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 Model B 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, $\boldsymbol{\beta}$:

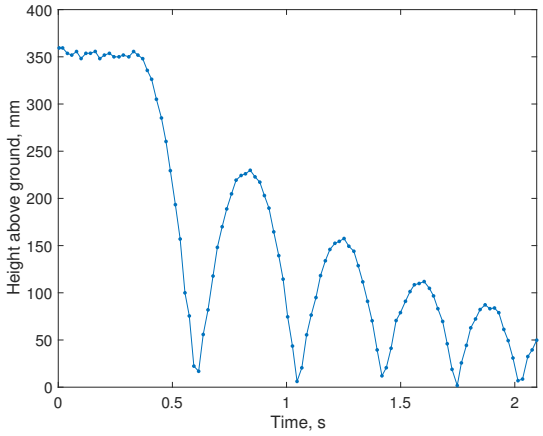
$$\hat{\boldsymbol{\beta}}^{\text{OLS}} = \left(\mathbb{X}^T\mathbb{X}\right)^{-1}\mathbb{X}^T\mathbf{Y}.$$

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).

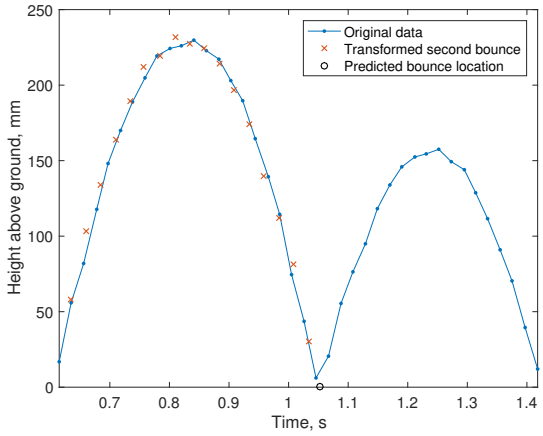
Specifically, the bounce height and duration are completely determined by the velocity of the ball after the bounce, so if R is the coefficient of restitution (i.e. the ratio between energy after and before a bounce, or equivalently, between the maximum height before and after a bounce), the velocity 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

Let θ be the angle between global forward (the y-axis in **Figure 2**) and the current orientation of the robot (the direction specified by the line labeled "F" on the robot in **Figure 2**). Let α be the angle between global forward and the desired heading (the green line in **Figure 2**). Let $M1$, $M2$, etc. be the magnitude and

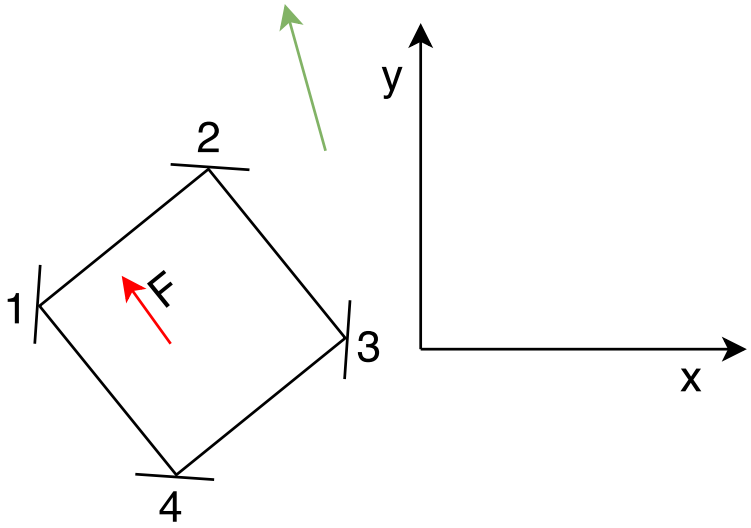


Figure 2: **Global directions for omni-drive theory.**

direction of motors 1, 2, etc., where positive values indicate forward direction (the direction required to make the robot move forward relative to the robot's orientation), and negative indicate backwards. The range of these values then is $[-1, 1]$.

The simple drive case is the one in which the robot is not rotating. In this case, $M1 = M3$ and $M2 = M4$. If we switch to a coordinate system aligned to the axis of the motors as in **Figure 3**, then it is clear that $\tan\beta = \frac{M2}{M1}$, where β is the desired heading of the car relative to the new coordinate system. In terms of θ

