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.

Contents

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:

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

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

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

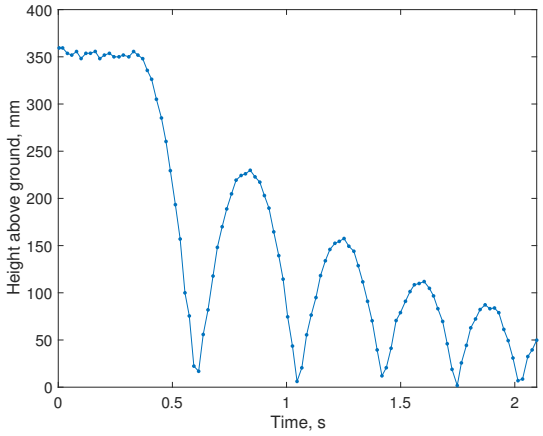
β_{x0}, β_{y0} and β_{z0} all represent the initial positions (at $t = 0$); β_{x1}, β_{y1} and β_{z1} all represent the initial velicities (at $t = 0$); and β_{z2} represents the acceleration due to gravity.

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, $\boldsymbol{\beta}$:

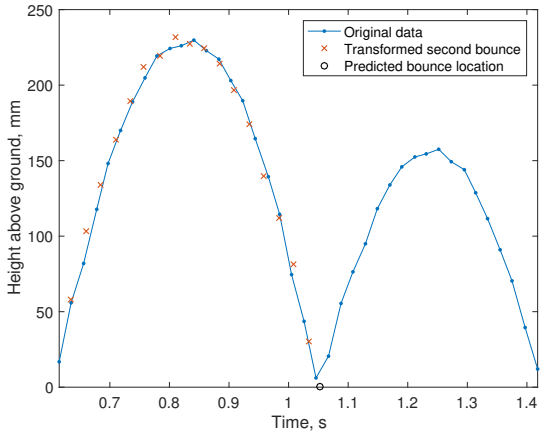
$$\hat{\boldsymbol{\beta}}^{\text{OLS}} = \left(\mathbb{X}^T \mathbb{X}\right)^{-1} \mathbb{X}^T \mathbf{Y}.$$

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



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



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

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

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: $z \sim (\beta_0 + \beta_1 t + \beta_2 t^2)/R$. 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.

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 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

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

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

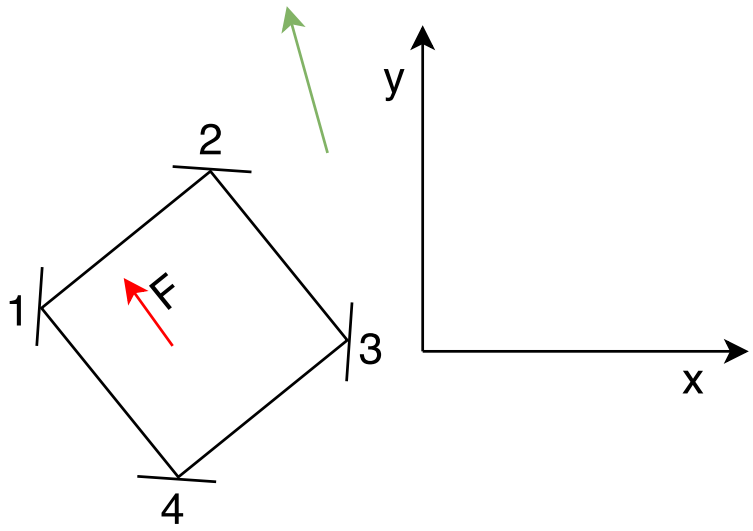


Figure 2: **Global directions for omni-drive theory.**

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.

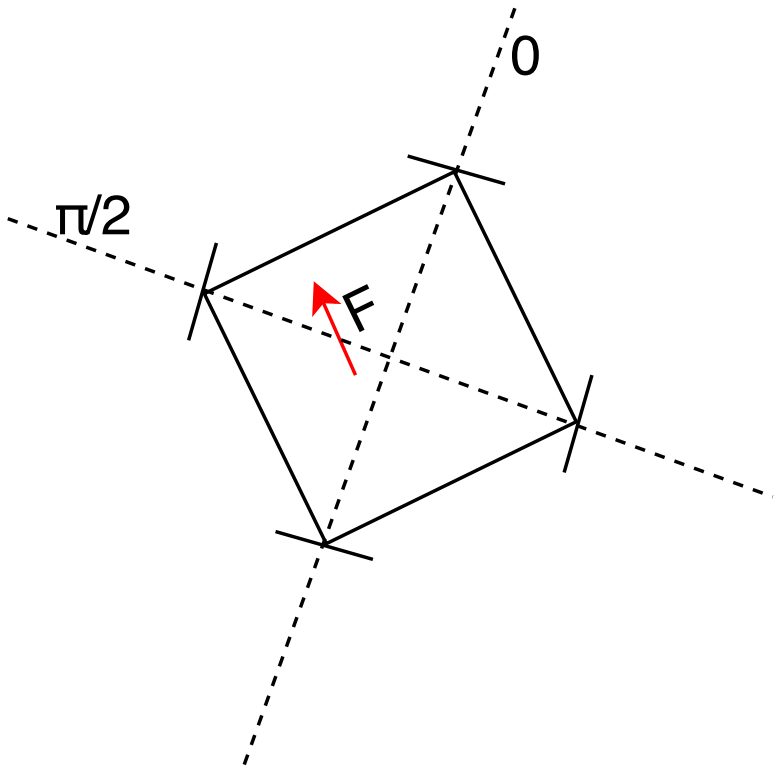
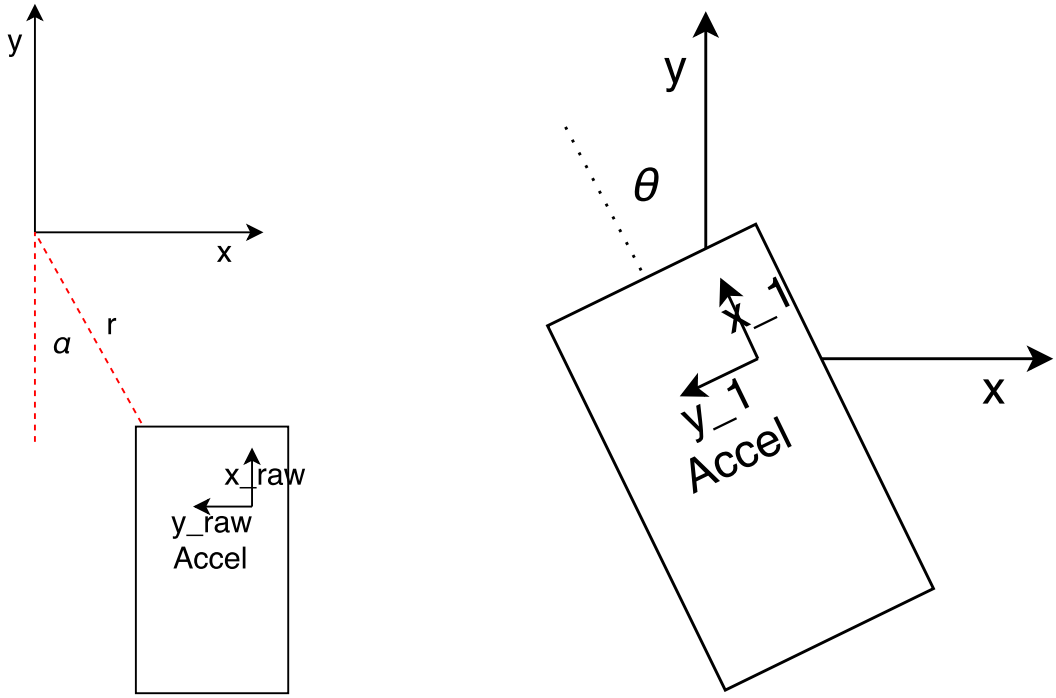


Figure 3: **Relative directions for omni-drive theory.**

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.



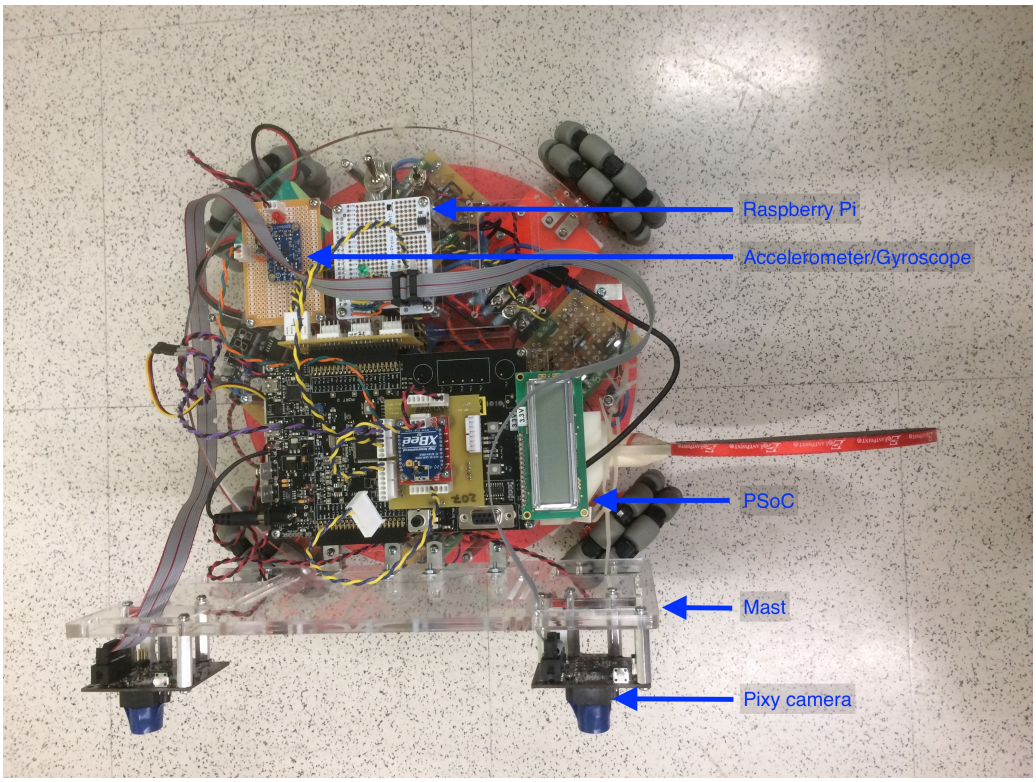
(a) Definitions for centrifugal force corrections to accelerometer readings.

(b) Angle definitions for accelerometer theory.

