionTime + LOOP_DURATION*int(dt / LOOP_DURATION);
217
218 // If something messed up such that this loop took ridiculously long, prevent
219 // massive values of dt next time through the loop.
220 // This happens during state transitions sometimes, especially if lots of serial data is
221 // being printed
222 if (millis() - lastActionTime > 500) {
223     lastActionTime = millis();
224 }
225 }
226
227 // Using a basic clipped PID controller, memorizes the shape of the course.
228 void memorizeLine(float leftAvg, float rightAvg)
229 {
230     // Whether the current path curves right or left (preconfigured)
231     bool useLeftSensor = currentPath->useLeftSensor;
232
233     // Step 1: compute the error from a simple PID line follower
234     float error = lineOffset(leftAvg, rightAvg, useLeftSensor);
235     PIDerror = error;
236
237     pid.Compute();
238
239     // whether the robot will turn right or left (positive is right)
240     float turnFactor = PIDoutput;
241
242     if (!useLeftSensor){
243         turnFactor *= -1;
244     }
245
246     // Step 2: constrain that error to make sure the robot doesn't turn (much) toward the line
247     if(useLeftSensor){
248         turnFactor = min(turnFactor, OUTER_TURNLIMIT);
249     } else {
250         turnFactor = max(turnFactor, -OUTER_TURNLIMIT);
251     }
252
253     leftPower = FORWARD_POWER_INITIAL + turnFactor * TURN_POWER_INITIAL;
254     rightPower = FORWARD_POWER_INITIAL - turnFactor * TURN_POWER_INITIAL;
255
256     // Step 3: Record the current motion into the path.

```

```

257 currentPath->attemptUpdate( &robotPose );
258
259
260 // Step 4: determine whether the robot has ended the current segment
261
262 float offReading = lineOffset(leftAvg, rightAvg, !useLeftSensor);
263
264 // Note guards to prevent segments from finishing immediately (less than 5cm)
265 // or outlasting their allocated memory.
266 if ((offReading > 0.5 && robotPose.distAlong > 5) || robotPose.distAlong > currentPath->
    allocatedPoints){
267     // The robot's off-line sensor has seen a line
268     Serial.println("segment end detected");
269
270
271     stop();
272     // delay(500);
273     robotPose.reset();
274
275     if(currentPathId < numPaths - 1){
276         currentPathId++;
277         currentPath = paths[currentPathId];
278     }else{
279         // trigger state transition
280         state = STATE_REPLAY;
281     }
282
283 }
284
285 // Print debug information
286 if(loopCount % 50 == 0){
287     writePoseSerial();
288 }
289 }
290
291 // Sets motor values for replaying a recorded path with input from sensor readings leftAvg
    and rightAvg
292 void replayLine(float leftAvg, float rightAvg, int dt) {
293
294     // Step 0: Handle exception for control parameters on the 5th segment (id=4)
295     double awayAdjustAmount = LINE_ANGLE_ADJUSTMENT_RATE_AWAY;
296     if(currentPathId == 4){
297         awayAdjustAmount *= 1.8;
298     }
299
300     bool useLeftSensor = currentPath->useLeftSensor;
301
302     // Step 1: adjust current odometry estimate on the basis of sensor readings.
303     // Note that this is clipped such that it will adjust dramatically to avoid
304     // a line and gradually to find it again.
305     // This prevents errors caused by the asymmetry of one-sensor following.
306
307     double lineError = lineOffset(leftAvg, rightAvg, useLeftSensor);
308
309     double lineCorrection = max(awayAdjustAmount, LINE_ANGLE_ADJUSTMENT_RATE_TOWARD)
        * dt * lineError;
310
311
312     lineCorrection = constrain(lineCorrection, -LINE_ANGLE_ADJUSTMENT_RATE_TOWARD,
        awayAdjustAmount);
313
314     robotPose.angleFrom += lineCorrection;
315
316
317     // Step 2: Drive the robot to follow the recorded path
318
319     PathPoint *target = currentPath->getPoint(robotPose.distAlong);
320
321     // error is positive if the path is left of the robot

```