and α, then, $\beta = \alpha - \theta + \frac{\pi}{4}$, so

$$\tan(\alpha - \theta + \frac{\pi}{4}) = \frac{M2}{M1}.$$

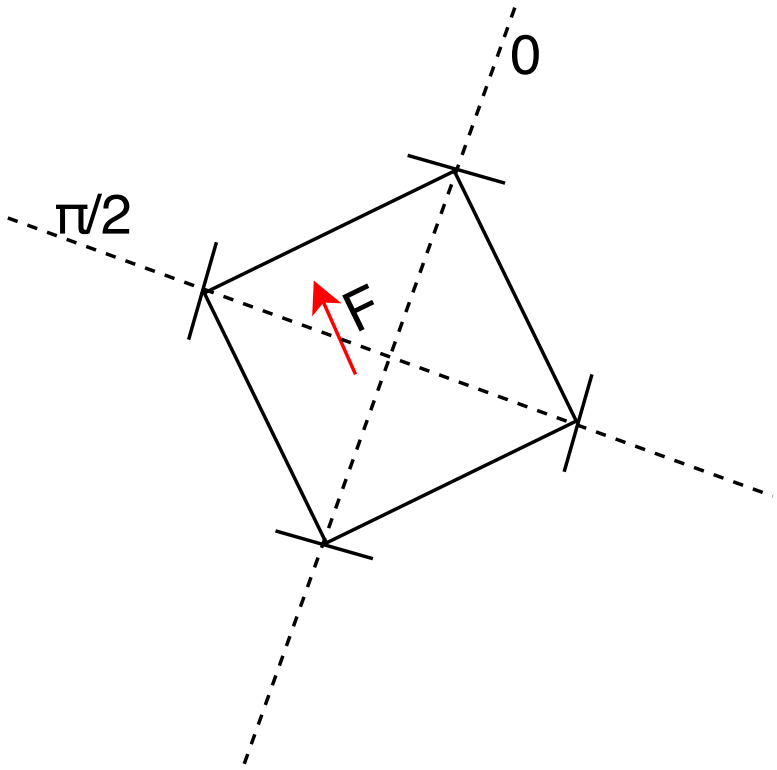


Figure 3: **Relative directions for omni-drive theory.**

Allowing rotations of the car only complicates this slightly. Rotations can be achieved by simply offsetting the speeds of $M1$ and $M3$ relative to each other, and the same for $M2$ and $M4$. Let s be the speed of rotation from $[-1, 1]$. Then

$$s = \frac{M2 - M4}{2} = \frac{M3 - M1}{2}.$$

When the car is rotating, the average of $M2$ and $M4$ remains constant for any s and likewise for $M1$ and $M3$, so our original equation can now be rewritten as

$$\tan\left(\alpha - \theta + \frac{\pi}{4}\right) = \frac{(M2 + M4)/2}{(M1 + M3)/2} = \frac{M2 + M4}{M1 + M3}.$$

The final equation of motion is simply a constant that $\max\{M1, M2, M3, M4\} = M$, where M is the magnitude of movement in the direction specified by β. Together, these three equations, functions of α, θ, s, and M, fully specify the control of the four wheels.

2.4 Accelerometer

An accelerometer measures acceleration relative to its own orientation. Therefore, as the robot spins and the accelerometer spins with it, the accelerometer will not be measuring acceleration in a consistent direction. To get meaningful readings from the accelerometer, we need to use information about our orientation from the gyroscope to correctly project the accelerometer's acceleration readings to global, constant axes. Further, the accelerometer is not mounted at the center of the robot, so as the robot spins, centrifugal

force will add a false acceleration to the accelerometer. This acceleration needs to be subtracted off to get accurate readings.

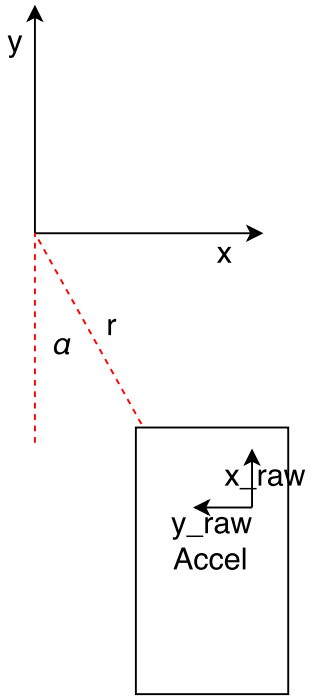
Let x_{raw} and y_{raw} be the original readings of the accelerometer. First, we need to subtract off the effect of centrifugal force. As in **Figure 4(a)**, let the angle between the negative y-axis and the accelerometer be α, the distance between the center of mass of the robot and the accelerometer be r, and the rate of rotation be ω. Then

$$x_1 = x_{\text{raw}} + r\omega^2 \cos\alpha$$

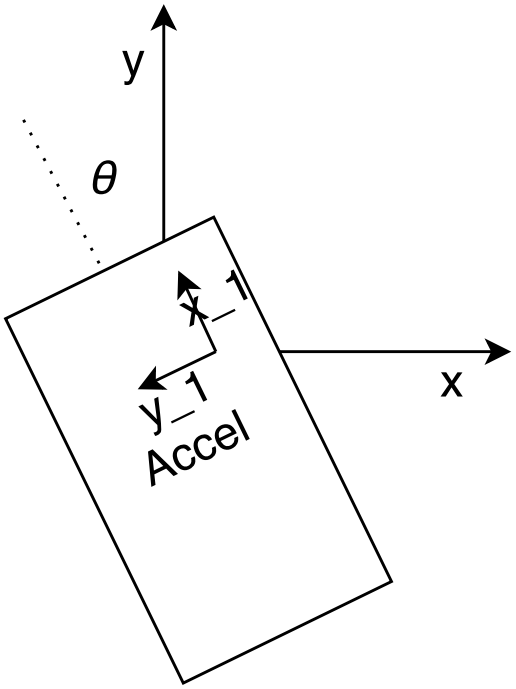
and

$$y_1 = y_{\text{raw}} + r\omega^2 \sin\alpha,$$

where x_1 and y_1 are the adjusted accelerations.



(a) Definitions for centrifugal force corrections to accelerometer readings.



(b) Angle definitions for accelerometer theory.

