ELE 302 – INDEPENDENT PROJECT WRITE-UP

TJ Smith Byung-Cheol Cho
(Bench 207)

Due May 19, 2017

1 Overview

The aim of this project was to build a robot capable of playing basic ping pong. Prior to beginning this project, we established four ideal objectives for the end product:

1. Track a ping pong ball, estimate its trajectory and predict the location where the ball will land
2. Determine (1) when it is the robot's turn to hit the ball and (2) when the ball will hit the net or leave the playing area
3. Move the robot to the required location before the ball bounces twice and hit the ball back over the net
4. Avoid leaving the playing field (area enclosed by table and net)

By Demo Day, we had implemented the hardware and software to achieve objectives 1, 3 and the first half of objective 2, and we had the capability of achieving the remainder of the objectives had we decided to implement them in software.

We used two Pixy cameras (CMUcam5) placed a fixed distance apart on the robot to triangulate the location of a bright red ball in three-dimensional coordinates. Once the camera system had collected enough data points to accurately predict all future positions of the ball until the second bounce, the robot moved to a location where the ball would bounce to the center of a paddle attached to its side, and provided a brief flick to the paddle to return the ball.

The speed and flexibility of movement we needed (we had to move up to 3 feet in less than half a second) required us to abandon the chassis used for our previous speed control and navigation assignments and adopt the large and powerful omni-wheel drive used by Ethan Gordon and Luke Pfleger in 2016. We used an accelerometer and gyroscope for dead reckoning of position (we could no longer use wheel rotation tracking because of slippage). In addition, we continued to use the PSoC 5LP because of its extensive motor control capacity (we required four PWM modules) while adding a Raspbery Pi 3 because of the intensive computation we expected to require for ball tracking and trajectory estimation.

2 Theory

2.1 Stereo vision

We used a pinhole camera model for our Pixy cameras. Under this model, points in 3-D world coordinates are projected onto an imaging plane using homogenous coordinates. In the simplified case when the camera is at the origin pointing in the positive z direction (i.e. the image plane is at $z = 1$), we convert from world coordinates $(x, y, z)^T$ to image coorinates $(m, n)^T$ using basic linear algebra. First, we define $\left(x', y', z'\right)^T$ as follows:

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_t \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

The 3×3 matrix is known as the **camera intrinsic matrix**, F, with entries f_x and f_y that represent the scale of conversion between world units and pixel units, and c_x and c_y that are the horizontal and vertical offsets of the image origin in pixels. Since these are all parameters internal to the camera, they are known as intrinsic parameters. Typically, c_x and c_y correspond to half of the width and height of the output image respectively. Then, to convert to image coordinates we simply divide by the z'-component:

$$\begin{pmatrix} m \\ n \end{pmatrix} = \begin{pmatrix} x'/z' \\ y'/z' \end{pmatrix}$$

Concisely, we can write $\mathbf{w}_h \equiv F\mathbf{w}$ where $\mathbf{w}_h = (m, n)^T$ and $\mathbf{w} = (x, y, z)^T$, and $\equiv$ represents the process of dividing by the third component. If the camera is located at a point $\mathbf{p}$ from the origin, we can simply modify this equation to read $\mathbf{w}_h \equiv F(\mathbf{w} - \mathbf{p})$.

The problem with position estimation with one camera is that this perspective projection is not reversible (the input has three degrees of freedom, while the output only has two), so we need some additional source of positional information in order to recover the third degree of freedom. We briefly considered using a single camera (for simplicity) and the observed size of the ball, since linear dimensions should scale with $1/z$ with distance from the camera, while the area should decrease with $1/z^2$. However, we quickly switched to using two cameras because (1) at reasonable distances (even as close as three feet away from the camera), the observed size of the ball was too small (on the order of tens of pixels) that it would have produced unacceptable granularity in our depth measurements; and (2) the image processing of the Pixy camera was noisy and caused the area to fluctuate too much, producing noise in depth that would have been unacceptable for trajectory estimation.

Given image coordinates from two identical cameras with a known translational separation (and to simplify our calculations, no rotation), we can recover the original z-component in world coordinates by solving

$$\begin{cases} \mathbf{w}_{h1} & \equiv F(\mathbf{w} - \mathbf{p}_1) \\ \mathbf{w}_{h2} & \equiv F(\mathbf{w} - \mathbf{p}_2) \end{cases}$$
$$\implies \begin{pmatrix} m_1 - m_2 \\ n_1 - n_2 \\ z \end{pmatrix} \equiv F(\mathbf{p}_2 - \mathbf{p}_1)$$

If there is only an x-displacement between the two cameras, we can estimate z directly from only the horizontal disparity in the images:

$$\implies z = \frac{f_x t_x}{m_1 - m_2}$$

where $t_x = (\mathbf{p}_2 - \mathbf{p}_1)_x$.