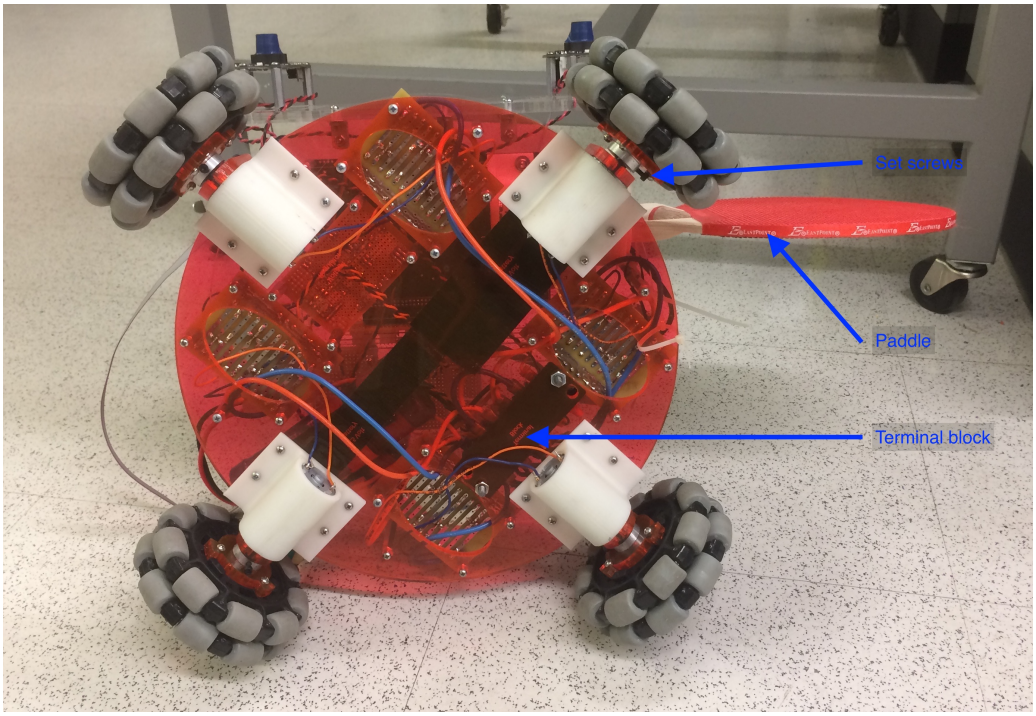
Figure 4: **Definitions for accelerometer theory.**

3 Hardware

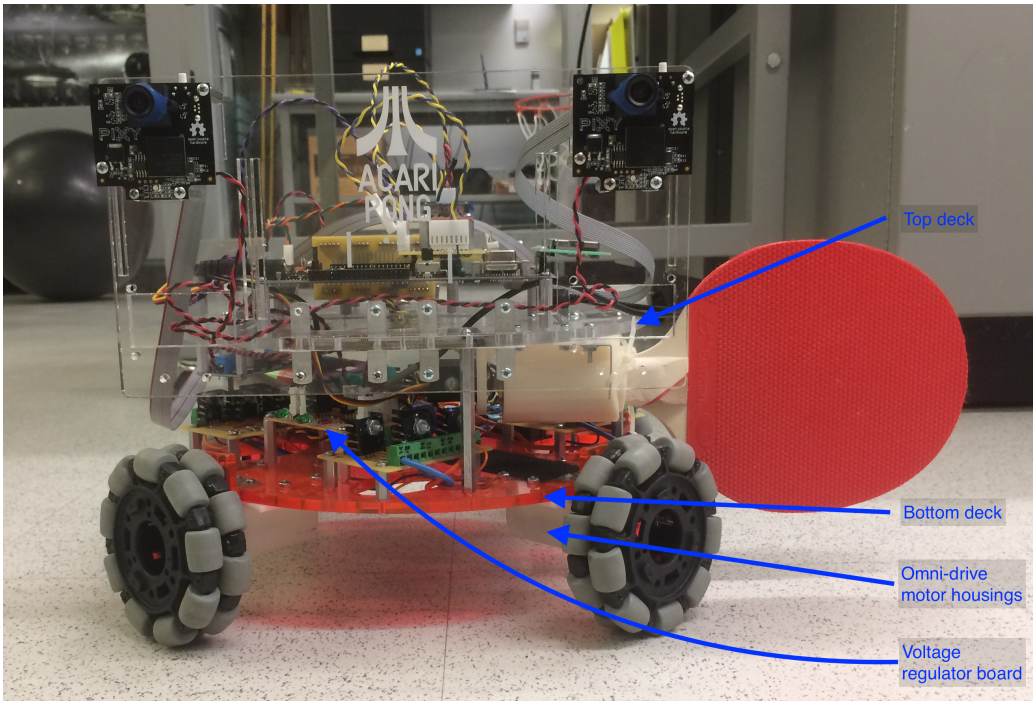
Figure 5 shows photographs of the final product from various angles, annotated with important components.



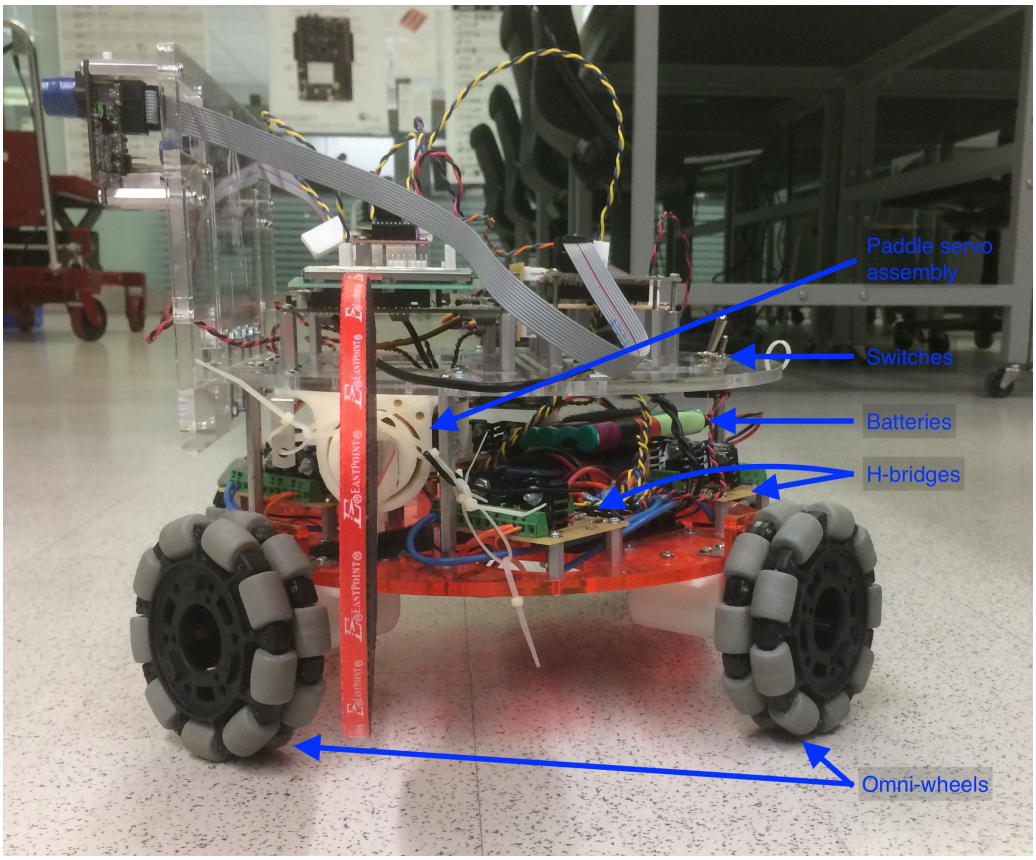
(a) Top view.



(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.

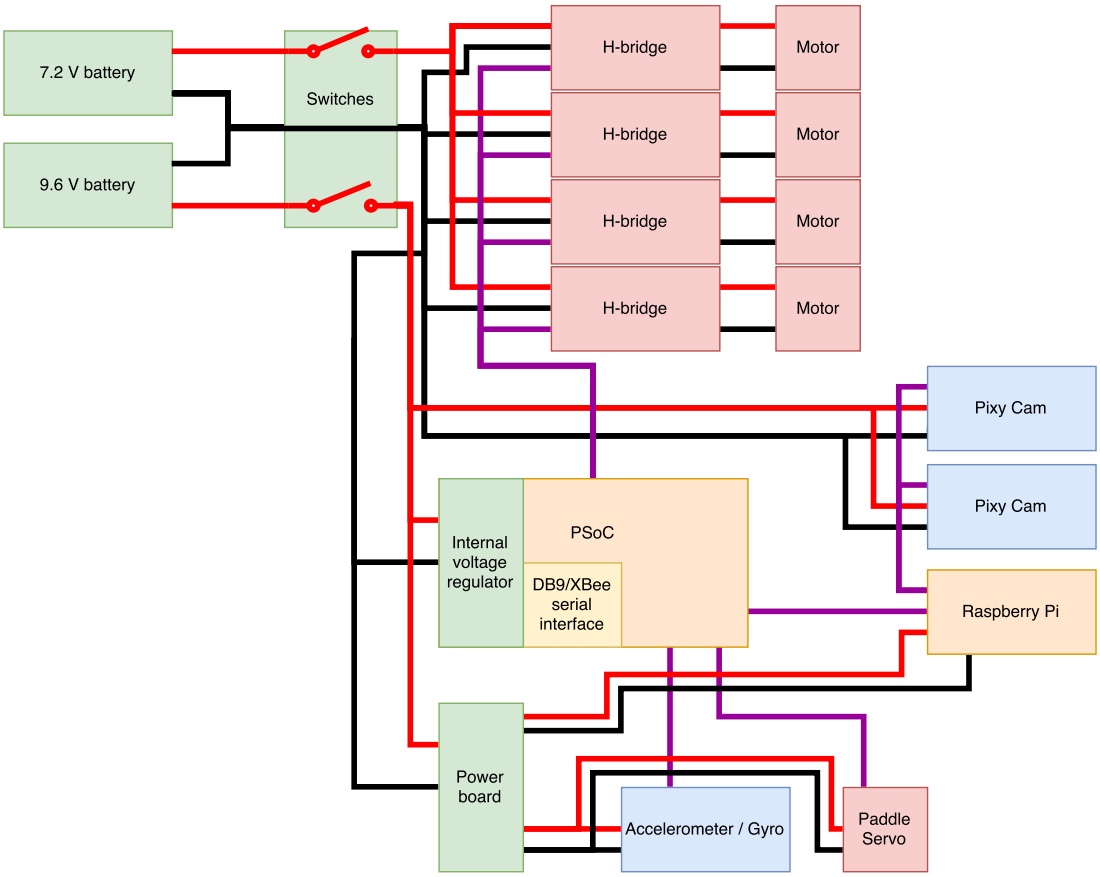


Figure 6: **Block diagram of the hardware setup for ping pong.** Black connections indicate ground connections; red connections indicate power (both regulated and unregulated); all other connections indicate data connections (e.g. from the PSoC to the H-bridges and servo, various I^2C connections). Green 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:

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).