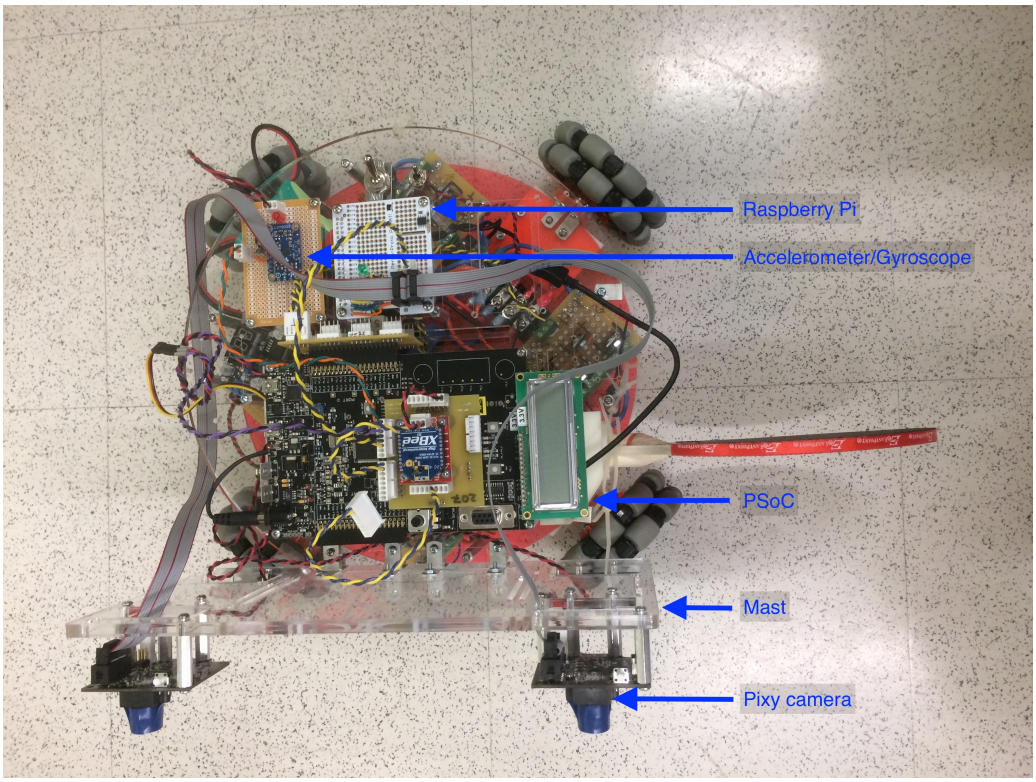
Figure 4: **Definitions for accelerometer theory.**

Let θ be the angle between the accelerometer and global coordinates as in **Figure 4(b)**. Then $y = x_1 \cos\theta - y_1 \sin\theta$ and $x = -x_1 \sin\theta - y_1 \cos\theta$, where x and y are global x and y accelerations.

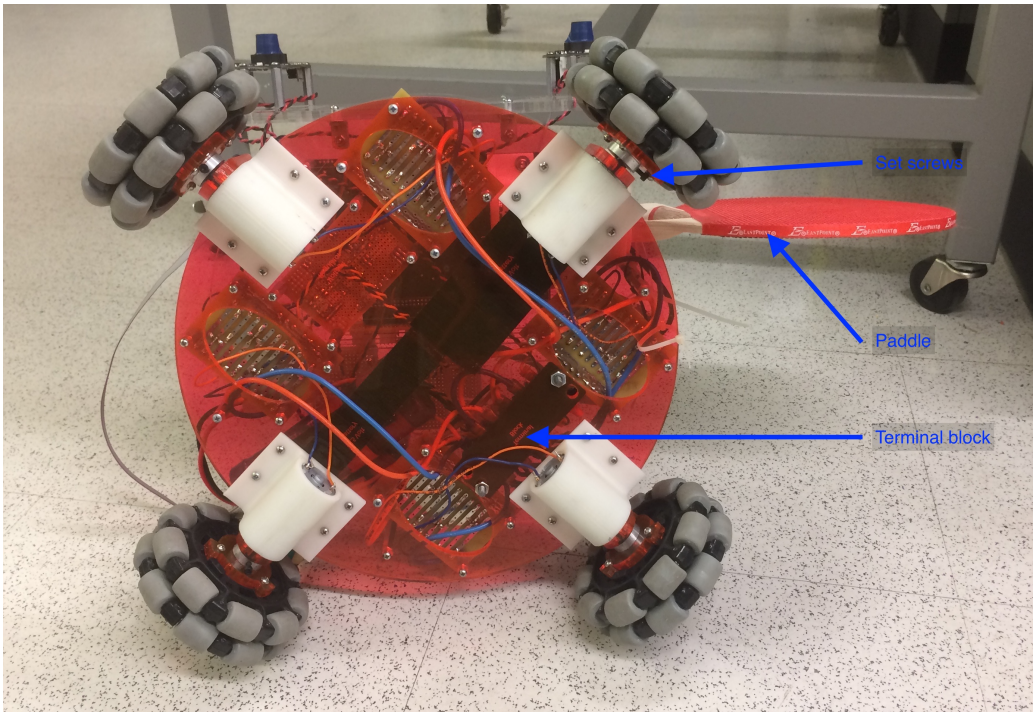
With these calculations, the global x and y accelerations can be double integrated to determine the position of the car at any time.