```

322 double pathError = target->wrappedAngle - robotPose.angleFrom;
323
324 // whether the robot will turn right or left (positive is right)
325 double turnFactor = -pathError * PATHSTEERING_RATE;
326
327 leftPower = POWER_REPLAY * (1 + turnFactor);
328 rightPower = POWER_REPLAY * (1 - turnFactor);
329
330 // Step 3: Determine whether this segment of path is finished
331
332 float offReading = lineOffset(leftAvg, rightAvg, !useLeftSensor);
333
334 if (offReading > 0.5 && robotPose.distAlong > currentPath->usedPoints*0.7){
335     // The robot's off-line sensor has seen a line
336     Serial.println("segment end detected");
337
338     stop();
339     // delay(200);
340     robotPose.reset();
341
342     if(currentPathId < numPaths - 1){
343         currentPathId++;
344         currentPath = paths[currentPathId];
345     }else{
346         state = STATE_STOP;
347     }
348 }
349
350 // Print debug information
351 if(loopCount % 50 == 0){
352     Serial.print("pathError = ");
353     Serial.print(pathError);
354     Serial.print("\t\t");
355     Serial.println(turnFactor);
356 }
357 }
358 }
359
360 // normalizePowers ensures that
361 // abs(*left) < limit and abs(*right) < limit
362 // while maintaining their ratio.
363 // Useful for constraining desired speeds to be
364 // achievable by the motors.
365 void normalizePowers(int *left, int *right, int limit){
366     int maxabs = max(abs(*left), abs(*right));
367     if(maxabs > limit)
368     {
369         *left = (*left * limit) / maxabs;
370         *right = (*right * limit) / maxabs;
371     }
372 }
373
374 // Returns how much the selected sensor is on the line, with
375 // -1 reflecting "completely off" and 1 meaning "completely on"
376 // Note that the output is not strictly gauranteed to be in this range.
377 float lineOffset(float leftAvg, float rightAvg, bool useLeftSensor)
378 {
379     if(useLeftSensor){
380         return map(leftAvg, MIN_SENSOR_LEFT, MAX_SENSOR_LEFT, -100, 100) / 100.0;
381     } else {
382         return map(rightAvg, MIN_SENSOR_RIGHT, MAX_SENSOR_RIGHT, -100, 100) / 100.0;
383     }
384 }
385
386 // If there is any data in the serial buffer, attempts to read in that data as new P, I, and
387 // D values.
388 void handleIncomingSerial()
389 {

```

```

389  if(Serial.available() > 0){
390
391      Serial.setTimeout(100);
392      // Read first number from serial stream
393      kp = Serial.parseFloat();
394      Serial.read();
395      // Read second number from serial stream
396      ki = Serial.parseFloat();
397      Serial.read();
398      // Read third number form serial stream
399      kd = Serial.parseFloat();
400      Serial.read();
401
402      // Ingest remainder of serial buffer in case something went wrong
403      while(Serial.available()){
404          Serial.read();
405      }
406
407      pid.SetTunings(kp, ki, kd);
408      writeTuningsSerial();
409  }
410 }
411
412 void writePoseSerial(){
413     Serial.print("Pose is: \t");
414     robotPose.writeOut();
415     Serial.println();
416 }
417
418 void writeTuningsSerial()
419 {
420     Serial.println("Tunings set to (kp, ki, kd) = ");
421     Serial.print("\t");
422     Serial.print(kp);
423     Serial.print(", ");
424     Serial.print(ki);
425     Serial.print(", ");
426     Serial.print(kd);
427     Serial.print("\n");
428 }
429
430 void stop(){
431     leftPower = 0;
432     rightPower = 0;
433     driveMotors();
434 }
435
436 void driveMotors(){
437     // Inputs leftPower and rightPower vary from -255...255
438     // Code in this function is based on https://learn.adafruit.com/adafruit-motor-shield-v2-for-arduino/using-dc-motors
439
440     normalizePowers(&leftPower, &rightPower, 255);
441
442     // For each motor, decide whether to run it FORWARD, BACKWARD, (or RELEASE)
443     // These are ternary operators, returning FORWARD if power > 0 and backward otherwise.
444     byte leftDirection = (leftPower > 0) ? FORWARD : BACKWARD;
445     byte rightDirection = (rightPower > 0) ? FORWARD : BACKWARD;
446
447     // Set motor speeds
448     leftMotor -> setSpeed(abs(leftPower));
449     rightMotor -> setSpeed(abs(rightPower));
450
451     // Set motor directions
452     leftMotor -> run(leftDirection);
453     rightMotor -> run(rightDirection);
454
455 }

```