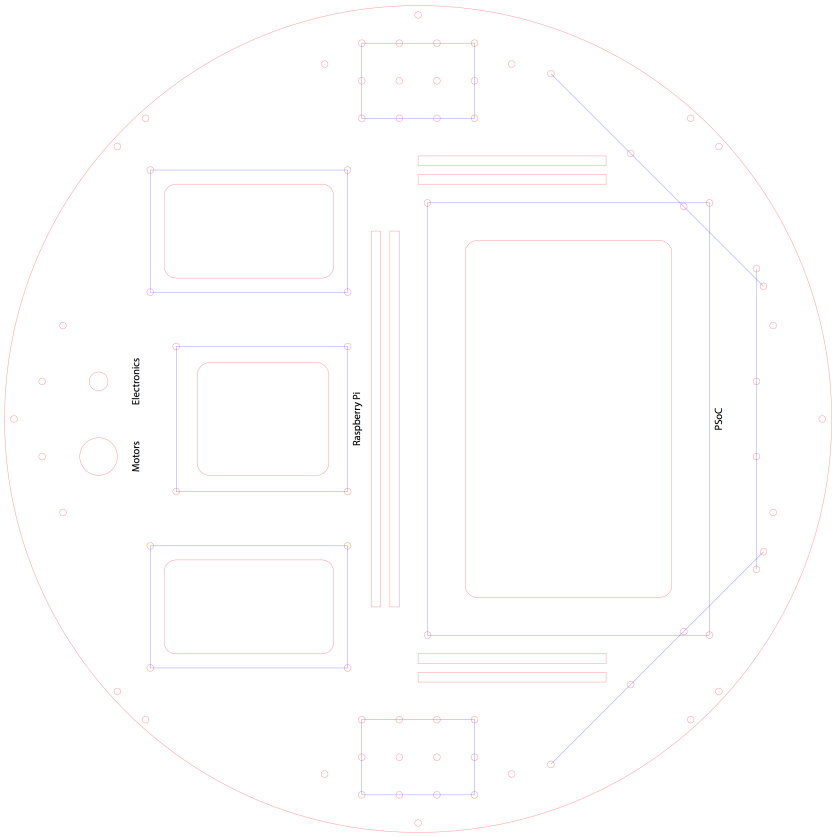


Figure 7: **Laser cut template for top deck.**

3.2 Omni-wheel drive

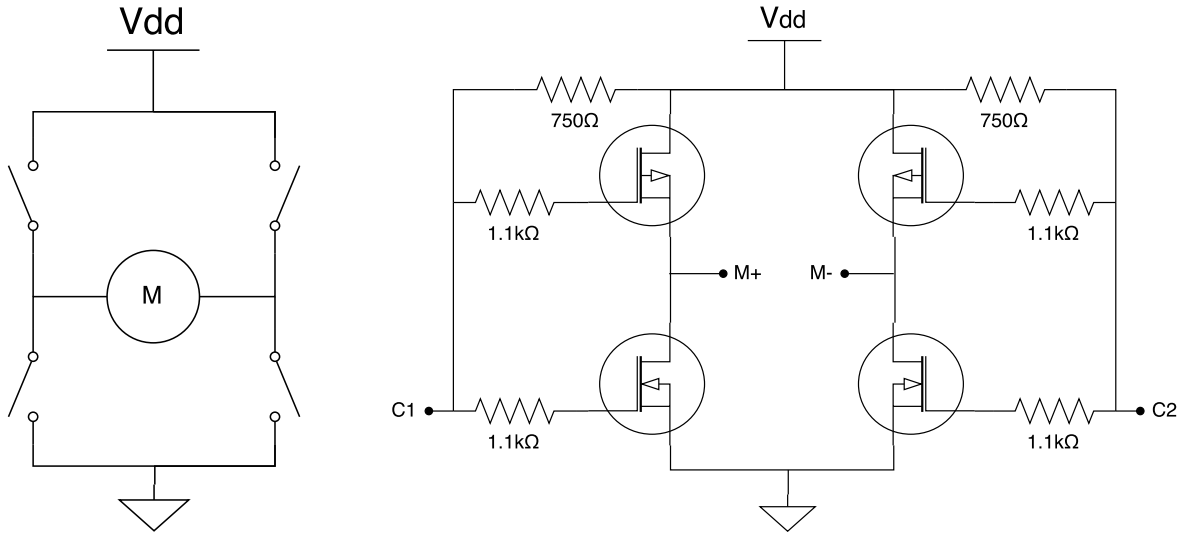
Because the ping-pong ball can land anywhere in space, we needed a robot that could quickly move anywhere in space. Because it is very difficult to do this with conventional steering systems, we decided to use an omni-directional drive (or omni-drive). An omni-drive consists of four wheels mounted at 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.



(a) H-bridge dummy circuit diagram.

(b) Actual H-bridge circuit diagram.

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

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 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.

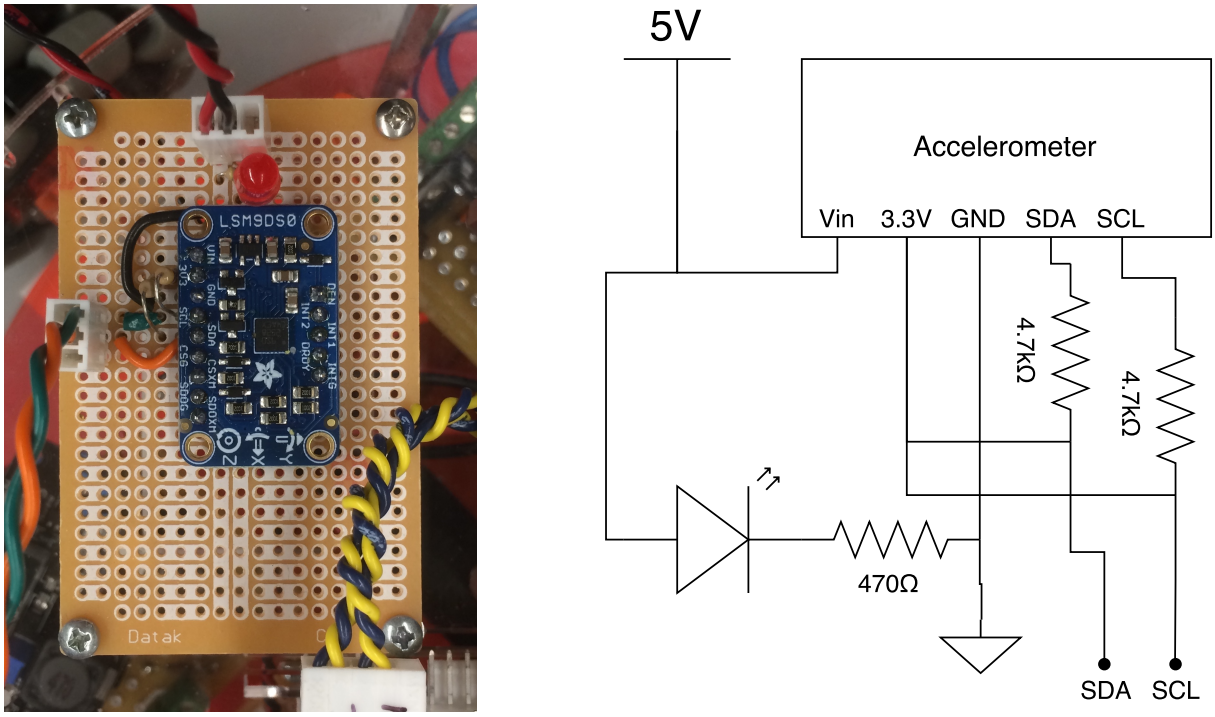
The specific details of the control signals `C1` and `C2` required to drive the H-bridge in sign-magnitude mode will be given when we discuss the hardware programmable hardware required to implement them in **Section 3.7.1**.

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. See **Figure 9** for a photograph and a circuit diagram of the accelerometer/gyroscope board:

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.



(a) Photograph of the accelerometer board.

(b) Circuit diagram of the accelerometer board.

Figure 9: **Detail of the accelerometer board.**

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 10** 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 11** for images of all the parts. The servo mounts right in line with the paddle, and directly drives its 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

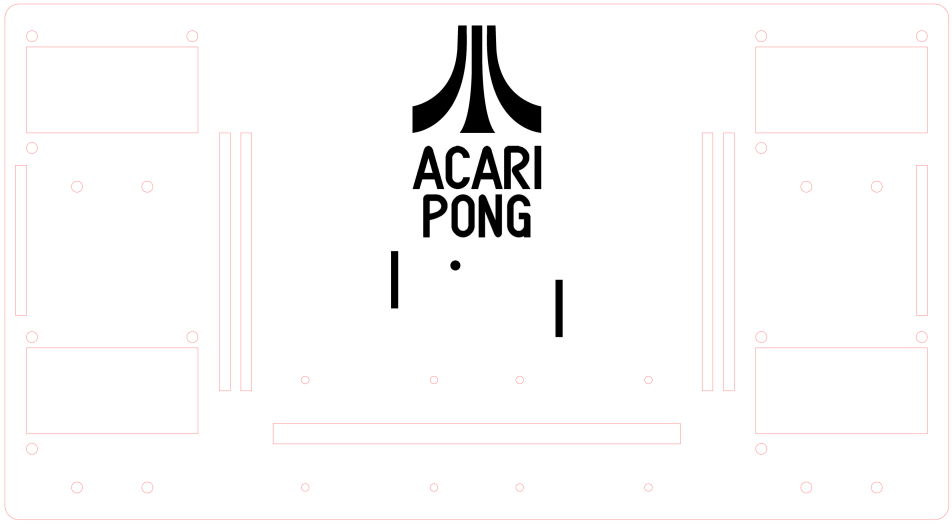


Figure 10: **Laser cut template for camera mast.**

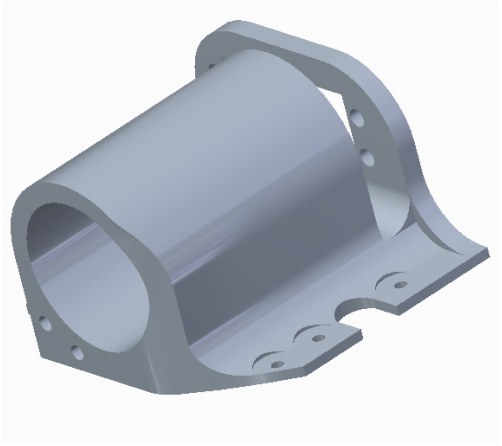
slid in, to stop the paddle from sliding out when the robot rotates. However, we did not have long enough screws or threaded rods to attach such a faceplate, so we instead ran some zip-ties across the face to secure the paddle.

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 13**, 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.

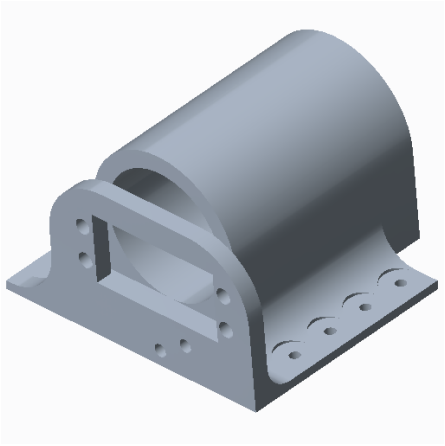
3.6 Power supply

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.

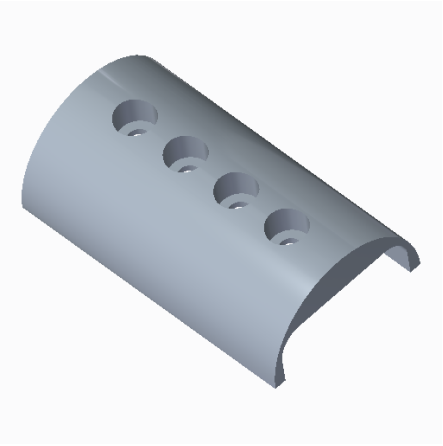
We initially attempted powered the Raspberry Pi with regulated 5 V through the GPIO pins, because the 5 V pins and ground are shared across the whole device. This is highly discouraged these rails are not protected, while the micro USB port is protected with a polyfuse (a resettable fuse). We quickly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



(a) Outer cylinder mount, front diagonal view.