The convention for camera coordinates typically has z along the principal axis of the camera, but since our camera was mounted on the front of the robot, we performed a basic coordinate transform so that z was vertical (normal to the ground):

$$x_{\text{world}} = x_{\text{camera}} \qquad y_{\text{world}} = z_{\text{camera}} \qquad z_{\text{world}} = y_{\text{camera}}$$

For trajectory estimation and prediction, we used only world coordinates to avoid confusion.

It is important to note that the Pixy camera is not an ideal pinhole camera. Critically, it exhibits radial distortion. We attempted correcting for this by calibrating using MATLAB's camera calibration methods (e.g. the `cameraCalibrator` app), but we found little significant improvement in position estimation, and the small position differences from the distorted image and coordinates were sufficient for our purposes.

2.2 Trajectory estimation and prediction

Trajectory estimation is the process of performing statistical estimation of kinematic parameters, such as initial position and velocity in all three spatial dimensions, and acceleration in the world z-axis. Among our many design iterations, we also estimated the coefficient of restitution from our observed data. We approached trajectory estimation with a linear regression model with various modifications, and then used the estimated kinematic parameters to estimate the time and location of bounces (i.e. perform trajectory prediction). We decided to use regression instead of finite difference methods because we expected our stereo ball tracking data to be rather noisy (finite differences tend to amplify high-frequency noise).

We divided our model of the ball's trajectory into the global x, y and z coordinates (relative to the robot, since we simplified our problem by tracking the ball only while the robot was stationary). For the x and y coordinates, we performed simple linear regression over all detected ball positions:

$$x \sim \beta_{x0} + \beta_{x1} t \qquad\qquad y \sim \beta_{y0} + \beta_{y1} t$$

For the z coordinate, we performed linear regression with quadratic time features:

$$z \sim \beta_{z0} + \beta_{z1} t + \beta_{z2} t^2$$

β_{x0}, β_{y0} and β_{z0} all represent the initial positions (at $t = 0$); β_{x1}, β_{y1} and β_{z1} all represent the initial velicities (at $t = 0$); and β_{z2} represents the acceleration due to gravity.

All of these regression models can be written as $\mathbf{Y} \sim \mathbb{X}\boldsymbol{\beta}$ where $\mathbb{X}$ is the $n \times d$ feature matrix: for the x and y coordinates, $\mathbb{X}$ consists of a column of ones, and a column for t; for the z coordinate, $\mathbb{X}$ consists of a column of ones, a column for t, and a column for t^2. Then, assuming that $n \geq d$ so that $\mathbb{X}$ is not rank-deficient, by ordinary least squares (OLS) we have a simple formula for the regression coefficients, $\boldsymbol{\beta}$:

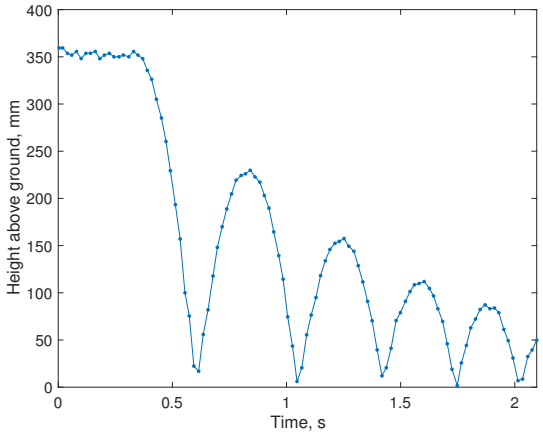
$$\hat{\boldsymbol{\beta}}^{\text{OLS}} = \left(\mathbb{X}^T\mathbb{X}\right)^{-1}\mathbb{X}^T\mathbf{Y}.$$

The z-coordinate regression model is, however, rather limited: since it represents a pure parabola, it cannot take bouncing into account. To overcome this, we implemented a transform mapping points from after a bounce back onto the original parabola to continue to improve our regression coefficients; this transform was based on the observation that the parabola after a bounce is a scaled version of the original parabola (see **Figure 1(a)** below).