```

456
457 void getMeasurements(int *leftRead , int *rightRead)
458 {
459     *leftRead = analogRead(leftSensorPin);
460     *rightRead = analogRead(rightSensorPin);
461 }

```

#### Path library (.h)

```

1 // PathPoint represents the data associated with each point
2 // along a recorded segment of path.
3 // Currently this is just an angle because the speed adjustment flags
4 // needed to be removed before they could be fully implemented.
5 class PathPoint
6 {
7 public:
8     double wrappedAngle;
9
10    PathPoint();
11 };
12
13 PathPoint::PathPoint( void ){
14     wrappedAngle = 0;
15 }
16
17 // A Path is a collection of PathPoints that represents a segment of the course.
18 class Path
19 {
20 public:
21     Path(int length , bool useLeft);
22
23     PathPoint *points;
24     int allocatedPoints; // = 0;
25     int usedPoints; // = 0;
26
27     bool useLeftSensor; // = false;
28
29     void writeOut( void );
30     bool attemptUpdate( Pose *pose );
31     void smooth( byte smoothingLength );
32     PathPoint *getPoint( double distAlong );
33 };
34
35 // Note that the points array is dynamically allocated to save space.
36 Path::Path(int length , bool useLeft){
37     useLeftSensor = useLeft;
38
39     allocatedPoints = length;
40
41     points = (PathPoint*) malloc(sizeof(PathPoint) * length);
42
43     for (int i = 0; i < allocatedPoints; ++i)
44     {
45         points[i] = PathPoint();
46     }
47
48     points[0].wrappedAngle = 0;
49 }
50
51 PathPoint *Path::getPoint( double distAlong ){
52     int index = min(distAlong , usedPoints - 1);
53     if (index < 0)
54     {
55         return NULL;
56     }
57     return &points[index];
58 }
59

```

```

60 // Updates the path with new information if the information is not currently contained in
    the path.
61 bool Path::attemptUpdate( Pose *pose ) {
62     if (int(pose->distAlong) > usedPoints && usedPoints < allocatedPoints){
63         usedPoints++;
64         points[usedPoints].wrappedAngle = pose->angleFrom;
65         return true;
66     } else {
67         return false;
68     }
69 }
70
71 // Prints Path data to serial for debugging and analysis.
72 void Path::writeOut(){
73     Serial.println("id\tspeed\tisOffline\tangle");
74     for (int i = 0; i < usedPoints; ++i)
75     {
76         Serial.print(i);
77         Serial.print("\t");
78         Serial.print(points[i].wrappedAngle);
79         Serial.println();
80     }
81 }
82
83 // Runs a n-point leading average to enable both forward-looking controls predictions
84 // and to smooth out irregular behavior in the training follow.
85 void Path::smooth(byte smoothingLength){
86     for (int i = 0; i < usedPoints - smoothingLength; ++i)
87     {
88         double total = 0;
89         for (int j = 0; j < smoothingLength; ++j)
90         {
91             double point = points[i+j].wrappedAngle;
92
93             total += point;
94         }
95         double average = total / smoothingLength;
96
97         points[i].wrappedAngle = average;
98     }
99 }

```

#### Odometry code (.h)

```

1 #include <Arduino.h>
2
3 // This file attempts to estimate the ego-motion of the robot
4 // using the commanded powers of the motors. Expect accuracy of +/- 15%
5
6 class Pose
7 {
8 public:
9     Pose();
10    void odometryUpdate(int leftPower, int rightPower, int timestep);
11    void writeOut( void );
12    void reset();
13    // Measured in cm, with positive being forward of the starting position
14    double distAlong;
15    // Measured in degrees, with positive being nose left.
16    double angleFrom;
17 };
18
19 Pose::Pose() {
20     reset();
21 }
22
23 void Pose::writeOut( void ) {
24     Serial.print("d, theta) = (");

```



