ELE 302: Ball Catcher

Ethan Gordon and Luke Pfleger

Car Lab
July 18, 2017

Honor Statement

This paper represents our own work in accordance with University regulations.

Ethan K. Gordon '17

Luke Pfleger '17

Contents

Abstract

In the face of multiple partially-successful attempts in years past, we were determined to build a robot that could catch a ball in real time. We constructed a Holonomic Drive Train that allowed our robot to maneuver at speeds on the order of 6-7 feet per second in any direction, and it could reach those speeds in under a second. Relying on the Pixy Cam, we were able to determine the position of the target ball in 3D space relative to the robot once every 20ms. With a gyroscope, we could keep our robot facing the same direction as it moved to catch the ball. In the end, any ball tossed within a 5-6ft radius of the robot would at very minimum hit the rim. While our robot is far from perfect, we think that we put forward a decent attempt to solve this high-speed, high-percision problem.

1 Drive Train

By far the most involved system on our robot is the drive train. Looking for quick response time and flexibility, we decided to go with a Holonomic Drive.

1.1 Holonomic Drive Theory and Implementation

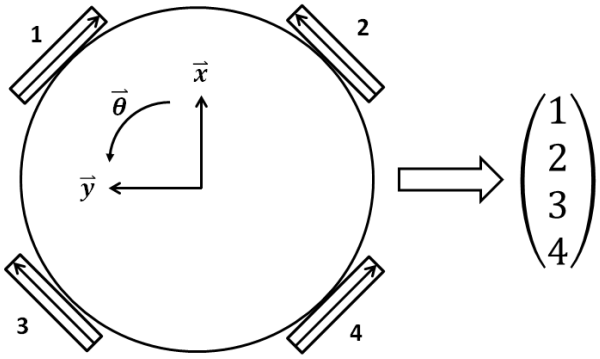

Figure 1: An example of a 4-wheel holonomic drive train. The control of this system can be represented as a column vector of size 4.

Formally, a system is *holonomic* when all constraints in the system can be represented solely as a function of position, rather than velocity. A standard car drive is an example of a *non-holonomic* system, since the center of mass velocity is constrained by the fact that the back wheels are fixed. Put another way: a car cannot drive sideways! Since the ball could end up landing anywhere and we do not have the time to solve an effective parallel parking problem on each throw, we needed a system without any velocity constraints. One way to realize such a system with fixed wheels is to:

1. Place wheels so that the force vectors span the entire two dimensions of translation (x, y) and possibly an extra dimension for rotation (θ). For both the translation and rotation, this requires at least 3 wheels.

2. Allow the wheels to glide orthogonal to their intended force vector, so friction does not modify the direction of the force. The easiest way to do this is to add rollers in the orthogonal direction, making what are called *omni wheels*.

An example using 4 wheels is shown in Figure 1. With 4 wheels, control can be represented as a 4-vector, and it is possible to construct a basis in the three-dimensional parameter space out of orthogonal control vectors:

$$\hat{\mathbf{x}} = \frac{1}{2}\begin{bmatrix}1\\1\\1\\1\end{bmatrix} \tag{1}$$

$$\hat{\mathbf{y}} = \frac{1}{2}\begin{bmatrix} -1 \\ 1 \\ 1 \\ -1 \end{bmatrix} \tag{2}$$

$$\hat{\boldsymbol{\theta}} = \frac{1}{2}\begin{bmatrix} -1 \\ 1 \\ -1 \\ 1 \end{bmatrix} \tag{3}$$

While a 4th basis vector does exist in the control vector space, it has all four wheel forces cancelling out (both front wheels moving forward, both back wheels moving backwards), providing no useful movement, and it is therefore a useless extra dimension.
The code to implement the Drive Train can be found in Section 6.2. The primary function, DriveTrain_Set(), takes as arguments the intended velocity vector (x, y, θ), and multiplies by the unit vector basis to get a control vector:

$$\begin{bmatrix} \text{FL Wheel} \\ \text{FR Wheel} \\ \text{BL Wheel} \\ \text{BR Wheel} \end{bmatrix} = k \begin{bmatrix} 1 & -1 & -1 \\ 1 & 1 & 1 \\ 1 & 1 & -1 \\ 1 & -1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} \tag{4}$$

The vector is scaled (by a constant k) such that the final control vector is clamped (in the case of this specific drive train) to the interval [-500, 500]. Finally, due to the implementation of our firmware explained in Section 6.1, each value in the interval [-500, 500] is mapped to the interval [0, 1000] before being written. The result is a drive train that allows our car to translate in any direction with a zero turn radius, vital for the high-speed maneuvering needed for ball catching.

1.2 Mechanical Design and Part Selection

The most important aspect of the motor and wheel choice is the speed-torque-power trade-off. Gears can be used to raise top speed in exchange for slower acceleration (less torque) or vice versa, but more power is needed in order to increase both. With this trade-off in mind, we went with Pololu's 25mm Diameter Gearmotor. While the power draw is considerable due to a high free current (550mA @ 6V) and stall current (6.5A @ 6V), and therefore required us to construct our own custom controllers from Power MOSFETs (as discussed in Section 1.3), it boasts some impressive mechanical specs. It has a 460RPM free speed and a 53.9 N-cm stall torque. We also chose to use 4-inch omni-wheels from Vex, making for the following final mechanical specs assuming a robot mass of about 3kg:

$$\vec{v}_{max} = (4\pi \text{ in})(460 \text{ rpm}) = 8.029\frac{ft}{s} \tag{5}$$

$$\vec{a}_{max} = \frac{(53.9 \text{ N-cm})}{(3 \text{ kg})(2 \text{ in}} = 11.5\frac{ft}{s^2} \tag{6}$$

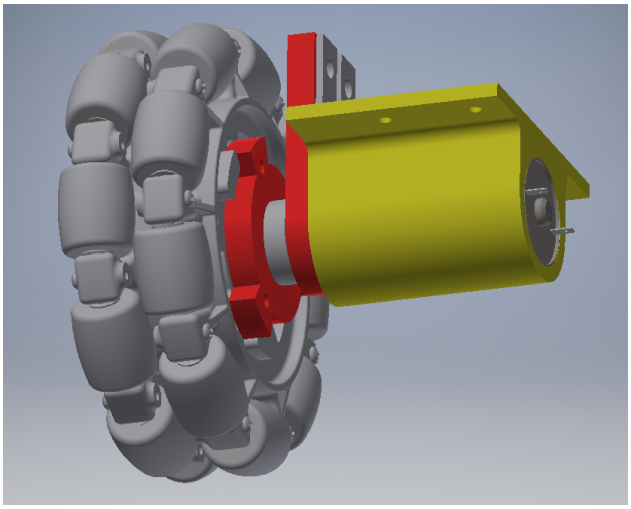

Figure 2: CAD of the wheel and motor assembly. Red represents Acryllic, while Yellow represents a 3D-Printed component.