(b) Outer cylinder mount, back diagonal view.



(c) One half of the inner cylinder mount for the paddle.

Figure 11: **CAD images of 3-D-printed paddle mount parts.** Models were created in Creo and exported to `.stl` files for 3-D printing.

3.7 Programmable hardware

Figure 14 shows the top-design of our programmable hardware in PSoC Creator 2.1.

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 sig-

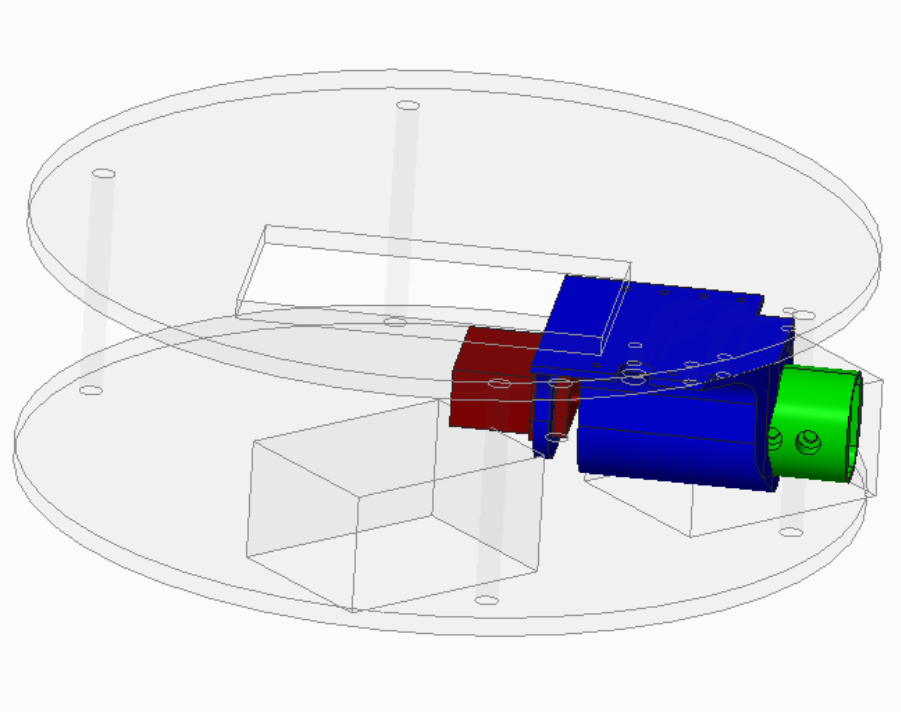


Figure 12: **Wireframe mock model of robot for paddle mount assembly.** This model depicts the top and bottom decks, clearance boxes, and the paddle mount assembly. Inner mount is green; outer mount is blue; servo is red; other important areas for clearance (i.e. H-bridges) are in transparent gray.

Figure 13: **Wireframe mock model of robot for paddle mount assembly.**

nals 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 I^2C 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.

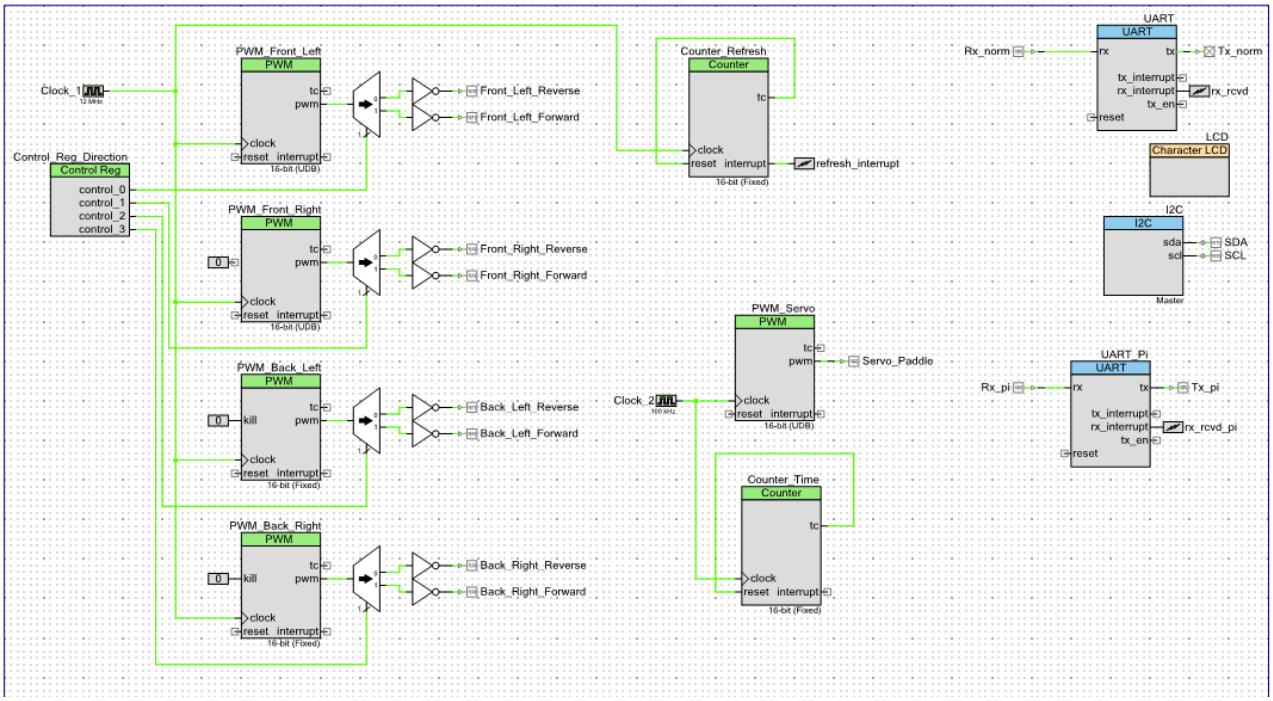


Figure 14: **Top-design of the ping pong project.**

4 Software for Raspberry Pi

4.1 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. In the terms of protocol stack models, `WiringPi` (implementing SPI/I^2C), `LinkSPI`, and `TPixy` implement sequentially higher protocol layers. 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;
}
```

As mentioned earlier, however, SPI proved not flexible enough for communication with two Pixy cameras—at least in the `WiringPi` implementation of the SPI library—so we transitioned to I^2C. This transition ended up being rather simple, because we only had to change a handful of functions to handle the difference in endianness, and to use the `WiringPiI2C.h` header instead. In order to do this, we modified `LinkSPI` to `LinkI2C`. The fully implemented communication code can be found in the appendix.

4.2 Stereo vision

Most of the code for stereo vision is direct implementation of the theory. To aid the implementation, I first defined a `Point3D` class that represents a point with x, y, z coordinates and t for time, with basic binary operations and methods such as Euclidean distance and norm:

```
struct Point3D {
    private:
        double x, y, z, t;
    public:
        /* Constructors */
        Point3D() :
            x(0), y(0), z(0), t(0) {};

        Point3D(double _x, double _y, double _z) :
            x(_x), y(_y), z(_z), t(0) {};

        Point3D(double _x, double _y, double _z, double _t) :
            x(_x), y(_y), z(_z), t(_t) {};

        /* Getters */
        double getX() { return x; }
        double getY() { return y; }
        double getZ() { return z; }
        double getT() { return t; }

        /* Operator overloads */
        // Addition and subtraction
        Point3D operator+(const Point3D& p);
        Point3D operator-(const Point3D& p);
        // Scalar multiplication and division
```

```
        Point3D operator*(const double s);
        Point3D operator/(const double s);

        /* Methods */
        double maxAbsSpatial();
        double distTo(Point3D& p);
        double norm();
};
```

The most challenging part of implementing stereo vision was identifying which pairs of identified objects (if more than one) correspond to each other. Because the image processing was implemented in the Pixy cameras' hardware, the time to process each possible pair of points (on the order of n^2 pairs) was in practice less than 1×10^{-4} s. Therefore, we would have plenty of time to examine the top approximately 10 points from each camera; in practice, the number of points detected was on the order of 3–5 points at most.

The strategy for filtering valid pixels followed several steps:

Step 1. Examine all pairs of points detected, and filter roughly round objects, according to some fractional tolerance. If it passes the roundness criterion, we add this pair as a "plausible pair" to the queue of pairs to process with the stereo code:

```
std::queue<std::pair<int, int> > plausiblePairs;
for (int ind0 = 0; ind0 < nBlocks0; ind0++) {
    for (int ind1 = 0; ind1 < nBlocks1; ind1++) {
        int w0 = pixy0.blocks[ind0].width;
        int h0 = pixy0.blocks[ind0].height;
        int w1 = pixy1.blocks[ind1].width;
        int h1 = pixy1.blocks[ind1].height;

        // Filter roughly round objects
        if (!(w0 * (1.0 - SIZE_TOL) < h0 && h0 < w0 * (1.0 + SIZE_TOL))) {
            continue;
        }
        if (!(w1 * (1.0 - SIZE_TOL) < h1 && h1 < w1 * (1.0 + SIZE_TOL))) {
            continue;
        }

        // Add plausible pair to queue (implicitly in decreasing order of
 ↪  size)
        plausiblePairs.push(std::make_pair(ind0, ind1));
    }
}
```

Step 2. For each plausible pair, calculate the position in global coordinates (in mm) with the formulas derived in **Section 2.1**. We then validate with the results from this calculation: if $z_{\text{camera}} = y_{\text{world}} < 0$, or if z_{camera} exceeds some maximum, or if the calculated width/height does not match the expected width/height of the ball.

```
/* Validate with stereo results */
if (x_diff < 0) { // x_diff makes no sense
```

```
    continue;
}
if (z_world > Z_MAX) { // max z
    continue;
}
if (!(BALL_DIAM * (1.0-SIZE_TOL) < real_w && real_w < BALL_DIAM *
 ↪  (1.0+SIZE_TOL))) {
    // wrong width
    continue;
}
if (!(BALL_DIAM * (1.0-SIZE_TOL) < real_h && real_h < BALL_DIAM *
 ↪  (1.0+SIZE_TOL))) {
    // wrong height
    continue;
}
```

Step 3. Finally, validate any remaining pairs with trajectory prediction. We use the most recent kinematic coefficients to predict the location of the ball at the current time; if the observed point in world coordinates exceeds the expected position by some threshold, we disregard the data point.