3 Hardware

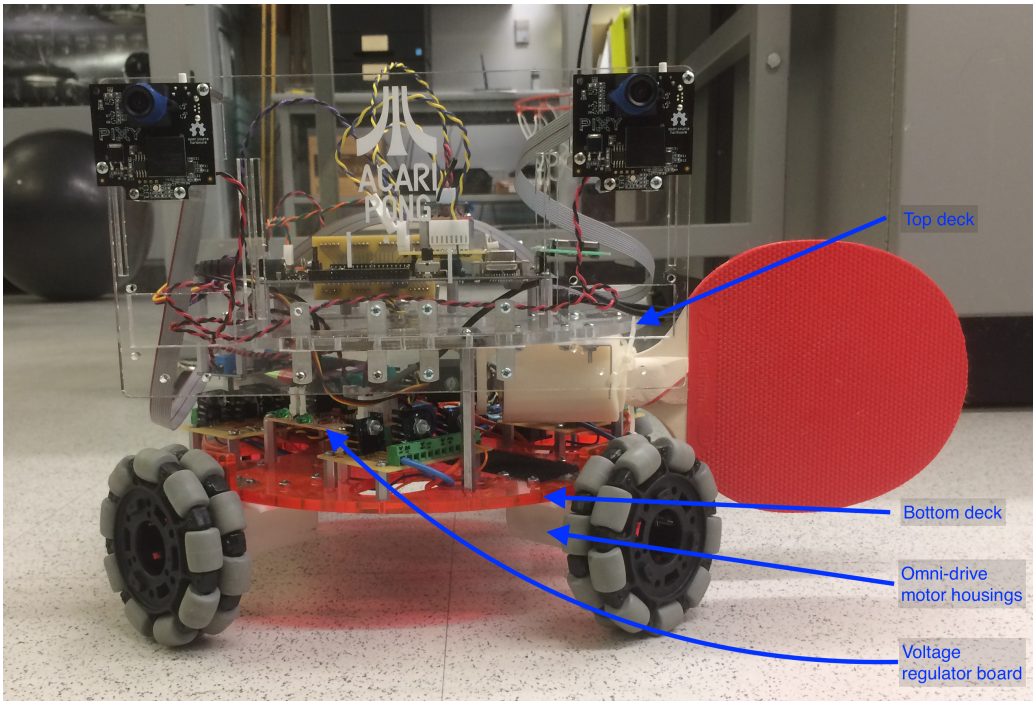
Figure 5 shows photographs of the final product from various angles, annotated with important components.



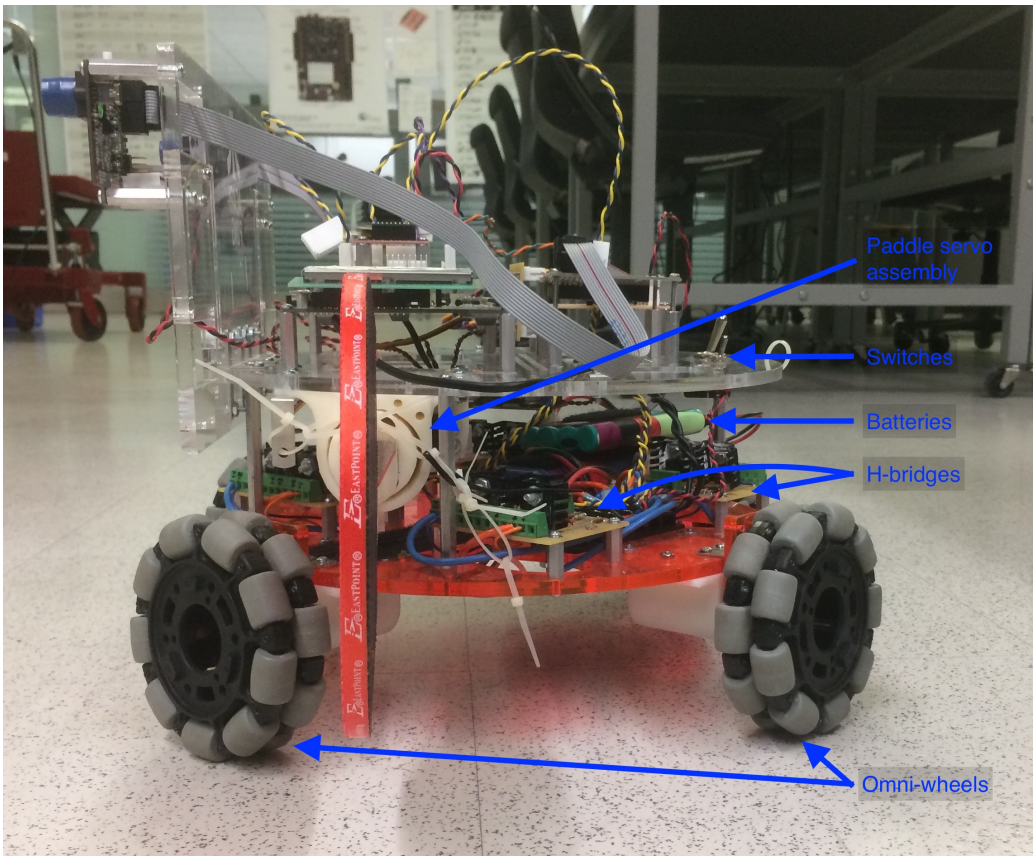
(a) Top view.



(b) Bottom view.



(c) Front view.



(d) Side view.

Figure 5: **Photographs of the final robot.**

Figure 6 shows a block diagram of the hardware set-up.