From these numbers, the robot will be able to reach its maximum speed in about 0.7s. Assuming the ball stays in the air for about 2 seconds, this corresponds with an effective catch radius of about 13ft. In practice, due to the fact that the bot is going slower than free speed, we expected a much smaller radius, but the high theoretical number allowed for plenty of breathing room when tossing the ball.

The mechanical design itself is shown in Figure 2. A 3D-printed mount constrained the motor's movement to one axis and bore the majority of its weight. An acryllic front-plate with two mounting bolt holes constrained the final dimension. The axle of the motor, a D-shaft, jutted out from the acryllic and attached to a hub with a 4-40 set screw. Another acryllic adapter connected the metric hub and the imperial omni-wheel. The final constructed Drive Train can be seen in Figure 3.

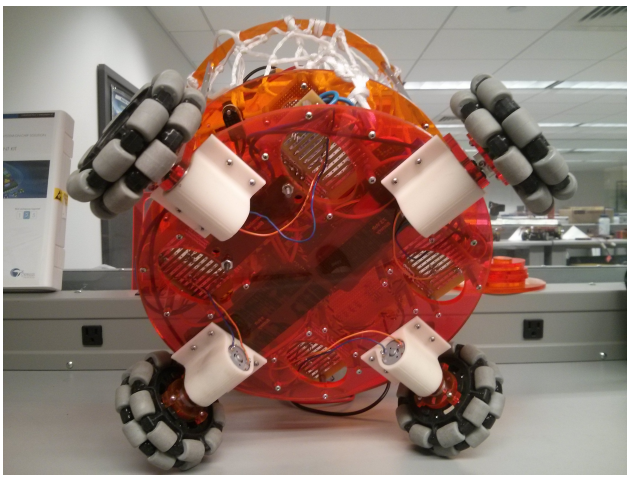

Figure 3: Picture of final constructed Drive Train.

1.3 H-Bridge

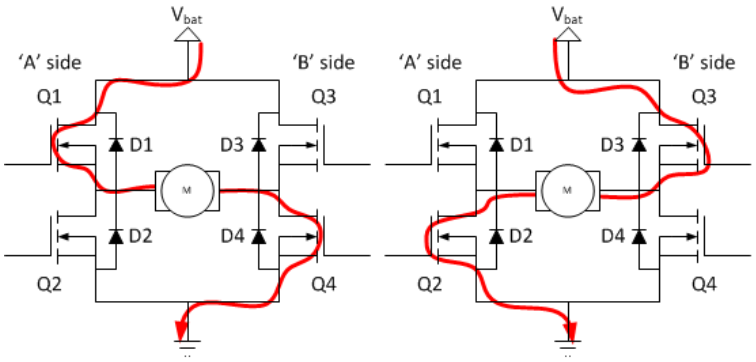

Figure 4: Schematic of an operating H-Bridge. In lock anti-phase drive, the left corresponds to the PWM "on" state, and the right is the PWM "off" state.

All four motors on our Drive Train need independent control, with a speed controller that can sustain 1A of continuous current and pulses that could peak at 6.5A. To meet this high power demand, we constructed four H-Bridges out of power MOSFETs.
The principle behind an H-Bridge is that MOSFETS are placed in a configuration where current can flow through the load (usually a motor) in either direction, depending on which FETs are conducting current, as shown in Figure 4. The final schematic looks like an H, with the horizontal line being the motor and each verticle line a MOSFET. To ensure stable source voltages in all FETs, the two pull-up FETs were PMOS, while the two pull-down FETs were NMOS. This configuration also allowed the gates of the lower FETs to be merged with the gates of the upper FETs, ensuring that the power supply would never be shorted across a conducting PMOS and NMOS pair. Only 2 control signals are required to control each H-Bridge.

Two common ways to control an H-Bridge are called sign-magnitude and lock anti-phase drive. In the former, one side has the NMOS always on, where the intended direction, or sign of the velocity, determines which side. The other side oscillates between NMOS on and PMOS on using a PWM signal, so the duty cycle sets the speed, or magnitude of the velocity. While safer and more stable, this drive scheme can be a little slow, especially in switching direction, due to the extra calculates involved in switching which side gets the PWM signal. Also, sign-magnitude drive requires two uncorrelated input signals. Lock anti-phase drive, on the other hand, gets around these problems by allowing the H-Bridge to be controlled by a single pair of complementary PWM signals. In the "on" state of the PWM signal, one PMOS and one NMOS turns on, and in the "off" state, the other PMOS and NMOS turn on. Therefore, the speed and direction are both set by the duty cycle: 50% duty cycle is a speed of 0, while 100% is full in one direction, and 0% is full in the other. This was the drive scheme that we used for our H-Bridges.

One addition we had to make to the H-Bridge circuit was a comparator. The PSoC outputs a 3.3V signal, which was not high enough to turn the PMOS off. Therefore, we

added an Open Drain Comparator comparing the PSoC output and a reference voltage (1.5V achieved with a voltage divider) that would up the magnitude of the PWM signal to the 7.2V drive voltage via a pull-up resistor on the H-Bridge board. The combined H-Bridge and Comparator schematic is shown in Figure 6, while pictures of both boards are shown in Figure 6. Due to the ability of the comparator to invert the signal by just switching which input had the reference voltage, only 1 signal was actually required per H-Bridge from the PSoC, minimizing some of the wiring work.