```

25 Serial.print(distAlong);
26 Serial.print(", ");
27 Serial.print(angleFrom);
28 Serial.print("\n");
29 }
30
31 void Pose::reset(){
32     distAlong = 0;
33     angleFrom = 0;
34 }
35
36 // Neither of these constants matter much as long as they do not
37 // change between recording and playback.
38 // MAX_FORWARD_SPEED is scaled to approximately 1 = 1cm on our robot,
39 // while MAX_TURN_SPEED is approximately 1 = 1/10 degree.
40 const double MAX_FORWARD_SPEED = 100;
41 const double MAX_TURN_SPEED = 1000;
42
43 double adjustPower( int power ){
44     double powerfrac = power / 255.0;
45     return powerfrac - .05 * powerfrac * powerfrac;
46 }
47
48 // Update the estimated pose of the robot based on wheel (estimated) odometry.
49 // Currently uses an almost linear approximation, with slight falloff at high powers.
50 // timestep is in ms
51 void Pose::odometryUpdate( int leftPower, int rightPower, int timestep) {
52     if (timestep > 100)
53     {
54         // Prevent large timesteps from causing jumps in odometry readings (for example, on
55         // first boot)
56         return;
57     }
58     double leftPowerFrac = adjustPower(leftPower);
59     double rightPowerFrac = adjustPower(rightPower);
60     double forwardSpeed = (leftPowerFrac + rightPowerFrac) / 2 * MAX_FORWARD_SPEED;
61     double turnSpeed = (rightPowerFrac - leftPowerFrac) / 2 * MAX_TURN_SPEED;
62
63     distAlong += forwardSpeed * timestep / 1000.0;
64     angleFrom += turnSpeed * timestep / 1000.0;
65 }

```

## 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)

```

1 #ifndef ADAFRUIT_MOTORSHIELD_H
2 #define ADAFRUIT_MOTORSHIELD_H
3
4 // This class emulates the behavior of the Adafruit Motor Shield V2 for the Arduino.
5
6 #include "utils.h"
7 #include "arduino.h"
8 #include "robot.h"
9
10 #define FORWARD 1
11
12 #define BACKWARD 2

```

```

13
14 class Adafruit_DCMotor{
15 private:
16     int pin;
17     int speed;
18 public:
19     Adafruit_DCMotor(int pin);
20     void setSpeed(int);
21     void run(byte&);
22 };
23
24 class Adafruit_MotorShield
25 {
26 private:
27     Adafruit_DCMotor *motor_left, *motor_right;
28 public:
29     Adafruit_MotorShield();
30     ~Adafruit_MotorShield();
31     void begin();
32     Adafruit_DCMotor* getMotor(int);
33 };
34
35 #endif // ADAFRUIT_MOTORSHIELD.H

```

#### Adafruit\_MotorShield.cpp (definition)

```

1 #include "Adafruit_MotorShield.h"
2
3 // function definitions for the emulation of the Adafruit Motor Shield V2
4 // just to have it interface with the "virtual arduino"
5
6 Adafruit_DCMotor::Adafruit_DCMotor(int pin):pin(pin){
7
8 }
9
10 void Adafruit_DCMotor::run(byte & direction){
11     float dir;
12     if(direction == FORWARD){
13         dir = 1;
14     }else if (direction == BACKWARD){
15         dir = -1;
16     }
17
18     if(robot){
19         switch(pin){
20             case 1: //left
21                 robot->setPowerL(dir * speed);
22                 break;
23             case 2:
24                 robot->setPowerR(dir * speed);
25                 break;
26         }
27     }
28 }
29
30
31 void Adafruit_DCMotor::setSpeed(int speed){
32     this->speed = speed;
33 }
34
35 Adafruit_MotorShield::Adafruit_MotorShield()
36 {
37     this->motor_left = new Adafruit_DCMotor(1);
38     this->motor_right = new Adafruit_DCMotor(2);
39 }
40
41 Adafruit_MotorShield::~Adafruit_MotorShield(){
42     if(motor_left){

```

```

43     delete motor_left;
44     motor_left = nullptr;
45 }
46 if(motor_right){
47     delete motor_right;
48     motor_right = nullptr;
49 }
50 }
51
52 void Adafruit_MotorShield::begin(){
53     // don't need to do anything here
54 }
55
56 Adafruit_DCMotor* Adafruit_MotorShield::getMotor(int id){
57     switch(id){
58     case 1:
59         return this->motor_left;
60         break;
61     case 2:
62         return this->motor_right;
63         break;
64     default:
65         return nullptr;
66     }
67 }

```

arduino.h (declaration)