```
/* Validate with trajectory prediction */
Point3D ptExp;
// don't check trajectory if we don't have a reliable model
if (points.size() >= MIN_REGRESSION_POINTS) {
    // Max change in position from expected next position
    Point3D ptFirst = points.front();
    double t_first = ptFirst.getT();
    ptExp = predictPosition(beta_x, beta_y, beta_z, beta_R, ptFirst,
 ↪  t_elapsed);
    // Current ball position is too far from expected position
    if ((ptNow - ptExp).maxAbsSpatial() > MAX_POS_DIFF) {
        continue;
    }
}
```

If we miss too many points, we throw away the current model and start over in the hopes that we capture the ball more accurately next time.

Once a pair of points has passed these validation steps, we consider it accepted, and add it to our buffer of accepted points for regression.

4.3 Trajectory estimation

If the number of points is at least some reasonable number of points to perform regression on, we populate our observation matrices and response vectors for linear regression:

```
// Calculate bounce time
double t_bounce = predictBounceTime(beta_z, points.front().getZ());
// Regress using current coefficient of restitution
```

```
double new_beta_R = beta_R;
/* Populate predictor matrices and response vectors */
arma::mat t(points.size(), 2); // for global x, y
arma::mat t2(points.size(), 3); // for global z
arma::vec res_x(points.size()); // global x
arma::vec res_y(points.size()); // global y
arma::vec res_z(points.size()); // global z
for (int i = 0; i < points.size(); i++) {
    // Populate predictor matrices
    t(i, 0) = 1; // constant for bias/offset
    t2(i, 0) = 1;
    t(i, 1) = points[i].getT() - points.front().getT(); // t (linear term)
    t2(i, 1) = t(i, 1);
    t2(i, 2) = std::pow(t2(i, 1), 2); // t^2 (quadratic term)

    // Populate response vectors
    res_x(i) = points[i].getX();
    res_y(i) = points[i].getY();
    res_z(i) = points[i].getZ();

    // Has it bounced?
    if (t(i, 1) > t_bounce) { // If so...
        // Transform z, t to lie on the original parabola
        double tnew = t_bounce - (t(i, 1) - t_bounce) / std::sqrt(new_beta_R);
        double znew = res_z(i) / new_beta_R;
        t2(i, 1) = tnew;
        t2(i, 2) = std::pow(tnew, 2);
        res_z(i) = znew;
        res_z(i) = (beta_z(1) + (1-std::sqrt(beta_R))*beta_z(2)*t_bounce) * t(i,
 ↪  1) - res_z(i);
    }
}
```

Note that for speed, we are using the Armadillo linear algebra library (hence the `arma::mat` and `arma::vec`). This is particularly important in the following step:

```
/* Perform regression to estimate new kinematic coefficients */
arma::vec beta_x = (t.t() * t).i() * t.t() * res_x;
arma::vec beta_y = (t.t() * t).i() * t.t() * res_y;
arma::vec beta_z = (t2.t() * t2).i() * t2.t() * res_z;
```

The Armadillo library compiles consecutive linear algebra operations into a single operation in order to optimize performance. This operation is slower when done using standard `for` loops.

We then validate our results from regression: if the t^2 coefficient from the z-coordinate regression is too low (i.e. the ball appears to not be falling fast enough), we recognize that regression is not functioning correctly, so we flush the point buffer and start over. Unlike previous validation steps, however, since we have accepted this current point, we do not reject the point. Instead, we add it back to the point buffer and wait for more points to be accepted into the point buffer.

After linear regression is complete, we predict the newly updated bounce location (or the location that the robot needs to move to in order to hit the ball with the paddle) using the new kinematic coefficients. All

of this is some pretty ugly C++ implementation of the equations in **Section 2.2**, so the code is reproduced in full in the appendix. Here, we provide the two shortest methods for predicting the bounce time and position:

```
// Predict time when ball will first bounce
//   bz: 3-vector, z coefficients for naive kinematic fit
double predictBounceTime(arma::vec& bz, double z0) {
    // Solves quadratic equation
    //   z(t) = z(0) + vz(0) t + 0.5 g t^2 = 0
    double g = bz(2) * 2; // G_ACC * 1000;
    return (-1.0 / g) * (bz(1) + std::sqrt(std::pow(bz(1), 2) - 2 * g * (z0 -
 ↪  BALL_DIAM / 2)));
}

// Predict location of first bounce
//   bx, by, bz: vectors of length 2, 2, 3 respectively;
//               coefficients of naive kinematic fit
//   pt1: position of ball at time 0; point on which bx, by, bz are based
Point3D predictBounceLocation(arma::vec& bx, arma::vec& by, arma::vec& bz,
        Point3D& pt1) {
    double t_bounce = predictBounceTime(bz, pt1.getZ());
    double newx = pt1.getX() + bx(1) * t_bounce;
    double newy = pt1.getY() + by(1) * t_bounce;
    double newz = pt1.getZ() + bz(1) * t_bounce
        + bz(2) * std::pow(t_bounce, 2);

    return Point3D(newx, newy, newz, t_bounce);
}
```

5 Software for PSoC

5.1 Motor control

The low-level motor control is handled by the function `drive()`. It simply takes a magnitude and direction for each motor and interfaces with the programmable hardware to do it.

```
static void drive(int dirs[], double mags[]) {
    int i;
    for (i = 0; i < 4; i++) {
        if (mags[i] < 0) mags[i] = 0;
        if (mags[i] > 1) mags[i] = 1;
        if (dirs[i] != 0 && dirs[i] != 1) dirs[i] = 0;
    }

    PWM_Front_Left_WriteCompare(mags[0] * PWM_Front_Left_ReadPeriod());
    PWM_Front_Right_WriteCompare(mags[1] * PWM_Front_Right_ReadPeriod());
    PWM_Back_Left_WriteCompare(mags[3] * PWM_Back_Left_ReadPeriod());
    PWM_Back_Right_WriteCompare(mags[2] * PWM_Back_Right_ReadPeriod());
```

```
    int regVal = dirs[0] | (dirs[1] << 1) | (dirs[3] << 2) | (dirs[2] << 3);
    Control_Reg_Direction_Write(regVal);
}
```

The high level drive control is in the function `driveMagPhase()`, and implements the equations discussed in **Section 2.3**. `driveMagPhase()` implements the equations in terms of the dominant (i.e. greatest absolute magnitude) motor. The dominant partner is then the motor on the same axis as the dominant motor. The off dominant motor is the motor of greatest magnitude on the other axis, and the off dominant partner is the remaining motor. The first half of the function uses some bit masking to determine which motor is the dominant one, first by figuring out which axis is dominant, and then figuring out which motor on that axis has the greater magnitude. Once these designations have been found, the magnitudes and directions of each motor can be calculated basically independently. The magnitude of the dominant motor is $\max(1, M + |x|)$, the magnitude of the dominant partner is $|M_{\text{dominant}} - 2|s||$, the magnitude of the off dominant motor is $(M_{\text{dominant}} - |s|)a + |s|$, where

$$a = \min\left(\left|\tan\left(\alpha - \theta + \frac{\pi}{4}\right)\right|, \frac{1}{\left|\tan\left(\alpha - \theta + \frac{\pi}{4}\right)\right|}\right),$$

and the magnitude of the off dominant partner is $(M_{\text{dominant}} - |s|)a - |s|$.

Then the function uses some of the bit masks from earlier to find the final directions of each motor, and sends the completed direction-magnitude pairs to `drive()`.

```
// direction and current angle are CCW w.r.t global forward (in rads),
// magnitude is from 0 to 1, angleMag is CCW
static void driveMagPhase(double direction, double magnitude, double angleMag,
 ↪  double currentAngle) {
    // angle of drive CCW w.r.t the axis formed by the front right and back left
 ↪  wheels
    double angle = fmod(direction - currentAngle + (M_PI/4), 2*M_PI);
    angle = angle < 0 ? angle + 2*M_PI : angle;

    if (angleMag > 1) angleMag = 1;
    if (angleMag < -1) angleMag = -1;
    if (magnitude > 1) magnitude = 1;
    if (magnitude < -1) magnitude = -1;

    // 1 is dominant, 0 is non-dominant, motor 4 is lsb
    uint8 dirDominant = 0;
    if (fabs(tan(angle)) > 1) dirDominant = 0b0101;
    else dirDominant = 0b1010;

    // 0 is backward, 1 is forward
    uint8 directionChangeDir = 0;
    if ((angle > 0 && angle < M_PI/2) || (angle > 3*M_PI/2 && angle < 2*M_PI))
 ↪  directionChangeDir |= 0b1010;
    else directionChangeDir &= 0b0101;
    if (angle > 0 && angle < M_PI) directionChangeDir |= 0b0101;
    else directionChangeDir &= 0b1010;
```

```
    // 0 is backward, 1 is forward
    uint8 angleChangeDir = 0;
    if (angleMag > 0) angleChangeDir = 0b0110;
    else angleChangeDir = 0b1001;

    // 1 is dominant, 0 is non-dominant
    uint8 angleDominant = ~(directionChangeDir ^ angleChangeDir);
    uint8 dominant = dirDominant & angleDominant;
    int dominantNum = (dominant & 8) ? 0 : (dominant & 4) ? 1 : (dominant & 2) ? 2
↪   : 3;
    uint8 offDominant = ~dirDominant & angleDominant;
    int offDominantNum = (offDominant & 8) ? 0 : (offDominant & 4) ? 1 :
↪   (offDominant & 2) ? 2 : 3;

    // Calculate mags
    double domMag = (magnitude + fabs(angleMag));
    if (domMag > 1) {
        angleMag /= domMag;
        domMag = 1;
    }
    double domPartnerMag = domMag - 2*fabs(angleMag);
    double offMultiplier = fabs(tan(angle));
    offMultiplier = offMultiplier > 1 ? (1/offMultiplier) : offMultiplier;
   double offDomMag = (domMag - fabs(angleMag)) * offMultiplier + fabs(angleMag);
    double offDomPartnerMag = (domMag - fabs(angleMag)) * offMultiplier -
↪   fabs(angleMag);

    // Assign mags
    double mags[4];
    mags[dominantNum] = domMag;
    mags[(dominantNum + 2) % 4] = fabs(domPartnerMag);
    mags[offDominantNum] = offDomMag;
    mags[(offDominantNum + 2) % 4] = fabs(offDomPartnerMag);

    // Find dirs
    int dirs[4];
    dirs[dominantNum] = ((dominant & directionChangeDir) != 0) ? 1 : 0;
    dirs[offDominantNum] = ((offDominant & directionChangeDir) != 0) ? 1 : 0;
    dirs[(dominantNum + 2) % 4] = domPartnerMag > 0 ? dirs[dominantNum] : 1 -
↪   dirs[dominantNum];
    dirs[(offDominantNum + 2) % 4] = offDomPartnerMag > 0 ? dirs[offDominantNum]
↪   : 1 - dirs[offDominantNum];

    drive(dirs, mags);
}
```

5.2 Dead reckoning

Reading the accelerometer and gyroscope is handled by the `accel.c` file, which is loosely ported from an Adafruit library. The `Accel_init()` function sets up the accelerometer and gyroscope. It sets the sample rate of the accelerometer, enables x, y, and z directions, and sets its range to $\pm 2g$. It also enables the gyro channels, sets up the limits for the built-in low- and high-pass filters, and sets its range to $\pm 245\,^\circ\,\mathrm{s}^{-1}$. The high-pass filter cutoff is 7.2 Hz, which kills any drift in the gyroscope readings. There are many private helper functions that handle the details of the I^2C interface and getting the raw data from the accelerometer and gyroscope.

The data from the accelerometer, though, is very noisy, so we do a fair amount of noise reduction in the function `Accel_refresh()`. This function is called periodically by `main.c`, and is the main function used to update the sensor readings. The first part of the function is an initialization step. The function takes the first `NUM_ZERO_READINGS`, 1000, accelerometer readings and uses them to create denoising functions. First, we simply take the (3 dimensional) mean of all the readings and subtract that from all future readings. Unlike the gyro, the accelerometer does not have a built in high-pass filter, so we have to do this subtraction manually.

Next, we noticed that the noise in the x and y directions almost perfects anti-correlates with the noise in the z direction as shown in **Figure 15**. Since we know that the z acceleration should always be zero, we can use the z acceleration reading to remove some of the noise from the x and y readings. Since our data after having the mean subtracted is zero centered, the correlations can simply be calculated by

$$C_{xz} = \frac{x \cdot z}{\|z\|^2}$$

and

$$C_{yz} = \frac{y \cdot z}{\|z\|^2},$$

where x, y, and z are vectors of the readings. The readings can then be decorrelated as follows: $x_{\text{decorr}} = x - zC_{xz}$ and $y_{\text{decorr}} = y - zC_{yz}$. **Figure 15** shows the result of this decorrelation. The decorrelation reduces the noise of the sensor by about a factor of two. All of the preceding calculations had to be performed using integer arithmetic, since double arrays were too large to fit on the PSoC's limited memory.

Once the means and correlation coefficients have been calculated over the first `NUM_ZERO_READINGS` readings, `Accel_refresh()` simply gets new raw readings and corrects them according to the calculated means and correlation coefficients. The public functions `Accel_getAccel()` and `Accel_getGyro()` are then used to get the corrected readings.