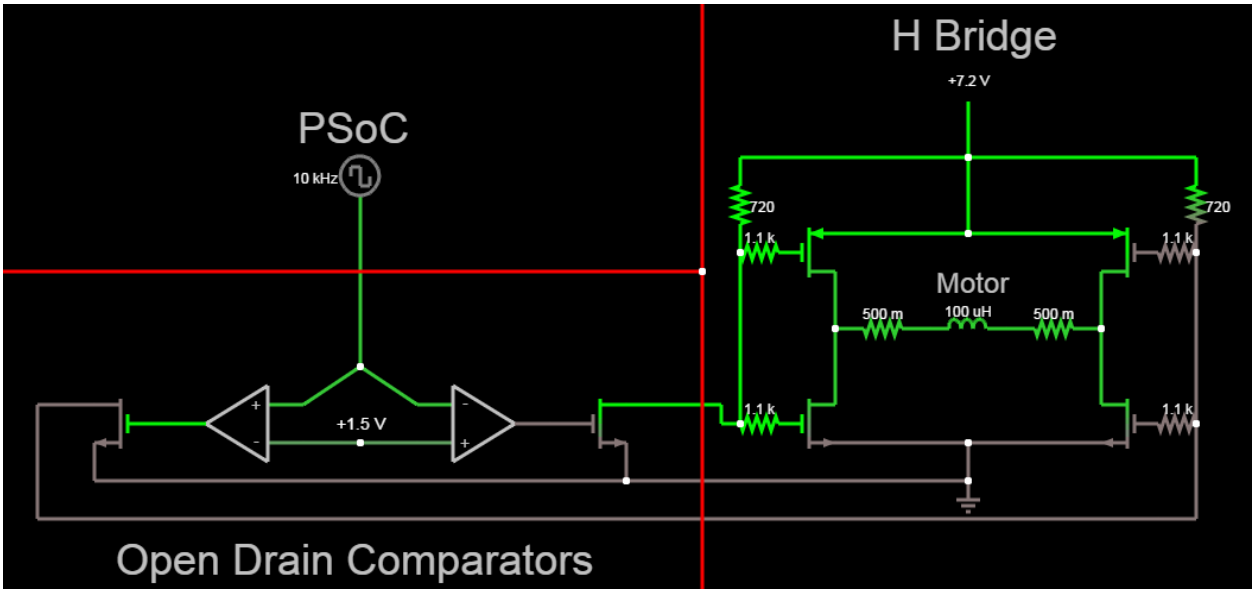

Figure 5: Schematic of whole drive circuit, from PWM signal at PSoC through Open Drain Comparators to the H-Bridge. Inductance and Resistance of motor was determined by empirical measurement of the time constant and rated stall current.

1.4 Pitfall: The Case of the Dead Motors

One of the most involved issues we faced during testing was the Case of the Dead Motors. Originally, our pull-up resistors were 7.2kΩ, and we tried to operate our H-Bridges at 100Hz, similarly to our MOSFET driver from Tasks 1 and 2. When we first connected power, two of our motors proceeded to burn out in approximately 10 seconds. Apparently, our motors had a particularly small inductance, leading to a small time constant on the order of 100μs. Therefore, when a 50% duty cycle PWM signal was sent to the H-Bridge, the current through the motor had time to rise to the stall current (6.5A) in both directions while not moving at all. We were basically electrically stalling our motors. The primary solution was to increase the frequency to near 10kHz, past the cutoff frequency for the inductor, so that the amplitude of the current signal would be severely reduced while the car was stopped.

Unfortunately, yet another problem arose. Our MOSFETs began to heat up exceptionally quickly, and our wheels still did not turn. Using calculations from the respective

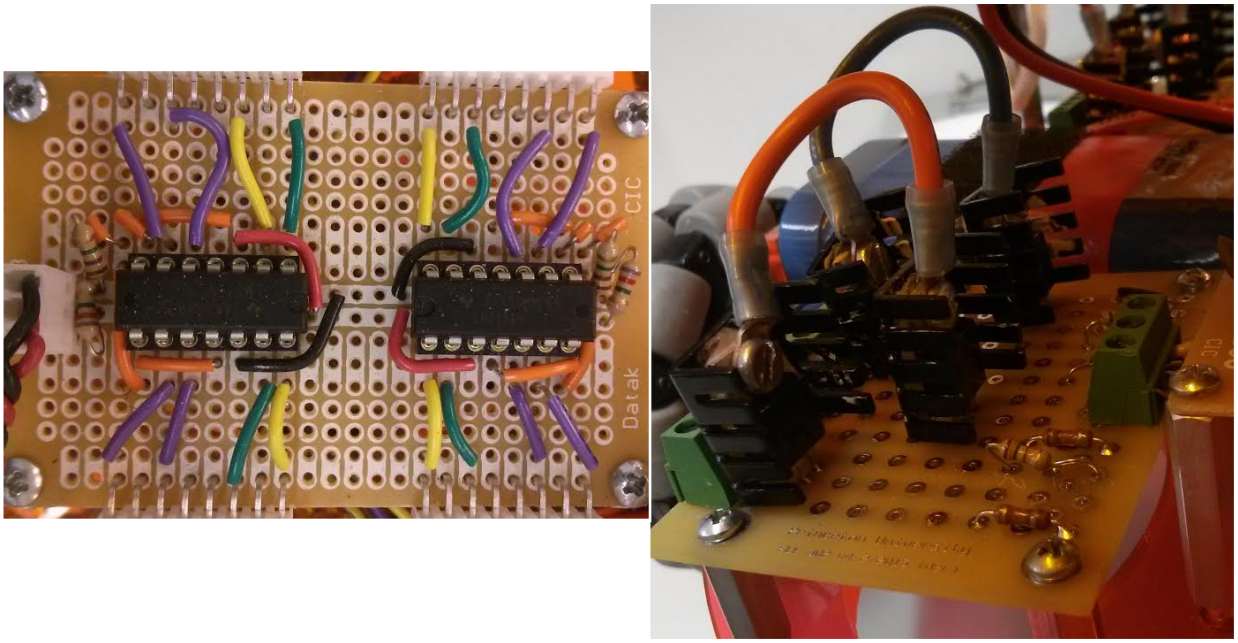

Figure 6: (Left) Photograph of the comparator board with two quad-comparators to produce 8 PWM signals for the 4 H-Bridge Board. (Right) Photograph of a single H-Bridge board. The two MOSFETS connected by their phalanges are a PMOS-NMOS pair with merged drains. Visible are 3 resistors, the 2 1.1kΩ gate buffers and the 720Ω pull-up resistor in the middle. Similar resistors are on the other side of the board for the other pair of gates.