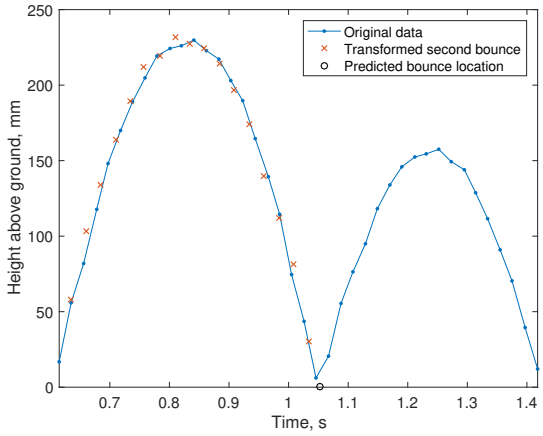
Specifically, the bounce height and duration are completely determined by the velocity of the ball after the bounce, so if R is the coefficient of restitution (i.e. the ratio between energy after and before a bounce, or equivalently, between the maximum height before and after a bounce), the velocity after the bounce is scaled by $\sqrt{R}$, the height is scaled by R, and the duration of the bounce is scaled by $\sqrt{R}$ (since bounce duration is given by $2v_{\text{init}} - gt_{\text{duration}}$). Therefore, in order to rescale the data back to the first parabola, we reflect time points $t \mapsto t_{\text{bounce}} - (t - t_{\text{bounce}})/\sqrt{R}$ and scale $z \mapsto z/R$. This is demonstrated in **Figure 1(b)** below.

However, this introduced a new free parameter, R, that happens to be critical for accurate transformation. We could not estimate R using linear regression, because the model was no longer linear: $z \sim \left(\beta_0 + \beta_1 t + \beta_2 t^2\right)/R$. Thus, we were forced to either hard-code a value for the coefficient of restitution, attempt to minimize the mean-squared error among a range of R values (since reasonable values of R were between 0.2 and 0.8), or iteratively regress for better values of R and $\boldsymbol{\beta}$ using some form of block coordinate minimization. We initially attempted to implement the latter two algorithms, but in the late hours of Wednesday night / Thursday morning, something was wrong in our implementations, so we decided to hard-code a value of R based on the surface and type of ball we were working with.

Finally, once the kinematic parameters were estimated, it was trivial to predict the location of the ball in global coordinates for any point in time in the future. Specifically, the landing location could be estimated by computing the roots of the z polynomial, and by simple extension, the location of the ball at any height (since our paddle was vertically fixed) could be estimated using some more quadratic formulas.



(a) Plot of ball height over several bounces. The ball is dropped around $t = 0.35\,\text{s}$.



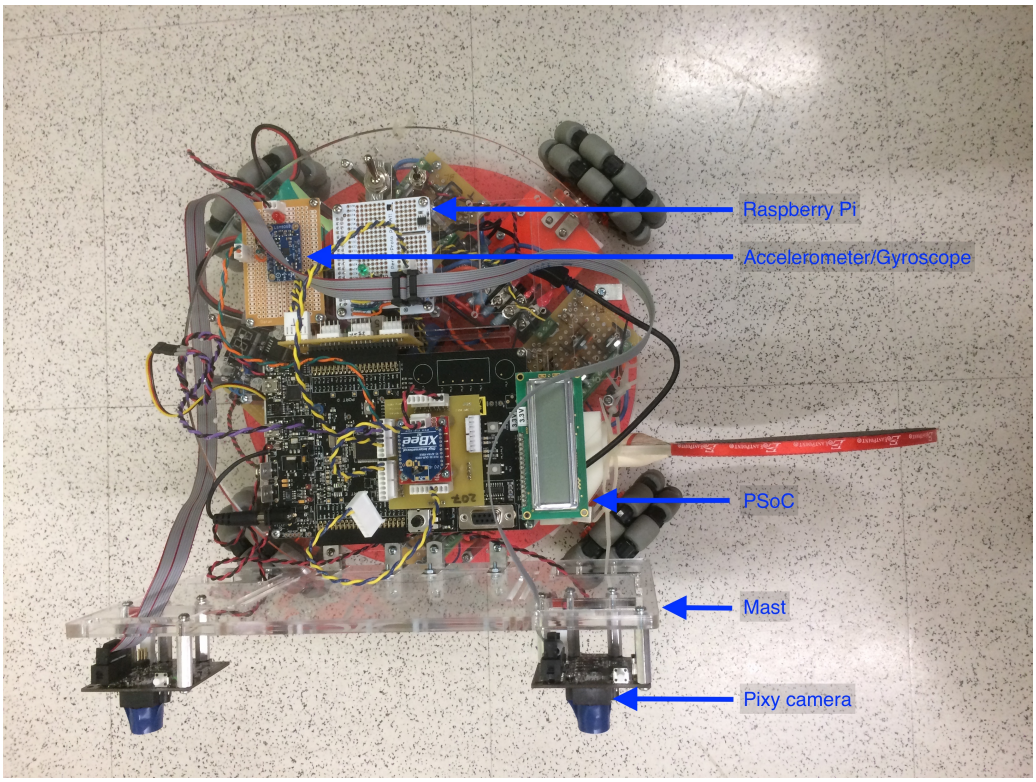
(b) Demonstration of the reflection and rescaling transform, using $R = 0.68$.