Figure 6: **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

The chassis consists of two decks of components: a lower deck for the motors, H-bridges, and power (voltage regulator) board; and an upper deck for the Raspberry Pi, PSoC, accelerometer/gyro, and mounting points for the mast, which contains the two Pixy cameras. Since we inherited Gordon and Pfleger's chassis from 2016, we re-used their lower deck after aesthetically modifying the H-bridges, replacing bent set screws from the wheel hubs, and replacing the power board with our own voltage regulator board. On the other hand, we completely re-designed the top deck to suit our purposes. See **Figure 7** for the laser cutting template for the top deck:

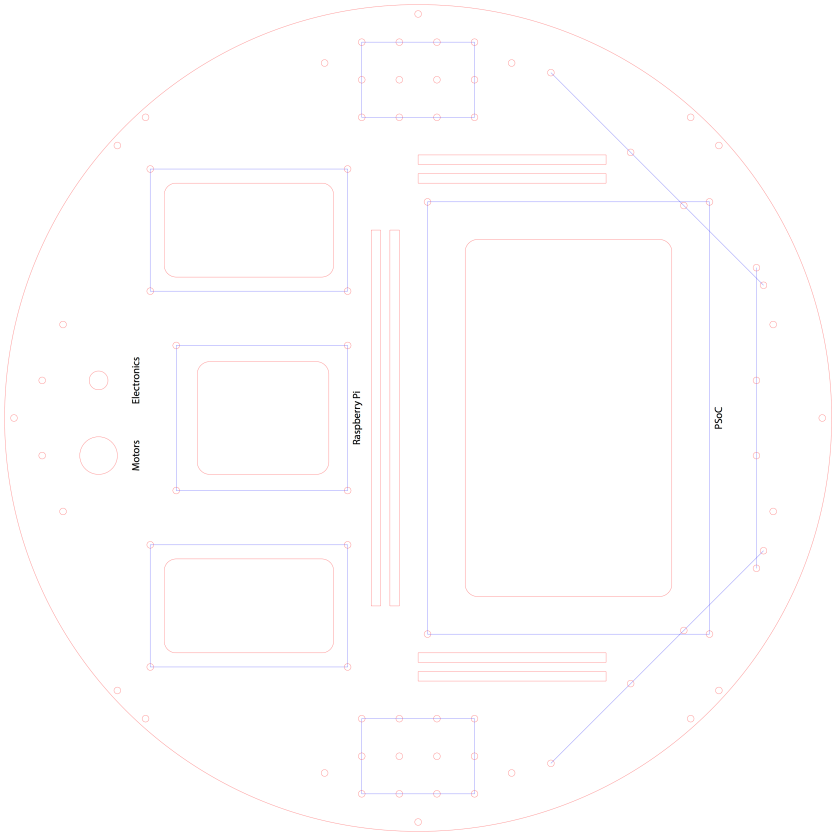


Figure 7: **Laser cut template for top deck.**

We adapted the voltage regulator board from the speed control and navigation assignments: it has two linear 5 V voltage regulators (LM7805) and many mounting points for both unregulated and regulated

voltages (see **Section 3.6** for details).

3.2 Omni-wheel drive

Because the ping-pong ball can land anywhere in space, we needed a robot that could quickly move anywhere in space. Because it is very difficult to do this with conventional steering systems, we decided to use an omni-directional drive (or omni-drive). An omni-drive consists of four wheels mounted at right angles to each other. These wheels have rollers along their circumference, allowing them to be pushed orthogonally to their direction of rotation. The rollers allow the robot to move in any direction, and the fact that the wheels can be independently controlled allows the sum of the forces to point in any direction. Together, this translates to a robot that can drive in any direction.

3.2.1 Motors and chassis

As mentioned previously in this report, we used an omni-drive chassis that had already been built last year by Ethan Gordon and Luke Pfleger. The main body is laser cut acrylic, with four 3D printed and acrylic motor mounts along the sides. The motors are 22 mm diameter, 20.4 : 1 geared, high power Pololu Gearmotors, and the wheels are 4" VEX omni-wheels. All four motors are run off a single 7.2 V NiCd battery. We made no real changes to this chassis, besides redoing some of the wiring and replacing the set screws for the wheels, as mentioned in the previous section.

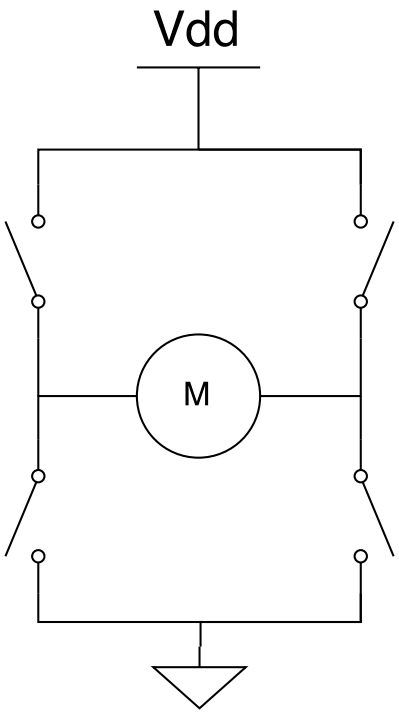
3.2.2 Motor controllers

An omni-drive requires 4 independent controllers capable of driving forwards and backwards and controlling throttle. To do this, we use H-bridges. H-bridges use four transistors to completely select power and ground for each terminal of the motor as shown in **Figure 8(a)**. By turning on different transistors, the H-bridge can direct current through the motor in either direction, and can brake the motor by connecting both terminals to either power or ground. Since the switching is controlled by transistors, they can be given PWM signals, which allows for throttle control.

This basic idea is very simple, but there are many complications in the design of an H-bridge. The first of these arises from our use of PWM signals. A PWM signal has both an "on" period and an "off" period. During the "on" period, one side of the motor must be connected to ground and the other must be connected to power. To make things concrete, let's assume we are driving the motor in a direction such that the left side is connected to ground and the right is connected to power. The question, then, is what to do during the "off" period. Since the motor has inductance, it is a bad idea to completely disconnect the motor during the "off" period, as this will force the back EMF to drop across the internal diodes of the transistors. Our remaining options then, are to reverse the motor during the "off" period, or to brake it. Reversing the motor during the "off" period is known as lock anti-phase drive, and braking it is known as phase-magnitude drive.