datasheets, we found out that the gate capacitance of our MOSFETs were on the order of 10nF. With a 1.1kΩ buffer and a 7.2kΩ pull-up resistor, we were able to calculate the rise time for our gates:

$$(1.1k\Omega + 7.2k\Omega) * (10nF) = 84\mu s \tag{7}$$

Sure enough, the time constant was comparable to our 10kHz frequency. With a voltage signal above the threshold for the NMOS and below the threshold for the PMOS, we were shorting our battery. Fortunately, those resistor values were designed with the PSoC's Open Drain digital pins, before we switched to using a comparator due to the fact that the PSoC could not handle 5V on its Open Drain pins. Therefore, we just lowered the pull-up resistor by an order of magnitude, to 720Ω, reducing the time constant at the transistor gates. This finally allowed us to operate at 10kHz, putting to rest the Case of the Dead Motors.

2 The Pixy Camera and Gyroscope

With a working drive train that had the speed and acceleration to get us underneath the ball, the next step was to add the sensors that could track the ball's position and keep the robot stable as it moved.

2.1 Mechanical Design and Part Selection

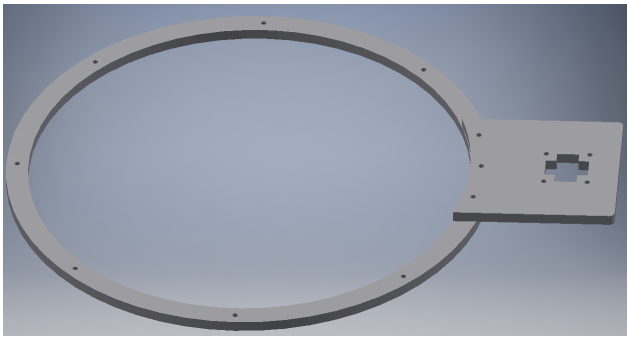

Figure 7: CAD for the rim and camera sub-mount for the pixy camera. Entirely cut from acryllic.

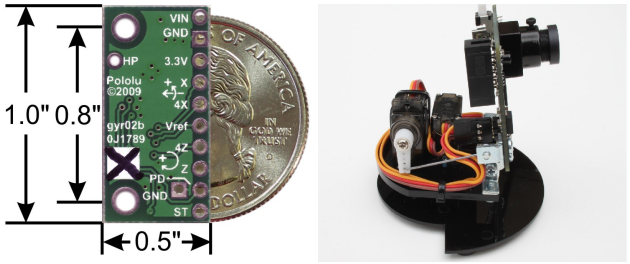

Figure 8: (Left) 2-Axis gyroscope from Pololu, with quarter for scale. (Right) Pixy Cam on the pan tilt mount. The limiting factor on the tilt were the servo and power connections on the back of the camera.

Our part selection for the sensors was fairly straightforward. The Pixy Camera was a shoe-in. It's ability to capture frames, identify the ball, and returns its size and position, all at 50Hz, is almost magical. We decided to buy the first-party pan-tilt mechanical mount. While easy to set up, this mount only had a maximum tilt of approximately 40°, which would severly affect our first control scheme attempt as mentioned in Section 3.2. We were able to bolt this camera mount to an acryllic sub-mount that attached to the acryllic rim on top of the car. Finally, in order to keep the car facing a constant direction, we acquired a simple 2-axis gyroscope (from which we only needed one axis) from Pololu. Both sensors can be seen in Figure 8, and the acryllic submount and rim are shown in Figure 7.

2.2 Data Collection

Once our sensors were collecting data, the next step was to get that data over to the PSoC in a language that the PSoC could understand. For the gyroscope, this was exceedingly easy. The gyroscope could actually be powered via a 3.3V power pin coming directly from the PSoC, and the output was simply an analog signal proportional to the angular velocity of the car, with $\frac{3.3V}{2} = 1.65V$ corresponding with no rotation. The gyro ended up needing no

hardware besides the PSoC itself to function properly. Our "circuit" for the gryo is shown in Figure 9.

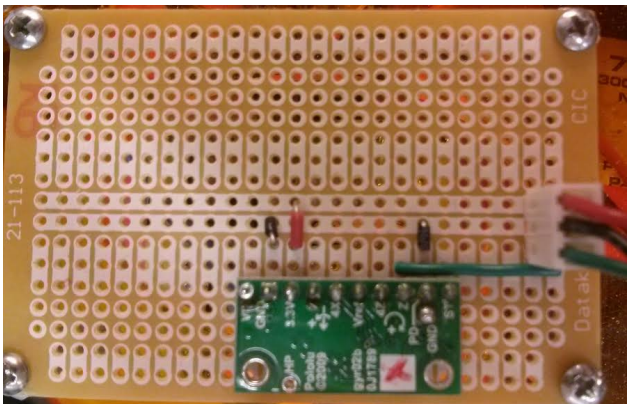

Figure 9: Gyroscope "circuit." Power comes from the 3.3V pin on the PSoC, and the output

Getting data off of the pixy cam was a little bit more involved. First of all, we needed the position of the servos. Rather than use the camera's API and slow down the serial data connection, we were actually able to listen to the data signal that the Pixy output directly to the servos by using a J-connector Y-junction. Just like in Task 2, the data consisted of a PWM signal with a pulse length that varied between 1ms (full left) and 2ms (full right), with 1.5ms being centered. The next step was to pull the image information (such as the x and y coordinates of the ball in the frame) from the camera over the serial connection. The default firmware on the Pixy came with two default programs:

1. A program that took the largest object in the frame and threw all the data out onto a configurable serial connnection (choice of SPI, I^2C, UART, or just a plain DAC).

2. A program that internally controlled the servos of the pan-tilt mount so that the largest detected object would be in the center of the screen.