Figure 1: **Kinematics of ball-bouncing.**

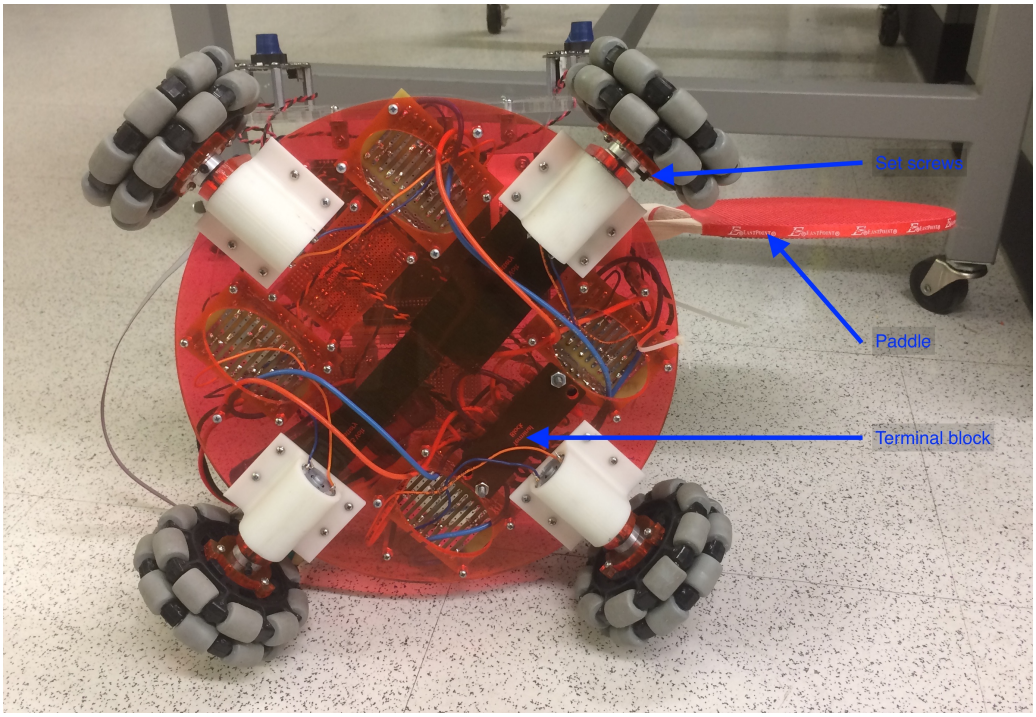
2.3 Omni-drive

3 Hardware

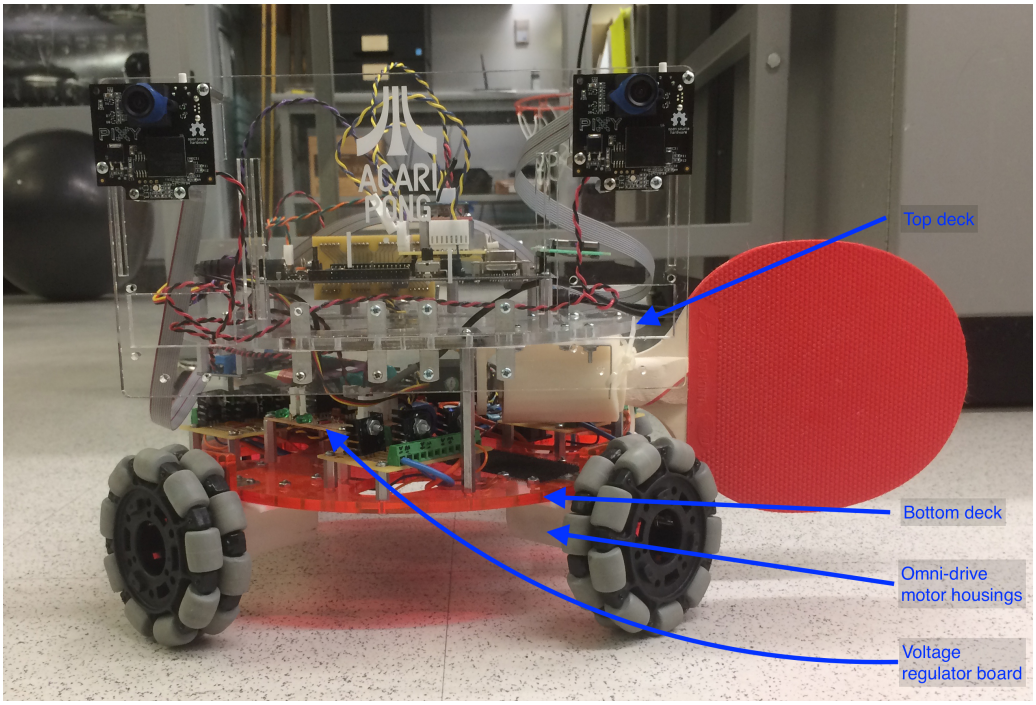