```
void Accel_refresh() {
    // Gather values for zeroing routine
    if (zeroed != 1) {
        readAccel(initAccelValsBuf[initBufPos]);
        readGyro(initGyroValsBuf[initBufPos]);
        initBufPos++;

        if (initBufPos >= NUM_ZERO_READINGS) {
            int i, j;
            // Calculate mean of init data
            for (i = 0; i < 3; i++) {
                for (j = 0; j < NUM_ZERO_READINGS; j++) {
                    accelOffset[i] += initAccelValsBuf[j][i];
                    gyroOffset[i] += initGyroValsBuf[j][i];
```

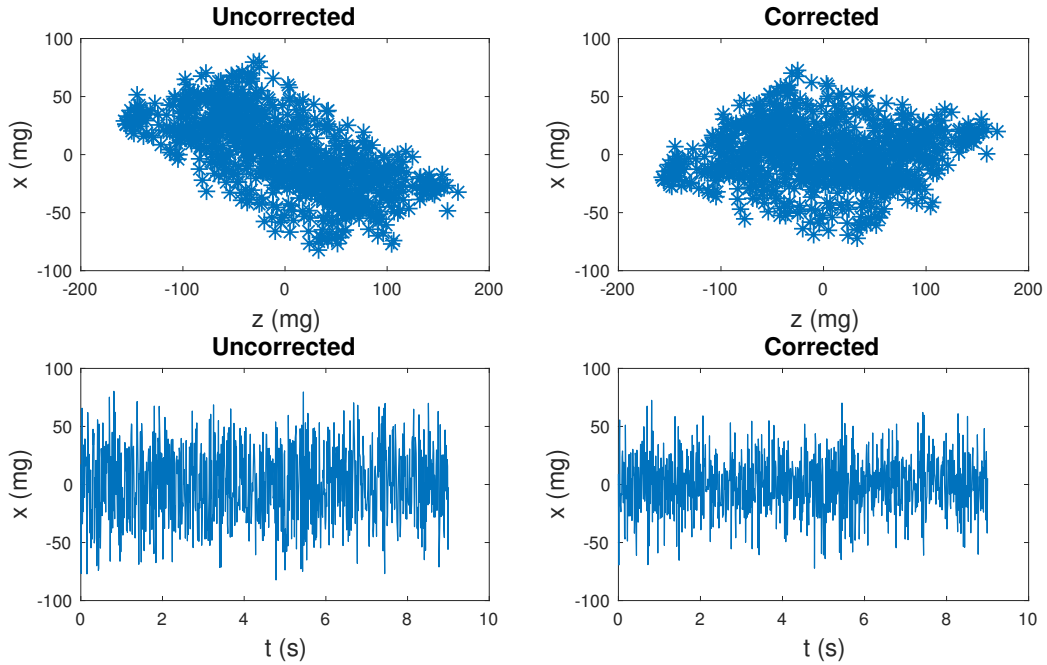


Figure 15: **Plots of accelerometer noise.** From left to right and top to bottom, uncorrected correlation between x and z (notice the negative correlation), corrected correlation between x and z, uncorrected x noise against time (in seconds), and corrected x noise against time. In all these measurements, the robot was not moving, so all fluctuations were noise or minute vibrations in the ELE teaching lab. All accelerometer measurements are in milli-gs (1 milli-g is 9.81 mm s^{-2}).

```
            }
            accelOffset[i] /= NUM_ZERO_READINGS;
            gyroOffset[i] /= NUM_ZERO_READINGS;
        }

         // Calculate correlation coefficients
        int zxAccelInner = 0;
        int zyAccelInner = 0;
        int zAccelNormSquare = 0;
        for (i = 0; i < NUM_ZERO_READINGS; i++) {
            zxAccelInner += (initAccelValsBuf[i][2] - accelOffset[2]) *
                            (initAccelValsBuf[i][0] - accelOffset[0]) /
↪   NUM_ZERO_READINGS;
            zyAccelInner += (initAccelValsBuf[i][2] - accelOffset[2]) *
                            (initAccelValsBuf[i][1] - accelOffset[1]) /
↪   NUM_ZERO_READINGS;
            zAccelNormSquare += (initAccelValsBuf[i][2] - accelOffset[2]) *
                                (initAccelValsBuf[i][2] - accelOffset[2]) /
↪   NUM_ZERO_READINGS;
        }
```

```
            double zxCorrelationAccel = (double) zxAccelInner / (double)
↪   zAccelNormSquare;
            double zyCorrelationAccel = (double) zyAccelInner / (double)
↪   zAccelNormSquare;

            zeroed = 1;
        }
    }
    else if (zeroed == 1) {
        int i;

        // Subtract oldest vals from average
        for (i = 0; i < 3; i++) {
            accelValAvg[i] -= accelValsBuf[bufPos][i] / AVG_BUF_LEN;
            gyroValAvg[i] -= gyroValsBuf[bufPos][i] / AVG_BUF_LEN;
        }

        // Get new vals
        readAccel(accelValsBuf[bufPos]);
        readGyro(gyroValsBuf[bufPos]);

        // Subtract means
        for (i = 0; i < 3; i++) {
            accelValsBuf[bufPos][i] -= accelOffset[i];
            gyroValsBuf[bufPos][i] -= gyroOffset[i];
        }

        // Decorrelate
        accelValsBuf[bufPos][0] -= accelValsBuf[bufPos][2] * zxCorrelationAccel;
        accelValsBuf[bufPos][1] -= accelValsBuf[bufPos][2] * zyCorrelationAccel;

        // Add new vals to average
        for (i = 0; i < 3; i++) {
            accelValAvg[i] += accelValsBuf[bufPos][i] / AVG_BUF_LEN;
            gyroValAvg[i] += gyroValsBuf[bufPos][i] / AVG_BUF_LEN;
        }

        // Increment buffer counter
        bufPos = (bufPos + 1) % AVG_BUF_LEN;
    }
}
```

`main.c` uses the `Counter_Refresh` counter to call `Accel_refresh()`, and to translate the raw accelerometer data into usable data as described in **Section 2.4**. The gyroscope returns angular velocity data, so it also runs a single integrator on the angular velocity to get angular position. The integrators are discrete, and since the integration is done at a regular interval, they only need to keep a running sum, multiplied by the time between sums.

```
// Integrate angular velocity to get angle
currentAngle += gyroVals[2] * ACCEL_READ_PERIOD * (M_PI/180);

// Convert acceleration to mm/s^2
accelVals[0] *= 9.81;
accelVals[1] *= 9.81;

// Correct raw accel axis for centrifugal effects
double omega = gyroVals[2] * ACCEL_READ_PERIOD;
accelVals[0] += rAccelCorrect*cos(alphaAccelCorrect)*omega*omega;
accelVals[1] += rAccelCorrect*sin(alphaAccelCorrect)*omega*omega;

// Convert to true x and y
double trueAccel[2];
trueAccel[0] = -accelVals[0]*sin(currentAngle) - accelVals[1]*cos(currentAngle);
trueAccel[1] = accelVals[0]*cos(currentAngle) - accelVals[1]*sin(currentAngle);

// Integrate acceleration to get velocity and position
int i;
for (i = 0; i < 2; i++) {
    currentVelocity[i] += trueAccel[i] * ACCEL_READ_PERIOD;
    currentPos[i] += currentVelocity[i] * ACCEL_READ_PERIOD;
}
```

5.3 Fine tuning control

During testing, we noticed that if the car went to full speed immediately, it would slip a lot before it got traction. We decided the best way to limit this effect would be to put a cap on the maximum allowed acceleration. Since the motor speeds need to be updated very frequently (all the equations described above and in **Section 2.3** are only valid instantaneously), we simply implemented the acceleration cap in the same spot as we updated the motor speeds. We run through the update loop 100 times per second, so we simply keep track of the directional magnitude (M, in all the equations) at the last loop, and do not let the new directional magnitude exceed the old one by more than some fixed amount A. We settled on using $A = 0.03$, which allows the car to reach full speed in 0.33 s.

```
if (accelerationLimitEnabled) {
    static double oldMag;
    double setMag = (fabs(magnitude-oldMag) > accelLimit) ? (magnitude -
 ↪  oldMag)/fabs(magnitude-oldMag) * accelLimit + oldMag : magnitude;
    driveMagPhase(heading, setMag, rotation, enableGyro == 1 ? currentAngle : 0);
    oldMag = setMag;
}
else {
    driveMagPhase(heading, magnitude, rotation, enableGyro == 1 ? currentAngle :
 ↪  0);
}
```

The final major part of this system is to command the robot to drive to a given location in global coordinates. With an omni-drive, this is actually very easy. It simply calculates the angle between the car's

current position and the desired position, and then drives directly in that direction with a magnitude scaled by the distance left. It updates the required position and magnitude 100 times per second, so it continuously corrects for any errors in the drive. Once the car reaches the target location within a set margin, 5 cm, for our final tests, it stops. Since our goal was to be able to hit a ping ping ball, this code also contains a small "flick" routine. Given a time at which to flick, the robot will start rotating just before the target time, and stop rotating at the target time, creating a small flicking motion designed to hit the ball. We also added a safety timeout to this code to automatically stop if the target had not been reached in 2 seconds, to stop runaway behavior.