```

1 #ifndef ARDUINO_H
2 #define ARDUINO_H
3
4 // arduino.h essentially enables native arduino code to run seamlessly with the application.
5
6 #define BIN (2)
7
8 const int A0 = 0xA0;
9 const int A1 = 0xA1;
10
11 #include <iostream>
12 #include <bitset>
13
14 #include "utils.h"
15 #include "mainwindow.h"
16 #include "robot.h"
17
18 class _Serial{
19
20 public:
21     void begin(int){
22         // ignore baud rate
23     }
24
25     // all prints are redirected to stdout
26     template<typename T>
27     void print(T val){
28         std::cout << val;
29     }
30
31     template<typename T>
32     void print(T val, int flag){
33         if(flag == BIN){
34             std::cout << std::bitset<8>(val);
35         }
36         // ... not handling other flags yet
37     }
38
39     template<typename T>
40     void println(T val){

```

```

41     std::cout << val << std::endl;
42 }
43
44 void println(){
45     std::cout << std::endl;
46 }
47 };
48
49 extern _Serial Serial;
50 typedef unsigned char byte;
51
52 // Arduino APIs
53
54 extern long long millis();
55 extern int analogRead(const int pin);
56 extern void setup();
57 extern void loop();
58 extern void delay(int);
59
60 #endif // ARDUINO_H

```

arduino.cpp (definition)

```

1 #ifndef ARDUINO_H
2 #define ARDUINO_H
3
4 // arduino.h essentially enables native arduino code to run seamlessly with the application.
5
6 #define BIN (2)
7
8 const int A0 = 0xA0;
9 const int A1 = 0xA1;
10
11 #include <iostream>
12 #include <bitset>
13
14 #include "utils.h"
15 #include "mainwindow.h"
16 #include "robot.h"
17
18 class _Serial{
19
20 public:
21     void begin(int){
22         // ignore baud rate
23     }
24
25     // all prints are redirected to stdout
26     template<typename T>
27     void print(T val){
28         std::cout << val;
29     }
30
31     template<typename T>
32     void print(T val, int flag){
33         if(flag == BIN){
34             std::cout << std::bitset<8>(val);
35         }
36         // ... not handling other flags yet
37     }
38
39     template<typename T>
40     void println(T val){
41         std::cout << val << std::endl;
42     }
43
44     void println(){
45         std::cout << std::endl;

```

```

46     }
47 };
48
49 extern _Serial Serial;
50 typedef unsigned char byte;
51
52 // Arduino APIs
53
54 extern long long millis();
55 extern int analogRead(const int pin);
56 extern void setup();
57 extern void loop();
58 extern void delay(int);
59
60 #endif // ARDUINO.H

```

## 7.2.2 System Definition

Route.h (declaration)

```

1 #ifndef ROUTE.H
2 #define ROUTE.H
3
4 #include <QVector>
5 #include <QPointF>
6 #include <QPolygonF>
7 #include <QGraphicsPolygonItem>
8 #include <QPen>
9 #include <QString>
10 #include <QFile>
11 #include <QTextStream>
12
13 #include "utils.h"
14
15 struct Route
16 {
17     //QVector<QPointF> route;
18     QPolygonF poly;
19     QGraphicsPolygonItem poly_item;
20     QPen routePen;
21 public:
22     Route();
23
24     void draw();
25     void reset(int n); //randomize to length
26     void reset(const QVector<QPointF>& route);
27
28     void save(const QString& filename);
29     void load(const QString& filename);
30 };
31
32 extern Route* route;
33
34 #endif // ROUTE.H

```

Route.cpp (definition)

```

1 #include "route.h"
2 #include <random>
3 #include <iostream>
4
5 const double R_MIN = PXL_DIMS/8;
6 const double R_MAX = PXL_DIMS/2;
7
8 Route* route;
9
10 Route::Route()

```