The second complication with H-bridges is that if two transistors on the same side of the H-bridge are ever turned on at the same time, they create a direct short from power to ground. The H-bridge, then, must be designed in such a way that two same-side transistors cannot be on at the same time.

The chassis we used also had H-bridges that had already been built. The circuit diagram and board image is shown in **Figure 8(b)**. NMOS transistors are used on the ground side, and PMOS transistors are used on the power side. Each gate has an inline 1.1 kΩ resistor to protect the PSoC from back EMF spikes. The left side gates are wired directly together, as are the right side gates. Given a low signal, then, the PMOS will turn on and the NMOS will turn off, connecting the motor terminal to power. Given a high signal, the PMOS will turn off and the NMOS will turn on, connecting the motor terminal to ground. Since



(a) H-bridge dummy circuit diagram.

(b) Actual H-bridge circuit diagram.

Figure 8: **H-bridge circuit diagrams.**

we never want to motor to be floating (as that causes the back EMF to drop across the FET diodes), this arrangement works well.

Since we are operating the motors at 7.2, the PMOS transistors need an input close to that to turn off. The PSoC drives its outputs at 3.3 which is not nearly high enough to achieve this. The original design used open drain comparators with a threshold set at 1.5 to achieve the higher voltage. The 750 Ω pull-up resistors in the circuit diagram act as the pull-ups for the open-drain outputs. Since the PSoC has its own internally configurable open drain outputs, we simply used those to directly drive the H-bridges, eliminating the original comparators. The pull-up resistors also serve to connect both terminals of the motor to ground in the event of a PSoC failure, creating a safe failure condition.

Gordon and Pfleger used lock anti-phase drive, and this influenced several of their H-bridge design decisions. Since the "off" time PWM signal is in reverse, the stop condition for the motors is a 50%, duty cycle, such that the motor is connected forwards half the time and backwards the other half of the time. According to their report, the first time they tried this, with a fairly standard 100 Hz PWM, they burned out two motors. To avoid burning out their motors, they had to increase their frequency to about 10 kHz This high frequency required making the pull-up resistors very small, to make the time constant of the transistors' gate capacitances low enough to allow them to actually turn on and off with the input signal. We believe that the main cause of their motors burning out was their choice of phase anti-lock drive. The frequency was high enough that the motors did not turn, as intended, but was also low enough that there was enough time for the current to reach full stall current in each direction, thereby burning out the motors.

We chose to use phase-magnitude drive, if only for the simple reason that it is more efficient to brake the motor during the "off" time than it is to reverse it. This also came with the benefit that we believe we could have safely run the motors at a much lower PWM frequency, as the stop condition for phase-magnitude drive is simply braking without any PWM signal, so nothing can stall. We only apply a PWM signal when the motor is actually supposed to turn, so any frequency should be safe. If the motor were to

actually stall, it is possible that the high frequency would help protect the motor, but otherwise, we believe a lower frequency would have been safe. However, we decided to leave the frequency at 10 kHz, since we knew that was absolutely safe. We did occasionally see some problems where the motor would have trouble starting at low speeds (for example, going from off to a 30% duty cycle, the motor would occasionally stall briefly and require a push to get going). A lower frequency might have given the motor more sustained current and allowed it to start more reliably at lower speeds.

The specific details of the control signals `C1` and `C2` required to drive the H-bridge in sign-magnitude mode will be given when we discuss the hardware programmable hardware required to implement them in **Section 3.7.1**.

3.3 Dead reckoning

Since the very principle of an omni-drive's motion is dependent on the wheels slipping, we cannot track the wheels' motion to determine the position of our robot. Instead, we use dead reckoning. To implement dead reckoning, we use a 9-DOF combined accelerometer, gyroscope, and magnetometer, the LSM9DS0 chip from ST Microelectronics, mounted on an Adafruit breakout board. This chip communicates with the PSoC through I^2C.

3.4 Cameras and mast

The CMUcam5 (Pixy camera, or simply "Pixy") is a low-cost computer vision system developed by CMU and Charmed Labs that can be easily trained to recognize colored objects. The Pixy uses the OV9715 CMOS WXGA (1-MP) image sensor, which supports 1280×800 resolution at 25 fps or 640×400 resolution at 50 fps. Because of the memory limitations of the on-board microcontroller and to achieve the higher frame rate, the CMUcam5 default firmware restricts the resolution to 640×400. Furthermore, the object detection algorithm outputs object coordinates and dimensions at 320×200 resolution.

The Pixy outputs its data either through USB (which can interface with the Raspberry Pi with the `libpixyusb` library provided), or various serial protocols such as SPI, I^2C, and UART (it can even output as raw analog/digital). Initially, we attempted to interface with the Pixy cameras using SPI due to its speed, but because of bus collisions with two cameras (even with the chip-enable functionality), we decided to use I^2C with two addresses.

The Pixy cameras were mounted 8.5 inches apart on the mast at the front of the robot to permit stereo vision, and were connected to the Raspberry Pi with ribbon cables. See **Figure 9** for the laser cut template of the camera mast. We included four possible mounting positions for the Pixy cameras: the top two were the intended mounting positions, but if the cameras were too high to detect the ball, we would have moved the cameras to the bottom two mounting positions. With the cameras on the top mounting positions, the cameras were approximately 260 mm above the ground.