Unfortunately, we wanted the camera to do *both*, so we downloaded the open-source firmware code and the specific Keil uVision5 IDE that could compile it. All we had to do was literally add one line of code to the pan-tilt demo program, a line that really had no reason for being absent:

```
ser_getSerial()->update();
```

Thre rest of that firmware file can be found in the Appendix, Section 6.1.
Now we had the PWM for both servos and all of the image information on UART going into the PSoC at 50Hz for the firmware and software to process.

3 Control Scheme

3.1 PSoC Firmware

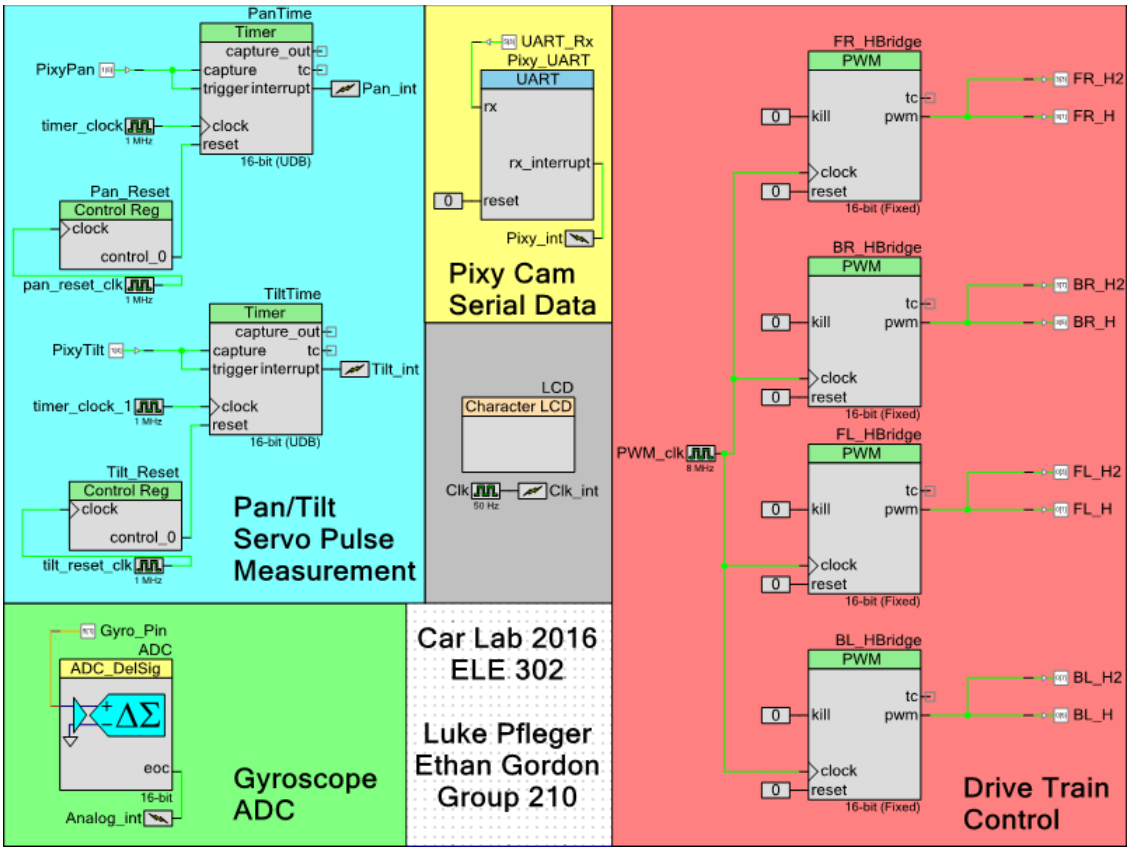

Figure 10: PSoC firmware schematic.

Due to the relative simplicity of our hardware interfaces, our firmware was fiarly straightforward. The Drive Train control consisted of four PWM modules to send to the four H-Bridges. Each PWM module had a 0.1% resolution in the duty cycle, so writing a 0 corresponded with 0%, while writing a 1000 corresponded with 100%. We had an ADC to read the analog value from the gyroscope, throwing an interrupt at each sample. While the ADC had a 16-bit resolution, the number 65535 corresponded with 5V, so the effective resolution was closer to 15-bits, which was still plenty precise for our uses. For the servo pulse measurement, we used a Timer module. The rising edge of the pulse would trigger the Timer to start, while the falling edge would trigger a capture and interrupt. A Control Register sends a pulse to reset the timer. In code, the interrupt pulls the capture data from the Timer, and then triggers the Control Register. Finally, a UART module facilitates incoming serial data from the Pixy Cam. It will throw an interrupt every time a byte is received.

3.2 Attempt 1: Basic PID

Our first attempt at a control scheme was just a simple set of PID loops on each degree of freedom, as shown in our Old Main Loop in Section 6.4. For rotation, we set up a PI loop with the gyroscope so the robot would always face the same direction. We initially set our gains very small (since the error from the gyro was on the order of 10^3), and fine-tuned appropriately. So that the I gain would not explode due to random noise, we added a deadband to the error as well, clamping it to 0 if it was within 30. During testing, we were able to drive forward a full 40 feet with less than 1 foot of deviation.

For the translational degrees of freedom, we set up a simple PD loop on the angle of the servos. If the tilt angle was above approximately 30°, we moved the robot backwards, else we moved the robot forwards. The PD loop on the pan angle worked similarly, with the center of the robot being the setpoint.

Overall, this control scheme worked fairly well to put the robot close to the landing point of the ball, but it was always far enough off that the ball would hit the outside of the rim. Also, even when the ball hit the inside of the rim, the oscillations due to the delay in the servo motion kept the bot moving in stead state, causing the ball to bounce off of the rim and get away from the bot. Also, while the camera could track the ball up to 60°, the servo itself only made it to an angle of about 40°, leaving very little headroom for the PID algorithm to operate, and preventing the set point from being much higher than 30°, even when it was clear that the robot usually ended up too far back to catch the ball. Another smoother control scheme that could track the ball beyond a 40° tilt.