Figure 2 shows photographs of the final product from various angles.



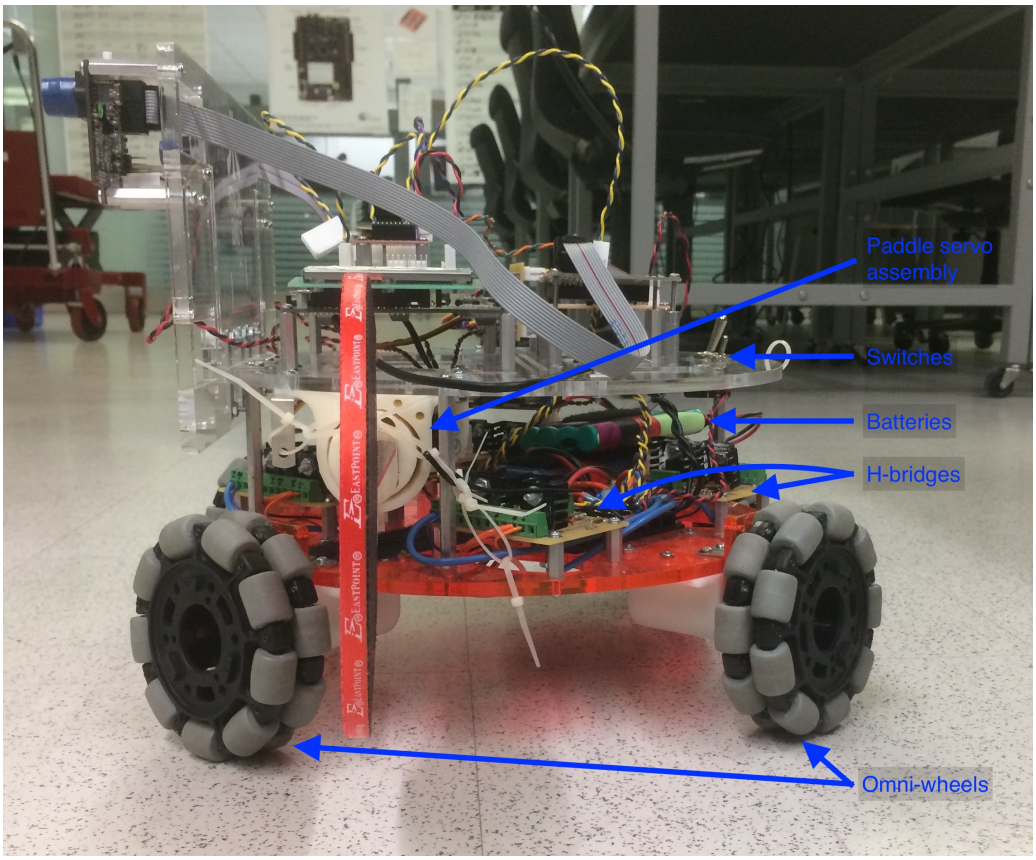
(a) Top view.