```

11 {
12     QBrush routeBrush = QBrush(QColor::fromRgb(0,0,0),Qt::SolidPattern);
13     routePen = QPen(routeBrush, c2p(1.7));
14 }
15
16 void Route::reset(int n){
17     // reinitialize to a route of n points
18
19     QVector<QPointF> route;
20
21     float x = R_MAX;
22     float y = R_MAX; // at center
23
24     for(int i=0; i<n; ++i){
25         float t = (2 * M_PI) * i / n;
26         float r = R_MIN + (R_MAX - R_MIN) * float(rand())/RAND_MAX;
27         route.push_back(QPointF(x + r*cos(t), y + r*sin(t)));
28     }
29     std::cout << "N : " << n << " ROUTE SIZE : " << route.size() << std::endl;
30     reset(route);
31 }
32
33 void Route::reset(const QVector<QPointF> &route){
34     // reset from vector
35     poly = QPolygonF(route);
36     poly_item.setPolygon(poly);
37     poly_item.setPen(routePen);
38 }
39
40 void Route::save(const QString& filename){
41     // save route to file
42     QFile file(filename);
43     if (file.open(QIODevice::ReadWrite)) {
44         QTextStream stream(&file);
45
46         for(auto& p : poly){
47             stream << p.x() << ',' << p.y() << '\n';
48         }
49     }
50     file.close();
51 }
52
53 void Route::load(const QString& filename){
54     // load route from file
55     QFile file(filename);
56     if (file.open(QIODevice::ReadOnly)) {
57         QTextStream stream(&file);
58         QVector<QPointF> route;
59
60         while(!stream.atEnd()){
61             QString line = stream.readLine();
62             QStringList pt_s = line.split(",");
63             QPointF pt(pt_s[0].toFloat(), pt_s[1].toFloat());
64             route.push_back(pt);
65         }
66
67         file.close();
68         Route::reset(route);
69     }
70 }

```

Robot.h (declaration)

```

1 #ifndef ROBOT_H
2 #define ROBOT_H
3
4 #include <QObject>
5 #include <QPointF>

```

```

6
7 #include <QPolygonF>
8
9 #include <QGraphicsRectItem>
10 #include <QGraphicsEllipseItem>
11
12 #include <QGraphicsScene>
13
14 #include "utils.h"
15 #include "robotitem.h"
16
17
18 class Robot
19 {
20
21 public:
22     QPointF pos; // current position
23     float theta; // heading, measured from horz. radians
24
25     QPointF irOffset; // position of ir from center
26
27     float vel_l, vel_r; //left-right velocity of motors
28
29     float h; // height of IR sensor, inches
30     float fov; //field of view of IR sensor, radians
31
32     float ir_val_l; //value of ir reflectance sensors
33     float ir_val_r;
34
35     // frequently computed values
36     float cr; //cone radius
37
38     RobotItem* body;
39
40 public:
41     Robot(QGraphicsScene& scene,
42          QPointF pos, float theta,
43          QPointF irOffset, float h, float fov);
44     ~Robot();
45
46     void reset(QPointF pos, float theta);
47     void reset(QPolygonF route);
48
49     void update();
50     void move(float delta, float dtheta);
51     void setVelocity(float left, float right);
52
53     void setPowerR(int r);
54     void setPowerL(int l);
55
56     void setVelocityR(float r);
57     void setVelocityL(float l);
58
59     void setIRHeight(float h);
60
61     void sense(QImage& img);
62     void setVisible(bool visible);
63 };
64
65 extern Robot* robot;
66
67 #endif // ROBOT.H

```

#### Robot.cpp (definition)