3.3 Attempt 2: Predictive PID

Given the fact that the previous PID loop was consistently 1-2ft off, we eventually decided to try a predictive algorithm instead. The rotation control did not change (as it was working fairly well to keep the robot facing the same direction), but we modified the translation control to predict what the X and Y coordinates of the ball would be (relative to the robot) as it landed, called the target coordinates. We set up a PD loop on each target coordinate, with (0,0) as the set point.

The first step to predicting the flight path of the ball was to determine its position in 3D space using spherical coordinates (r, θ, ϕ), where θ is the altitude and ϕ is the azimuth. We were able to determine the spherical radius r simply from the perceived width of the ball in pixels, a value provided by the Pixy Cam. Given that the horizontal field of the view of the Pixy Cam was 75°, the ball was 7in in diameter, and the horizontal resolution of the Pixy Cam was 320px, we could determine the spherical radius r to the ball as such:

$$tan(\frac{75^\circ}{2}) = \frac{\text{half-screen width in inches}}{r} \tag{8}$$

$$\text{half screen width in inches} = \text{half screen width in px} * \frac{\text{half ball width in inches}}{\text{half ball width in px}} \tag{9}$$

Let the measured width of the ball in pixels be w, plugging Equation 9 into Equation 8:

$$tan(\frac{75^\circ}{2}) = 160px * \frac{7in}{\frac{w}{2}r} \tag{10}$$

$$r = \frac{160px * 7in * 2}{w * tan(\frac{75^\circ}{2})} \tag{11}$$

This formula happend to be accurate to within a few inches out to at least 5 feet.

Once we had an accurate radial measurement, we turned to the angle measurement. Each angle was the sum of the angle of the servo and a correction based on the ball's location in the frame of the camera. The former is a simple linear map from the measured pulse length to the actual angle in degrees. The latter involved using the x and y coordinates of the ball in the frame. We'll use the pan angle and the x coordinate as an example:

$$tan(\text{pan correction}) = \frac{x}{r \text{ in px}} = \frac{x}{r * \frac{w}{7in}} = \frac{x * 7in}{r * w} \tag{12}$$

$$\phi = \angle pan + atan(\frac{x * 7in}{r * w}) \tag{13}$$

Analogously, letting h be the height of the ball in pixels:

$$\theta = \angle tilt + atan(\frac{y * 7in}{r * h}) \tag{14}$$

Once we had the spherical coordinates, we found their derivates by taking the difference between this frame's position and the last frame's position:

$$\begin{bmatrix} \dot{r} \\ \dot{\theta} \\ \dot{\phi} \end{bmatrix} = \left[\begin{bmatrix} r \\ \theta \\ \phi \end{bmatrix}_n - \begin{bmatrix} r \\ \theta \\ \phi \end{bmatrix}_{n-1} \right] * 50Hz \tag{15}$$

While we had a dedicated clock interrupt running at 50Hz to calculate the spherical coordinates, our main loop was taking those coordinates and converting them to cartesian:

$$x = rcos(\theta)cos(\phi) \tag{16}$$

$$\dot{x} = \dot{r}cos(\theta)cos(\phi) - rsin(\theta)cos(\phi)\dot{\theta} - rcos(\theta)sin(\phi)\dot{\phi} \tag{17}$$

$$y = rcos(\theta)sin(\phi) \tag{18}$$

$$\dot{y} = \dot{r}cos(\theta)sin(\phi) - rsin(\theta)sin(\phi)\dot{\theta} + rcos(\theta)cos(\phi)\dot{\phi} \tag{19}$$

$$z = rsin(\theta) \tag{20}$$

$$\dot{z} = \dot{r}sin(\theta) + rcos(\theta)\dot{\theta} \tag{21}$$

Finally, we had to determine the *target* X and Y coordinates. We first determined how long the ball would be in the air by finding the time when the ball's Z coordinate became 0:

$$0 = z + \dot{z}t - \frac{1}{2}gt^2 \tag{22}$$

$$t = \frac{\dot{z} + \sqrt{\dot{z}^2 + 2gz}}{g} \tag{23}$$

$$\textbf{X Target} = x + t\dot{x} \tag{24}$$

$$\textbf{Y Target} = y + t\dot{y} \tag{25}$$

As stated before, we set up a PD loop on each of these targets to complete the control scheme. While there are a lot of calculations here, the PSoC was *just* fast enough to complete them all. Occasionally, the clock interrupt that calculate the spherical coordinates would re-execute before the previous interrupt completed, causing some race conditions in memory. Fortunately, a little aveaging of the components was able to smooth out those outliers. The result was a more predictable and accurate control scheme, that was able to have the ball at least hit the rim the majority of the time.

4 Future Improvements and Discussion

While our final result did a fairly good job of getting the ball to hit the rim for the vast majority of throws, there are still many improvements that could have made the bot perform even better. First and foremost, we should have built a funnel or wider rim. For every ball that the robot caught, there were at least 3-5 balls that hit the top or inside of the rim and bounced away. Secondly, we should have either 3D-printed our own Pixy Mount or used a different third-party mount. It was pretty much impossible for our robot to move backwards, as the time between when the robot detected a negative X target and when it completely lost track of the ball was too small. Finally, on the control side, we failed to take into account the robot's own velocity when determining the ball's velocity. This added an effective damping factor to our approach, forcing us to use higher P gains that would occassionally overshoot the landing spot completely. For even more fine-tuning, the optimal P gain was heavily dependent on how much charge was left in the battery. Adding encoders to every wheel to ensure that a value of, for example, 200 *always* corresponds with the exact same speed would be a much better way of handling variable supply voltages.

Overall, we are happy with the results of this project. Whereas last year's attempt had to slowly move the ball to the ground to get the robot to track it at a decent speed, we were able to hold our own when the ball was tossed. We constructed our robot pretty much from scratch, from the chassis to the H-Bridges, relying heavily on CAD to make everything fit together. So even if we had to dunk our missed catches half of the time, we are glad that we demonstrated that this "impossible, ambitious" project is not nearly as impossible as it first seemed.