```
if (goToTarget) {
    int newCounterVal = Counter_Time_ReadCounter();
    int countsElapsed = oldCounterVal - newCounterVal;
    if (countsElapsed < 0) countsElapsed += 65535;
    timeElapsed += (double) countsElapsed / 100000.0;
    oldCounterVal = newCounterVal;

    double xDist = (target[0] - currentPos[0]);
    double yDist = (target[1] - currentPos[1]);
    if (yDist == 0) yDist = 0.0000001;
    heading = atan(-xDist/yDist);
    if (yDist < 0) heading += M_PI;
    double dist = sqrt(xDist*xDist + yDist*yDist);
    magnitude = P*dist;
    if (magnitude > 1) magnitude = 1;

    //targetAngle = M_PI/2;

    rotation = 0;
    enableGyro = 1;

    if (dist < targetMargin) {
        goToTarget = 0;
        magnitude = 0;
        rotation = 0;
        sprintf(strbuffer, "Target Reached in %fs\n", timeElapsed);
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
    }

    if (timeElapsed > 1) {
        goToTarget = 0;
        magnitude = 0;
        rotation = 0;
        sprintf(strbuffer, "Time Up\n");
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
    }

    if (timeElapsed > targetTime - flickTime && !rotOn && timeElapsed <
↪   targetTime) {
```

```
            rotation = -1;
            sprintf(strbuffer, "Rot on\n");
            terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
            rotOn = 1;
        }
        if (timeElapsed > targetTime && rotOn) {
            rotation = 0;
            sprintf(strbuffer, "Rot off\n");
            terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
            rotOn = 0;
        }
    }
```

5.4 Serial communication

We used serial communication both for debugging and for communicating between the PSoC and the Raspberry Pi. **Table 1** shows all the serial commands used.

Command	Description
`CTx.y`	Set motor `x` to throttle `y`
`CPx`	Set motor PWM period to `x`
`w`	Drive forward
`s`	Drive backward
`a`	Drive left
`d`	Drive right
`q or Q`	Stop
`c`	Rotate right
`z`	Rotate left
`x`	Stop rotating
`CMx`	Change magnitude of driving to `x`
`CHx`	Change heading of driving to `x` (in radians)
`CRx`	Set rotation speed to `x`
`R`	Reset inertial tracking
`P`	Print inertial tracking variables
`EG`	Enable gyro
`DG`	Disable gyro
`EAL`	Enable acceleration limit
`DAL`	Disable acceleration limit
`SALx`	Set acceleration limit to `x`
`GO(x, y, t, θ)`	Go to position (x, y) within time t and set paddle angle to θ
`CTMx`	Change stop margin for arriving at target to `x`
`SFT`	Set time of rotation need for flicking to `x`

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

6 Further work

In the end, we were able to somewhat accurately predict the location of a ping pong ball when thrown to the robot, but both the trajectory estimation and movement control were too slow (the ball had moved too far before we knew where to move, and we couldn't move there in time) to consistently hit the ball. With a larger red ball (such as a dodgeball), we were able to more consistently detect and move to the landing position, but there was a lot of room for improvement.

If we could work on this project further, we would have liked:

- Clearer and more precise object detection, perhaps using higher resolution cameras and performing our own image processing;
- More robust ball tracking, perhaps using better trajectory prediction and validation methods;
- More robust location tracking, perhaps using a different algorithm (or updating zero values of the accelerometer whenever we were stopped), or finding a better accelerometer, or using a line sensor for continual retuning, or abandoning dead reckoning altogether and adopting another form of position-finding;
- More robust UART communication, since we encountered difficulties with received buffer overflows and symbol mismatches that caused characters to be missed in sending commands through serial interfaces;
- Some form of PID control with a difference between global coordinates and camera coordinates, so that the landing position and target location of the robot could be updated in real time and so that the robot could move to the target location in time;
- And various other niceties, such as figuring out a better way to accelerate without slipping, further denoising of the accelerometer, and correcting the gyroscope's drift on fast turns.

All in all, we thoroughly enjoyed this experience, and we would like to thank Professors Houck and Thompson for their dedication and willingness to help at all hours of the day, and all the graduate and undergraduate TAs for the same. This would not have been possible without you.

We pledge our honor that this project report represents our own work in accordance with University regulations.
Byung-Cheol Cho
TJ Smith

7 Appendix: full listings

Contents

7.1 Raspberry Pi: `pingpong.cpp`

```
if (goToTarget) {
    int newCounterVal = Counter_Time_ReadCounter();
    int countsElapsed = oldCounterVal - newCounterVal;
    if (countsElapsed < 0) countsElapsed += 65535;
    timeElapsed += (double) countsElapsed / 100000.0;
    oldCounterVal = newCounterVal;

    double xDist = (target[0] - currentPos[0]);
    double yDist = (target[1] - currentPos[1]);
    if (yDist == 0) yDist = 0.0000001;
    heading = atan(-xDist/yDist);
    if (yDist < 0) heading += M_PI;
    double dist = sqrt(xDist*xDist + yDist*yDist);
    magnitude = P*dist;
    if (magnitude > 1) magnitude = 1;

    //targetAngle = M_PI/2;

    rotation = 0;
    enableGyro = 1;

    if (dist < targetMargin) {
        goToTarget = 0;
        magnitude = 0;
        rotation = 0;
        sprintf(strbuffer, "Target Reached in %fs\n", timeElapsed);
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
    }
```

```
    if (timeElapsed > 1) {
        goToTarget = 0;
        magnitude = 0;
        rotation = 0;
        sprintf(strbuffer, "Time Up\n");
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
    }

    if (timeElapsed > targetTime - flickTime && !rotOn && timeElapsed <
 ↪  targetTime) {
        rotation = -1;
        sprintf(strbuffer, "Rot on\n");
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
        rotOn = 1;
    }
    if (timeElapsed > targetTime && rotOn) {
        rotation = 0;
        sprintf(strbuffer, "Rot off\n");
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
        rotOn = 0;
    }
}
```

7.2 Raspberry Pi: `Pixy.h`

```
if (goToTarget) {
    int newCounterVal = Counter_Time_ReadCounter();
    int countsElapsed = oldCounterVal - newCounterVal;
    if (countsElapsed < 0) countsElapsed += 65535;
    timeElapsed += (double) countsElapsed / 100000.0;
    oldCounterVal = newCounterVal;

    double xDist = (target[0] - currentPos[0]);
    double yDist = (target[1] - currentPos[1]);
    if (yDist == 0) yDist = 0.0000001;
    heading = atan(-xDist/yDist);
    if (yDist < 0) heading += M_PI;
    double dist = sqrt(xDist*xDist + yDist*yDist);
    magnitude = P*dist;
    if (magnitude > 1) magnitude = 1;

    //targetAngle = M_PI/2;

    rotation = 0;
    enableGyro = 1;

    if (dist < targetMargin) {
```

```
        goToTarget = 0;
        magnitude = 0;
        rotation = 0;
        sprintf(strbuffer, "Target Reached in %fs\n", timeElapsed);
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
    }

    if (timeElapsed > 1) {
        goToTarget = 0;
        magnitude = 0;
        rotation = 0;
        sprintf(strbuffer, "Time Up\n");
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
    }

    if (timeElapsed > targetTime - flickTime && !rotOn && timeElapsed <
 ↪  targetTime) {
        rotation = -1;
        sprintf(strbuffer, "Rot on\n");
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
        rotOn = 1;
    }
    if (timeElapsed > targetTime && rotOn) {
        rotation = 0;
        sprintf(strbuffer, "Rot off\n");
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
        rotOn = 0;
    }
}
```

7.3 Raspberry Pi: `TPixy.h`

```
if (goToTarget) {
    int newCounterVal = Counter_Time_ReadCounter();
    int countsElapsed = oldCounterVal - newCounterVal;
    if (countsElapsed < 0) countsElapsed += 65535;
    timeElapsed += (double) countsElapsed / 100000.0;
    oldCounterVal = newCounterVal;

    double xDist = (target[0] - currentPos[0]);
    double yDist = (target[1] - currentPos[1]);
    if (yDist == 0) yDist = 0.0000001;
    heading = atan(-xDist/yDist);
    if (yDist < 0) heading += M_PI;
    double dist = sqrt(xDist*xDist + yDist*yDist);
    magnitude = P*dist;
    if (magnitude > 1) magnitude = 1;
```

```
    //targetAngle = M_PI/2;

    rotation = 0;
    enableGyro = 1;

    if (dist < targetMargin) {
        goToTarget = 0;
        magnitude = 0;
        rotation = 0;
        sprintf(strbuffer, "Target Reached in %fs\n", timeElapsed);
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
    }

    if (timeElapsed > 1) {
        goToTarget = 0;
        magnitude = 0;
        rotation = 0;
        sprintf(strbuffer, "Time Up\n");
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
    }

    if (timeElapsed > targetTime - flickTime && !rotOn && timeElapsed <
↪   targetTime) {
        rotation = -1;
        sprintf(strbuffer, "Rot on\n");
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
        rotOn = 1;
    }
    if (timeElapsed > targetTime && rotOn) {
        rotation = 0;
        sprintf(strbuffer, "Rot off\n");
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
        rotOn = 0;
    }
}
```

7.4 Raspberry Pi: `timer.h`

```
if (goToTarget) {
    int newCounterVal = Counter_Time_ReadCounter();
    int countsElapsed = oldCounterVal - newCounterVal;
    if (countsElapsed < 0) countsElapsed += 65535;
    timeElapsed += (double) countsElapsed / 100000.0;
    oldCounterVal = newCounterVal;

    double xDist = (target[0] - currentPos[0]);
```

```
    double yDist = (target[1] - currentPos[1]);
    if (yDist == 0) yDist = 0.0000001;
    heading = atan(-xDist/yDist);
    if (yDist < 0) heading += M_PI;
    double dist = sqrt(xDist*xDist + yDist*yDist);
    magnitude = P*dist;
    if (magnitude > 1) magnitude = 1;

    //targetAngle = M_PI/2;

    rotation = 0;
    enableGyro = 1;

    if (dist < targetMargin) {
        goToTarget = 0;
        magnitude = 0;
        rotation = 0;
        sprintf(strbuffer, "Target Reached in %fs\n", timeElapsed);
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
    }

    if (timeElapsed > 1) {
        goToTarget = 0;
        magnitude = 0;
        rotation = 0;
        sprintf(strbuffer, "Time Up\n");
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
    }

    if (timeElapsed > targetTime - flickTime && !rotOn && timeElapsed <
↪   targetTime) {
        rotation = -1;
        sprintf(strbuffer, "Rot on\n");
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
        rotOn = 1;
    }
    if (timeElapsed > targetTime && rotOn) {
        rotation = 0;
        sprintf(strbuffer, "Rot off\n");
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
        rotOn = 0;
    }
}
```

7.5 PSoC: `main.c`

```
if (goToTarget) {
```

```
    int newCounterVal = Counter_Time_ReadCounter();
    int countsElapsed = oldCounterVal - newCounterVal;
    if (countsElapsed < 0) countsElapsed += 65535;
    timeElapsed += (double) countsElapsed / 100000.0;
    oldCounterVal = newCounterVal;

    double xDist = (target[0] - currentPos[0]);
    double yDist = (target[1] - currentPos[1]);
    if (yDist == 0) yDist = 0.0000001;
    heading = atan(-xDist/yDist);
    if (yDist < 0) heading += M_PI;
    double dist = sqrt(xDist*xDist + yDist*yDist);
    magnitude = P*dist;
    if (magnitude > 1) magnitude = 1;

    //targetAngle = M_PI/2;

    rotation = 0;
    enableGyro = 1;

    if (dist < targetMargin) {
        goToTarget = 0;
        magnitude = 0;
        rotation = 0;
        sprintf(strbuffer, "Target Reached in %fs\n", timeElapsed);
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
    }

    if (timeElapsed > 1) {
        goToTarget = 0;
        magnitude = 0;
        rotation = 0;
        sprintf(strbuffer, "Time Up\n");
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
    }

    if (timeElapsed > targetTime - flickTime && !rotOn && timeElapsed <
↪   targetTime) {
        rotation = -1;
        sprintf(strbuffer, "Rot on\n");
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
        rotOn = 1;
    }
    if (timeElapsed > targetTime && rotOn) {
        rotation = 0;
        sprintf(strbuffer, "Rot off\n");
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
        rotOn = 0;
```