3.5 Paddle mount

We wanted to be able to control the angle of the ping-pong paddle as we hit the ball, so we needed to design a mount for the paddle that would allow a servo to rotate it. We used a Hitec HS-322HD servo, a very common hobby servo. We did not want the servo to be subjected to shear forces from the paddle, so our mount needed to constrain the paddle's position, and only allow it to rotate. We decided the best way to do this would be to encase the servo's handle in a cylinder, and then let that cylinder rotate within another cylinder. The first cylinder evens out the surface of the handle, and then the second constrains its motion to just rotation.

The inner mount was made in two halves so that we could simply screw each half into the handle. See **Figure 10** for images of all the parts. The servo mounts right in line with the paddle, and directly drives its

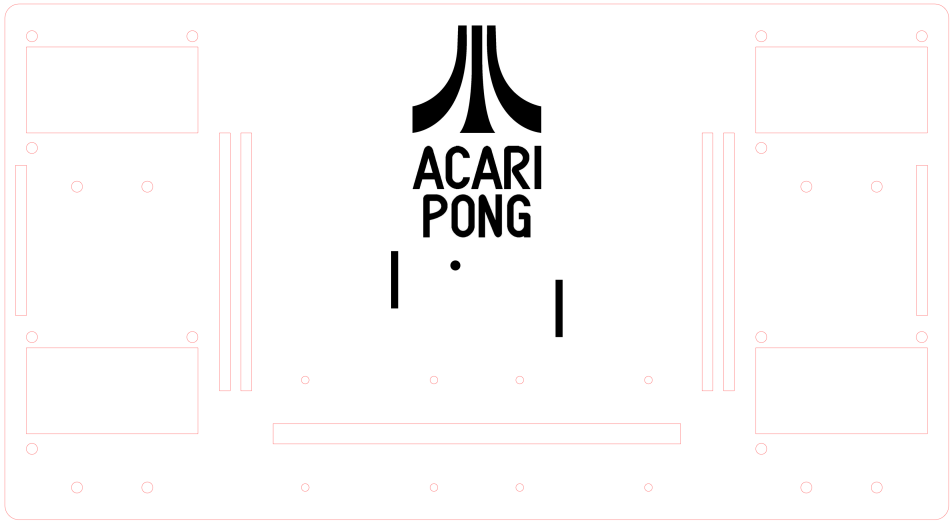


Figure 9: **Laser cut template for camera mast.**

rotation. Once the servo is attached to the mount, the paddle simply slides into the mount and connects to the servo. We included two holes through the mount to allow a faceplate to be attached after the paddle is slid in, to stop the paddle from sliding out when the robot rotates. However, we did not have long enough screws or threaded rods to attach such a faceplate, so we instead ran some zip-ties across the face to secure the paddle.

Figure 10: **Images of 3D-printed paddle mount parts.**

The mount had to fit in a very tight area, so to avoid having to go through many iterations as we discovered clearance issues, we made a mock model of our robot that included everything we thought could be a clearance issue. The mock robot model is shown in wireframe in **Figure 11**, with the inner mount in green, the outer mount in blue, and the servo in red. This assembly allowed us to successfully design the mount on our first try.

Figure 11: **Wireframe mock model of robot for paddle mount assembly.**

3.6 Power supply

To power the electronics of the robot (all but the four omni-drive motors), we used a single 9.6 V NiMH battery. Two linear voltage regulators provided power for the accelerometer board and the paddle servo motor (one regulator each), and unregulated 9.6 V was provided to the PSoC (which had an internal regulator), the two Pixy cameras (which had their own regulators), and the Raspberry Pi via an adjustable output buck (DC-DC step-down) converter.

We initially attempted powered the Raspberry Pi with regulated 5 V through the GPIO pins, because the 5 V pins and ground are shared across the whole device. This is highly discouraged these rails are not protected, while the micro USB port is protected with a polyfuse (a resettable fuse). We quickly decided to adapt a spare micro USB cable so that the Raspberry Pi could be powered through the USB port. We also

quickly discovered that the Raspberry Pi drew too much current (about 0.1 A when idle, and up to about 0.6 A when booting or compiling C++) to power from a linear voltage regulator: with a voltage drop of 4.6 V and a current of 0.5 A, a linear voltage regulator would dissipate about 2.3 W of power when booting; the voltage regulator would heat up unacceptably even with a heatsink, and it depleted the batteries too quickly for acceptable usage. Because of this, we powered the Raspberry Pi through a Drok LM2595 analog control voltage controller (a DC-DC step-down converter). Conveniently, this converter also doubled as a battery level meter for the 9.6 V battery.

3.7 Programmable hardware

Figure 12 shows the top-design of our programmable hardware in PSoC Creator 2.1.