5 Image Credits

Figure 4:

- http://modularcircuits.tantosonline.com/blog/articles/h-bridge-secrets/h-bridges-the-basics/

Figure 8:

- https://www.pololu.com/product/1266 (Gyro)
- http://www.amazon.com/Tilt-Servo-Motor-Pixy-CMUcam5/dp/B00IVOEN1Y (Pan/Tilt Mount)

6 Code Appendix

6.1 Camera Firmware: Pan-Tilt Program

```
//
// begin license header
//
// This file is part of Pixy CMUcam5 or "Pixy" for short
//
// All Pixy source code is provided under the terms of the
// GNU General Public License v2 (http://www.gnu.org/licenses/
   ↪ gpl-2.0.html).
// Those wishing to use Pixy source code, software and/or
// technologies under different licensing terms should contact
   ↪ us at
// cmucam@cs.cmu.edu. Such licensing terms are available for
// all portions of the Pixy codebase presented here.
//
// end license header
//

#include "progpt.h"
#include "pixy_init.h"
#include "camera.h"
#include "qqueue.h"
#include "blobs.h"
#include "spi.h"
#include "rcservo.h"
#include "cameravals.h"
#include "conncomp.h"
#include "param.h"
```

```
#include "serial.h"

extern Qqueue *g_qqueue;
extern Blobs *g_blobs;

Program g_progPt =
{
        "Pan/tilt_demo",
        "perform␣pan/tilt␣tracking",
        ptSetup,
        ptLoop
};

static ServoLoop g_panLoop(PAN_AXIS, 500, 800);
static ServoLoop g_tiltLoop(TILT_AXIS, 700, 900);


ServoLoop::ServoLoop(uint8_t axis, uint32_t pgain, uint32_t
   ↪ dgain)
{
        m_pos = RCS_CENTER_POS;
        m_axis = axis;
        m_pgain = pgain;
        m_dgain = dgain;
        m_prevError = 0x80000000;
}

void ServoLoop::update(int32_t error)
{
        int32_t vel;

        if (m_prevError!=0x80000000)
        {
                vel = (error*m_pgain + (error - m_prevError)*
                   ↪ m_dgain)/1000;
                m_pos += vel;
                if (m_pos>RCS_MAX_POS)
                        m_pos = RCS_MAX_POS;
                else if (m_pos<RCS_MIN_POS)
                        m_pos = RCS_MIN_POS;

                rcs_setPos(m_axis, m_pos);
```

```
                //cprintf("%d %d %d\n", m_axis, m_pos, vel);
        }
        m_prevError = error;
}

void ServoLoop::reset()
{
        m_pos = RCS_CENTER_POS;
        rcs_setPos(m_axis, m_pos);
}

void ServoLoop::setGains(int32_t pgain, int32_t dgain)
{
        m_pgain = pgain;
        m_dgain = dgain;
}


int ptSetup()
{
        // setup camera mode
        cam_setMode(CAM_MODE1);

        // extend range of servos (handled in params)
        // rcs_setLimits(0, -200, 200); (handled in rcservo
           ↪ params)
        // rcs_setLimits(1, -200, 200); (handled in rcservo
           ↪ params)

        // Increasing the PWM frequency makes the servos
           ↪ zippier.
        // Pixy updates at 50 Hz, so a default servo update
           ↪ freq of 50 Hz
        // adds significant latency to the control loop---
           ↪ increasing to 100 Hz decreases this.
        // Increasing to more than 130 Hz or so creates buzzing
           ↪ , prob not good for the servo.
        // rcs_setFreq(100); (handled in rcservo params)

        ptLoadParams();

        g_panLoop.reset();
```

```
        g_tiltLoop.reset();

        // load lut if we've grabbed any frames lately
        if (g_rawFrame.m_pixels)
                cc_loadLut();

        // setup qqueue and M0
        g_qqueue->flush();
        exec_runM0(0);

        return 0;
}

void ptLoadParams()
{
        prm_add("Pan P gain", PRM_FLAG_SIGNED,
                "@c Pan/tilt_Demo Pan axis proportional gain (
                  ↪ default 350)", INT32(350), END);
        prm_add("Pan D gain", PRM_FLAG_SIGNED,
                "@c Pan/tilt_Demo Pan axis derivative gain (
                  ↪ default 600)", INT32(600), END);
        prm_add("Tilt P gain", PRM_FLAG_SIGNED,
                "@c Pan/tilt_Demo Tilt axis proportional gain (
                  ↪ default 500)", INT32(500), END);
        prm_add("Tilt D gain", PRM_FLAG_SIGNED,
                "@c Pan/tilt_Demo Tilt axis derivative gain (
                  ↪ default 700)", INT32(700), END);

        int32_t pgain, dgain;

        prm_get("Pan P gain", &pgain, END);
        prm_get("Pan D gain", &dgain, END);
        g_panLoop.setGains(pgain, dgain);

        prm_get("Tilt P gain", &pgain, END);
        prm_get("Tilt D gain", &dgain, END);
        g_tiltLoop.setGains(pgain, dgain);
}


int ptLoop()
{
```

```
        int32_t panError, tiltError;
        uint16_t x, y;
        BlobA *blob, *blobs;
        BlobB *ccBlobs;
        uint32_t numBlobs, numCCBlobs;

        // create blobs
        g_blobs->blobify();

        blob = g_blobs->getMaxBlob();
        if (blob)
        {
                x = blob->m_left + (blob->m_right - blob->
                   ↪ m_left)/2;
                y = blob->m_top + (blob->m_bottom - blob->m_top
                   ↪ )/2;

                panError = X_CENTER-x;
                tiltError = y-Y_CENTER;

                g_panLoop.update(panError);
                g_tiltLoop.update(tiltError);
        }

        // send blobs
        g_blobs->getBlobs(&blobs, &numBlobs, &ccBlobs, &
           ↪ numCCBlobs);
        cc_sendBlobs(g_chirpUsb, blobs, numBlobs, ccBlobs,
           ↪ numCCBlobs);

        ////////////////////////// Added Line
           ↪ ////////////////////////////////
        ser_getSerial()->update();
        ////////////////////// End Added Line
           ↪ ////////////////////////////////

        cc_setLED();

        return 0;
}
```