(b) Bottom view.



(c) Front view.



(d) Side view.

Figure 2: **Photographs of the final robot.**

Figure 3 shows a block diagram of the hardware set-up.

Figure 3: **Block diagram of the hardware setup for ping pong.** Black connections indicate ground connections; red connections indicate power (both regulated and unregulated); all other connections indicate data connections (e.g. from the PSoC to the H-bridges and servo, various I^2C connections). Green components indicate power-related components; red components indicate actuators; blue components indicate sensors.

3.1 Chassis

3.2 Omni-wheel drive

3.2.1 Motors and chassis

3.2.2 H-bridges and motor control

3.3 Cameras and mast

Figure 4: **Camera mounted on mast.** Labels indicate the mast, the camera and the cable connecting the camera to the camera board.

3.4 Paddle mount

3.5 Power supply

3.6 Programmable hardware

Figure 6 shows the top-design of our programmable hardware in PSoC Creator 2.1.

(a) Photograph

(b) Schematic

(c) Circuit diagram

Figure 5: **Circuitry of the camera board.**

3.6.1 Hardware for motor control

3.6.2 Hardware for dead reckoning

4 Software for PSoC

4.1 Omni-drive control

4.2 Dead reckoning

4.3 Paddle mount servo control

5 Software for Raspberry Pi

5.1 Pixy camera communication

5.2 Stereo vision

5.3 Trajectory estimation

5.4 Serial communication

We continued to use serial communication extensively to debug the camera and navigation portions of our program and to adjust our PID variables. We also moved to using the DB9 header only for bench testing and a wireless XBee serial communication chip for track testing. We changed nearly no code or programmable hardware to move to the XBee, since it relied on the same UART communication as the DB9.

We heavily updated our list of serial commands to improve memorability and consistency, and to expand to control camera and navigation parameters. Our updated commands are listed in **Table 1**.

Command	Description
CTx	Change throttle
GS	Get all variables associated with speed control
TS	Toggle speed PID control
TDC	Toggle distance timeout for speed control
TDS	Toggle dynamic speed control
CPSx	Change proportional term for speed control
CISx	Change integral term for speed control
CDSx	Change derivative term for speed control
CSSx	Change steady-state throttle for speed control
CTSx	Change target speed for speed control
CTDx	Change target distance for speed control
TVS	Toggle verbose printout for speed PID control
GC	Get all camera variables
RC	Reset camera variables
CSx	Change steering/servo direction
CMMx	Change maximum permitted line misses
TN	Toggle navigation PID control
CPLx	Change proportional term for line position error
CILx	Change integral term for line position error
CIILx	Change double integral term for line position error
CDLx	Change derivative term for line position error
CPTx	Change proportional term for angle error
CITx	Change integral term for angle error
CDTx	Change derivative term for angle error
CTSNx	Change nominal target speed for navigation
TLE	Toggle line error tracking
TVN	Toggle verbose printout for navigation PID control
A	Abort (kill speed and navigation PID control)

Table 1: **Table of serial interface commands.**

6 Results

See **Table 2** for the final PID coefficients we used. We updated our coefficients from speed control because we are no longer dynamically updating the target speed to reach the distance in a specific time. We used only the proportional and derivative terms for line position and angle (no integral terms), and unfortunately, despite our efforts to calculate it accurately, the angle contributed very little to our control. Given more time, we may have been able to develop a better system of control to more robustly control steering at higher speeds.

See **Table 3** for our final times for one lap. The highest speed we managed to achieve was with a nominal 6.0 ft/s, though by reducing speed while turning corners, our average speed was only about 5.4 ft/s. We were unfortunately unable to consistently replicate this speed, and we performed our demo with a nominal target speed of 4.0 ft/s.

	P	I	D
Speed control	80.0	1.0	10.0
Line position	3.0	0.0	2.0
Line angle	0.5	0.0	0.0

Table 2: **Final PID coefficients**

Nominal target speed	Time for one lap	Average lap speed
4.0 ft/s	28 s	3.8 ft/s
5.0 ft/s	22 s	4.9 ft/s
6.0 ft/s	19.96 s	5.4 ft/s

Table 3: **Time trial results**

7 Further work

Figure 6: **Top-design of the navigation project.**

8 Appendix: full listings

Contents

8.1 `main.c`