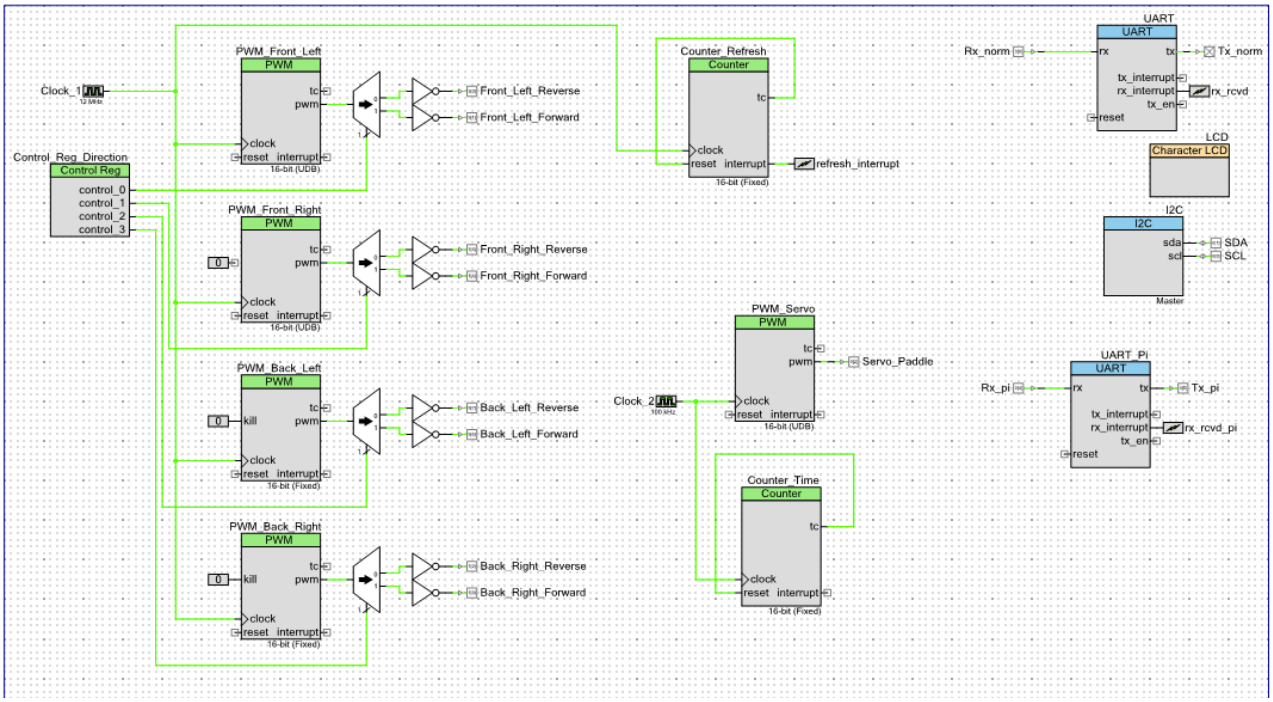


Figure 12: **Top-design of the ping pong project.**

3.7.1 Hardware for motor control

Phase-magnitude drive requires a PWM signal where the "on" time signal drives the motor in the intended direction and the "off" time signal brakes the motor. Since the H-bridge circuit as described in **Section 3.2.2** defaults to connecting each motor terminal to ground, this can be achieved by simply sending a normal PWM signal to one of the inputs `C1` or `C2` and letting the other input remain pulled high by the H-bridge circuit. When the PSoC drives `C1` low, the left terminal is connected to power and the motor goes forward, and when the PSoC releases `C1`, the left terminal is pulled back high an the motor brakes. To go in reverse, the PSoC simply needs to drive `C2` instead of `C1`.

This, then, breaks the control into two very natural signals, direction and magnitude. The PWM signal controls the magnitude of rotation, and changing which terminal it goes into controls the direction of rotation. We implemented this scheme in the programmable hardware with PWM blocks and demultiplexers (i.e., switches). There is one PWM block per motor, and each PWM block outputs into a demultiplexer which selects between forward and reverse based on a software control signal. The software control signals come from the `Control_Reg_Direction` block. This is the complete picture of the control at a high

level, but a couple more additions were required to make it actually work. The final control signal outputs of the PSoC are open drain, so "on" corresponds to a low signal and "off" corresponds to a high signal. Demuxes output low on all unused inputs, which is the opposite behavior to what is desired, so we had to invert all of the demux outputs to make them default to floating rather than pulling low. This also had the convenient effect of making the high side of the PWM signal correspond to driving the output low, which is what turns the motor on, so on the software side, the duty cycle maps directly to throttle rather than inversely.

3.7.2 Other hardware

The servo that controls the angle of the paddle is directly controlled by a single PWM block, `PWM_Servo`. We continued to use a UART block to communicate with our computers for debugging, and added a second, `Serial_Pi` to communicate with the Raspberry Pi. We also have a master I2C block that manages communication with the accelerometer/gyro chip. There is also a counter block, `Counter_Refresh`, which is used as a refresh timer. It interrupts at a rate of 1 kHz. Finally, we have another counter, `Counter_Time`, on a slower clock, that is used to accurately keep track of time.

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

Initially, to implement SPI communication with a single Pixy camera, we adapted the code from this GitHub repository: `https://github.com/omwah/pixy_rpi`. This has an implementation of SPI communication using `wiringPiSPI.h` from the `WiringPi` library by Gordon Henderson, as well as a general Pixy data processing interface using the object block format found in the Porting Guide of the Pixy camera: `http://cmucam.org/projects/cmucam5/wiki/Porting_Guide`. The files are organized such that `Pixy.h` defines the class `LinkSPI`, which abstracts the communciation protocol with the SPI interface, `TPixy.h` defines the class `TPixy`, which abstracts the object block protocol further and maintains a buffer of received blocks, and `echo.cpp` uses the `TPixy` class defined in `TPixy.h` to echo blocks to standard output. Abstracted in this way, the client code becomes very simple:

```
#include <Pixy.h>

Pixy pixy;

int main() {
    int j;
    uint16_t blocks;
    while (true) {
        blocks = pixy.getBlocks();
        if (blocks) {
```

```
            std::cout << "Detected " << blocks << std::endl;
            for (j = 0; j < blocks; j++) {
                std::cout << "  block " << j << " ";
                pixy.blocks[j].print();
            }
        } else {
            std::cout << "No blocks detected" << std::endl;
        }
    }
    return 0;
}
```

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**.

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.

7 Further work

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.**

	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**

8 Appendix: full listings

Contents

8.1 `main.c`