```
    }
}
```

7.6 PSoC: `hardware.h`

```
if (goToTarget) {
    int newCounterVal = Counter_Time_ReadCounter();
    int countsElapsed = oldCounterVal - newCounterVal;
    if (countsElapsed < 0) countsElapsed += 65535;
    timeElapsed += (double) countsElapsed / 100000.0;
    oldCounterVal = newCounterVal;

    double xDist = (target[0] - currentPos[0]);
    double yDist = (target[1] - currentPos[1]);
    if (yDist == 0) yDist = 0.0000001;
    heading = atan(-xDist/yDist);
    if (yDist < 0) heading += M_PI;
    double dist = sqrt(xDist*xDist + yDist*yDist);
    magnitude = P*dist;
    if (magnitude > 1) magnitude = 1;

    //targetAngle = M_PI/2;

    rotation = 0;
    enableGyro = 1;

    if (dist < targetMargin) {
        goToTarget = 0;
        magnitude = 0;
        rotation = 0;
        sprintf(strbuffer, "Target Reached in %fs\n", timeElapsed);
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
    }

    if (timeElapsed > 1) {
        goToTarget = 0;
        magnitude = 0;
        rotation = 0;
        sprintf(strbuffer, "Time Up\n");
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
    }

    if (timeElapsed > targetTime - flickTime && !rotOn && timeElapsed <
 ↪  targetTime) {
        rotation = -1;
        sprintf(strbuffer, "Rot on\n");
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
```

```
        rotOn = 1;
    }
    if (timeElapsed > targetTime && rotOn) {
        rotation = 0;
        sprintf(strbuffer, "Rot off\n");
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
        rotOn = 0;
    }
}
```

7.7 PSoC: `accel.h`

```
if (goToTarget) {
    int newCounterVal = Counter_Time_ReadCounter();
    int countsElapsed = oldCounterVal - newCounterVal;
    if (countsElapsed < 0) countsElapsed += 65535;
    timeElapsed += (double) countsElapsed / 100000.0;
    oldCounterVal = newCounterVal;

    double xDist = (target[0] - currentPos[0]);
    double yDist = (target[1] - currentPos[1]);
    if (yDist == 0) yDist = 0.0000001;
    heading = atan(-xDist/yDist);
    if (yDist < 0) heading += M_PI;
    double dist = sqrt(xDist*xDist + yDist*yDist);
    magnitude = P*dist;
    if (magnitude > 1) magnitude = 1;

    //targetAngle = M_PI/2;

    rotation = 0;
    enableGyro = 1;

    if (dist < targetMargin) {
        goToTarget = 0;
        magnitude = 0;
        rotation = 0;
        sprintf(strbuffer, "Target Reached in %fs\n", timeElapsed);
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
    }

    if (timeElapsed > 1) {
        goToTarget = 0;
        magnitude = 0;
        rotation = 0;
        sprintf(strbuffer, "Time Up\n");
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
```

```
    }

    if (timeElapsed > targetTime - flickTime && !rotOn && timeElapsed <
 ↪  targetTime) {
        rotation = -1;
        sprintf(strbuffer, "Rot on\n");
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
        rotOn = 1;
    }
    if (timeElapsed > targetTime && rotOn) {
        rotation = 0;
        sprintf(strbuffer, "Rot off\n");
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
        rotOn = 0;
    }
}
```

7.8 PSoC: `accel.c`

```
if (goToTarget) {
    int newCounterVal = Counter_Time_ReadCounter();
    int countsElapsed = oldCounterVal - newCounterVal;
    if (countsElapsed < 0) countsElapsed += 65535;
    timeElapsed += (double) countsElapsed / 100000.0;
    oldCounterVal = newCounterVal;

    double xDist = (target[0] - currentPos[0]);
    double yDist = (target[1] - currentPos[1]);
    if (yDist == 0) yDist = 0.0000001;
    heading = atan(-xDist/yDist);
    if (yDist < 0) heading += M_PI;
    double dist = sqrt(xDist*xDist + yDist*yDist);
    magnitude = P*dist;
    if (magnitude > 1) magnitude = 1;

    //targetAngle = M_PI/2;

    rotation = 0;
    enableGyro = 1;

    if (dist < targetMargin) {
        goToTarget = 0;
        magnitude = 0;
        rotation = 0;
        sprintf(strbuffer, "Target Reached in %fs\n", timeElapsed);
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
    }
```

```
    if (timeElapsed > 1) {
        goToTarget = 0;
        magnitude = 0;
        rotation = 0;
        sprintf(strbuffer, "Time Up\n");
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
    }

    if (timeElapsed > targetTime - flickTime && !rotOn && timeElapsed <
 ↪  targetTime) {
        rotation = -1;
        sprintf(strbuffer, "Rot on\n");
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
        rotOn = 1;
    }
    if (timeElapsed > targetTime && rotOn) {
        rotation = 0;
        sprintf(strbuffer, "Rot off\n");
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
        rotOn = 0;
    }
}
```

7.9 PSoC: `serial.h`

```
if (goToTarget) {
    int newCounterVal = Counter_Time_ReadCounter();
    int countsElapsed = oldCounterVal - newCounterVal;
    if (countsElapsed < 0) countsElapsed += 65535;
    timeElapsed += (double) countsElapsed / 100000.0;
    oldCounterVal = newCounterVal;

    double xDist = (target[0] - currentPos[0]);
    double yDist = (target[1] - currentPos[1]);
    if (yDist == 0) yDist = 0.0000001;
    heading = atan(-xDist/yDist);
    if (yDist < 0) heading += M_PI;
    double dist = sqrt(xDist*xDist + yDist*yDist);
    magnitude = P*dist;
    if (magnitude > 1) magnitude = 1;

    //targetAngle = M_PI/2;

    rotation = 0;
    enableGyro = 1;
```

```
    if (dist < targetMargin) {
        goToTarget = 0;
        magnitude = 0;
        rotation = 0;
        sprintf(strbuffer, "Target Reached in %fs\n", timeElapsed);
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
    }

    if (timeElapsed > 1) {
        goToTarget = 0;
        magnitude = 0;
        rotation = 0;
        sprintf(strbuffer, "Time Up\n");
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
    }

    if (timeElapsed > targetTime - flickTime && !rotOn && timeElapsed <
↪   targetTime) {
        rotation = -1;
        sprintf(strbuffer, "Rot on\n");
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
        rotOn = 1;
    }
    if (timeElapsed > targetTime && rotOn) {
        rotation = 0;
        sprintf(strbuffer, "Rot off\n");
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
        rotOn = 0;
    }
}
```

7.10 PSoC: `serial.c`

```
if (goToTarget) {
    int newCounterVal = Counter_Time_ReadCounter();
    int countsElapsed = oldCounterVal - newCounterVal;
    if (countsElapsed < 0) countsElapsed += 65535;
    timeElapsed += (double) countsElapsed / 100000.0;
    oldCounterVal = newCounterVal;

    double xDist = (target[0] - currentPos[0]);
    double yDist = (target[1] - currentPos[1]);
    if (yDist == 0) yDist = 0.0000001;
    heading = atan(-xDist/yDist);
    if (yDist < 0) heading += M_PI;
    double dist = sqrt(xDist*xDist + yDist*yDist);
    magnitude = P*dist;
```

```
    if (magnitude > 1) magnitude = 1;

    //targetAngle = M_PI/2;

    rotation = 0;
    enableGyro = 1;

    if (dist < targetMargin) {
        goToTarget = 0;
        magnitude = 0;
        rotation = 0;
        sprintf(strbuffer, "Target Reached in %fs\n", timeElapsed);
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
    }

    if (timeElapsed > 1) {
        goToTarget = 0;
        magnitude = 0;
        rotation = 0;
        sprintf(strbuffer, "Time Up\n");
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
    }

    if (timeElapsed > targetTime - flickTime && !rotOn && timeElapsed <
↪   targetTime) {
        rotation = -1;
        sprintf(strbuffer, "Rot on\n");
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
        rotOn = 1;
    }
    if (timeElapsed > targetTime && rotOn) {
        rotation = 0;
        sprintf(strbuffer, "Rot off\n");
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
        rotOn = 0;
    }
}
```

7.11 PSoC: `serial_pi.h`

```
if (goToTarget) {
    int newCounterVal = Counter_Time_ReadCounter();
    int countsElapsed = oldCounterVal - newCounterVal;
    if (countsElapsed < 0) countsElapsed += 65535;
    timeElapsed += (double) countsElapsed / 100000.0;
    oldCounterVal = newCounterVal;
```

```
    double xDist = (target[0] - currentPos[0]);
    double yDist = (target[1] - currentPos[1]);
    if (yDist == 0) yDist = 0.0000001;
    heading = atan(-xDist/yDist);
    if (yDist < 0) heading += M_PI;
    double dist = sqrt(xDist*xDist + yDist*yDist);
    magnitude = P*dist;
    if (magnitude > 1) magnitude = 1;

    //targetAngle = M_PI/2;

    rotation = 0;
    enableGyro = 1;

    if (dist < targetMargin) {
        goToTarget = 0;
        magnitude = 0;
        rotation = 0;
        sprintf(strbuffer, "Target Reached in %fs\n", timeElapsed);
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
    }

    if (timeElapsed > 1) {
        goToTarget = 0;
        magnitude = 0;
        rotation = 0;
        sprintf(strbuffer, "Time Up\n");
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
    }

    if (timeElapsed > targetTime - flickTime && !rotOn && timeElapsed <
↪   targetTime) {
        rotation = -1;
        sprintf(strbuffer, "Rot on\n");
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
        rotOn = 1;
    }
    if (timeElapsed > targetTime && rotOn) {
        rotation = 0;
        sprintf(strbuffer, "Rot off\n");
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
        rotOn = 0;
    }
}
```

7.12 PSoC: `serial_pi.c`

```
if (goToTarget) {
    int newCounterVal = Counter_Time_ReadCounter();
    int countsElapsed = oldCounterVal - newCounterVal;
    if (countsElapsed < 0) countsElapsed += 65535;
    timeElapsed += (double) countsElapsed / 100000.0;
    oldCounterVal = newCounterVal;

    double xDist = (target[0] - currentPos[0]);
    double yDist = (target[1] - currentPos[1]);
    if (yDist == 0) yDist = 0.0000001;
    heading = atan(-xDist/yDist);
    if (yDist < 0) heading += M_PI;
    double dist = sqrt(xDist*xDist + yDist*yDist);
    magnitude = P*dist;
    if (magnitude > 1) magnitude = 1;

    //targetAngle = M_PI/2;

    rotation = 0;
    enableGyro = 1;

    if (dist < targetMargin) {
        goToTarget = 0;
        magnitude = 0;
        rotation = 0;
        sprintf(strbuffer, "Target Reached in %fs\n", timeElapsed);
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
    }

    if (timeElapsed > 1) {
        goToTarget = 0;
        magnitude = 0;
        rotation = 0;
        sprintf(strbuffer, "Time Up\n");
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
    }

    if (timeElapsed > targetTime - flickTime && !rotOn && timeElapsed <
↪   targetTime) {
        rotation = -1;
        sprintf(strbuffer, "Rot on\n");
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
        rotOn = 1;
    }
    if (timeElapsed > targetTime && rotOn) {
        rotation = 0;
        sprintf(strbuffer, "Rot off\n");
```

```
        terminal == 0 ? UART_PutString(strbuffer) : UART_Pi_PutString(strbuffer);
        rotOn = 0;
    }
}
```