```

1 #include "utils.h"
2 #include "robot.h"
3

```

```

4 #include <iostream>
5
6 // 1 pxl = .5 cm
7 // 270 x 270 cm world
8
9 // robot = 20x16 cm
10
11 Robot* robot = nullptr;
12
13 float coneRadius(float h, float fov){
14     return h*tan(fov/2);
15 }
16
17 // RobotBody
18 Robot::Robot(QGraphicsScene& scene, QPointF pos, float theta, QPointF irOffset, float h,
19             float fov):
20     pos(pos),
21     theta(theta),
22     irOffset(irOffset),
23     h(h),
24     fov(fov),
25     cr(coneRadius(h, fov))
26 {
27     vel_l = vel_r = 0.;
28
29     body = new RobotItem(pos, QPointF(ROBOT.LENGTH,ROBOT.WIDTH),irOffset, theta, cr);
30     // RobotItem will take care of graphics
31     scene.addItem(body);
32 }
33 Robot::~Robot(){
34     if(body){
35         delete body;
36         body = nullptr;
37     }
38 }
39
40 void Robot::reset(QPointF p, float t){
41     pos=p; theta=t;
42     body->setPos(pos, theta);
43 }
44
45 void Robot::reset(QPolygonF route){
46     pos = route.front();
47     float dst_x = route[1].x();
48     float dst_y = route[1].y();
49
50     theta = atan2(pos.y() - dst_y, dst_x - pos.x());
51 }
52
53
54 void Robot::move(float delta, float dtheta){
55     theta += dtheta * DT;
56     pos += delta * QPointF(cos(theta), -sin(theta)) * DT;
57     update();
58 }
59
60 void Robot::update(){
61     float w = (vel_r - vel_l) / WHEELDISTANCE;
62     if(w != 0){
63         float R = (vel_l + vel_r) / (2*w);
64
65         QPointF ICC = pos + R * QPointF(-sin(theta), -cos(theta));
66         // ICC = virtual center of rotation
67         // update position based on obtained kinematics prediction
68
69         float x = pos.x();
70         float y = pos.y();

```



```

71     float iccx = ICC.x();
72     float iccy = ICC.y();
73
74     pos.setX(
75         cos(-w*DT) * (x - iccx) +
76         -sin(-w*DT) * (y - iccy) +
77         iccx
78     );
79     pos.setY(
80         sin(-w*DT) * (x - iccx) +
81         cos(-w*DT) * (y - iccy) +
82         iccy
83     );
84     theta += w * DT;
85 }else{
86     // if w == 0, then division by zero would be bad..
87     // since it's a special (and well-defined) case, we should handle this
88     pos += vel_l * QPointF(cos(theta), -sin(theta)) * DT;
89 }
90
91
92     body->setPos(pos, theta);
93 }
94 void Robot::setVelocityL(float v){
95     vel_l = v;
96 }
97
98 void Robot::setVelocityR(float v){
99     vel_r = v;
100 }
101
102 void Robot::setPowerL(int pow){
103     // convert power to velocity
104     // then convert it back to pixel units
105     setVelocityL(c2p(pow2vel(pow)));
106 }
107
108
109 void Robot::setPowerR(int pow){
110     // convert power to velocity
111     // then convert it back to pixel units
112     setVelocityR(c2p(pow2vel(pow)));
113 }
114
115
116 void Robot::setVelocity(float l, float r){
117     setVelocityL(l);
118     setVelocityR(r);
119 }
120
121 void Robot::setIRHeight(float h){
122     this->h = h;
123     this->cr = coneRadius(h, fov);
124     body->setCR(this->cr);
125 }
126
127 void Robot::sense(QImage& image){
128     // poll a conical section of the ground-plane beneath the IR sensors
129     // and average out the readings of pixels
130
131     float l_1 = irOffset.x();
132     float l_2 = irOffset.y();
133
134     QPointF ir_root = pos + QPointF(l_1 * cos(theta), -l_1 * sin(theta));
135     QPointF ir_l = ir_root - QPointF(l_2 * sin(theta), l_2 * cos(theta));
136     QPointF ir_r = ir_root + QPointF(l_2 * sin(theta), l_2 * cos(theta));
137
138     float cR = coneRadius(h, fov);

```

```

139     int i_cR = round(cR);
140
141     int n = 0;
142     float sum_l=0;
143     float sum_r = 0;
144
145     for(int offsetX = -i_cR; offsetX <= i_cR; ++offsetX){
146         for(int offsetY = -i_cR; offsetY <= i_cR; ++offsetY){
147             if((offsetX * offsetX + offsetY * offsetY) < (cR * cR)){
148                 QPointF offset(offsetX, offsetY);
149
150                 QRgb col_l = image.pixel((ir_l + offset).toPoint());
151                 QRgb col_r = image.pixel((ir_r + offset).toPoint());
152
153                 sum_l += qGray(col_l) / 255.0;
154                 sum_r += qGray(col_r) / 255.0;
155                 ++n;
156
157             }
158         }
159     }
160 }
161
162 // set ir values
163 if(n){
164     ir_val_l = sum_l / n;
165     ir_val_r = sum_r / n;
166 }
167 }
168
169 void Robot::setVisible(bool visible){
170     body->setVisible(visible);
171 }

```