6.2 Drive Train

6.2.1 Interface

```
/* ========================================
 *
 * Copyright YOUR COMPANY, THE YEAR
 * All Rights Reserved
 * UNPUBLISHED, LICENSED SOFTWARE.
 *
 * CONFIDENTIAL AND PROPRIETARY INFORMATION
 * WHICH IS THE PROPERTY OF your company.
 *
 * ========================================
*/

#ifndef DRIVE_TRAIN
#define DRIVE_TRAIN

void DriveTrain_Start(void);

void DriveTrain_Set(int x, int y, int a);

void DriveTrain_Stop(void);

#endif

//[] END OF FILE
```

6.2.2 Implementation

```
/* ========================================
 *
 * Copyright YOUR COMPANY, THE YEAR
 * All Rights Reserved
 * UNPUBLISHED, LICENSED SOFTWARE.
 *
 * CONFIDENTIAL AND PROPRIETARY INFORMATION
 * WHICH IS THE PROPERTY OF your company.
 *
 * ========================================
*/

#include "drivetrain.h"
```

```
#include <device.h>

/*constants*/
#define FL_OFFSET 0
#define FR_OFFSET 0
#define BL_OFFSET 0
#define BR_OFFSET 0

#define DEADBAND 10

void DriveTrain_Start(void)
{
    /* Start PWM Signals */
    FL_HBridge_Start();
    FR_HBridge_Start();
    BL_HBridge_Start();
    BR_HBridge_Start();

    DriveTrain_Stop();
}

void DriveTrain_Set(int x, int y, int a)
{
    /*initialize motor write values*/
    int writeValues[4] = {0};
    int max = 0;
    int i;

    /*project (x, y, a) vector onto 4-motor 4-vector space*/
    writeValues[0] = x + y - a;
    writeValues[1] = -(-x + y + a);
    writeValues[2] = -x + y - a;
    writeValues[3] = -(x + y + a);

    /*if any value larger than 5000 (our maximum), scale to
      ↪ 5000*/
    for (i = 0; i < 4; i++)
    {
        if (writeValues[i] > max)
            max = writeValues[i];
        else if (-writeValues[i] > max)
            max = -writeValues[i];
```

```
        /* Deadband */
        if ((writeValues[i] < DEADBAND) && (writeValues[i] > -
           ↪ DEADBAND))
            writeValues[i] = 0;
    }
    if (max > 500)
    {
        float div = (float)max / 500.0f;

        for (i = 0; i < 4; i++) {
            writeValues[i] = (int)((float)(writeValues[i])/div)
               ↪ ;
            if (writeValues[i] < -500)
                writeValues[i] = -500;
        }
    }

    /*Convert them from -500~500 to 0~1000*/
    writeValues[0] += (500 + FL_OFFSET);
    writeValues[1] += (500 - FR_OFFSET);
    writeValues[2] += (500 + BL_OFFSET);
    writeValues[3] += (500 - BR_OFFSET);

    FL_HBridge_WriteCompare(writeValues[0]);
    FR_HBridge_WriteCompare(writeValues[1]);
    BL_HBridge_WriteCompare(writeValues[2]);
    BR_HBridge_WriteCompare(writeValues[3]);
}

void DriveTrain_Stop(void)
{
    FL_HBridge_WriteCompare(500 + FL_OFFSET);
    FR_HBridge_WriteCompare(500 - FR_OFFSET);
    BL_HBridge_WriteCompare(500 + BL_OFFSET);
    BR_HBridge_WriteCompare(500 - BR_OFFSET);

}

/* [] END OF FILE */
```

6.3 Main Control Loop

```
/* ========================================
 *
 * Car Lab 2016: Independent Project
 * Ethan Gordon, Luke Pfleger
 *
 * Goal: Catch a ball.
 *
 * ========================================
*/
#include <device.h>
#include <stdio.h>
#include <math.h>
#include "drivetrain.h"

/* Pixy Params */
static short xScreen = 0;
static short yScreen = 0;
static short width = 0;
static short height = 0;

/* UART States */
#define STATE_SYNC 0
#define STATE_READ 1

/* Offsets */
#define X_OFFSET 160
#define Y_OFFSET 100

#define GYRO_SET 24425

#define GYRO_DEADBAND 30

/* Constants */
#define PI 3.14159f
#define GRAVITY 386.1f /* in/s^2 */
#define ALPHA 0.05f
#define ALPHAZ 0.2f


/* Trig Functions */
#define COS(x) (1 - (x)*(x)/2.0f)
#define SIN(x) ((x) - (x)*(x)*(x)/6.0f )
```

```
#define DEG(x) (x)*(180.0f / 3.14159f)
#define RAD(x) (x)*(3.14159f / 180.0f)

static int watchdog = 0;

/* Read data from the Pixy Cam */
static int dataRead = 0;
CY_ISR(uart)
{
    static int state = STATE_SYNC;
    static short word = 0;
    static int readPos = 0;

    unsigned char byte = Pixy_UART_GetChar();

    if (state == STATE_SYNC) {
        switch (readPos) {
        case 0:
            readPos = (byte == 0x55) ? 1 : 0;
            break;
        case 1:
            readPos = (byte == 0xaa) ? 2 : 0;
            break;
        case 2:
            if (byte == 0x55) {
                readPos = 3;
            } else {
                word = byte;
                readPos = 1;
                state = STATE_READ;
            }
            break;
        case 3:
            readPos = 0;
            state = (byte == 0xaa) ? STATE_READ : STATE_SYNC;
        break;
        default:
            readPos = 0;
        }
    } else {
        if (readPos%2) {
            word += (((short)byte) << 8);
```

```
                switch(readPos){
                case 5:
                    xScreen = word;
                break;
                case 7:
                    yScreen = word;
                break;
                case 9:
                    width = word;
                break;
                case 11:
                    height = word;
                    state = STATE_SYNC;
                    readPos = -1;
                    dataRead = 1;
                    watchdog = 0;
                break;
                }
            } else {
                word = byte;
            }
            readPos++;
        }

}

/* Read Gyro Data */
static int gyro = GYRO_SET;
CY_ISR(analog_isr)
{
    gyro = ADC_GetResult16();
}

/* Read the pan and tilt data from Pixy cam */
static float pan = 0.0f;

CY_ISR(pan_isr)
{
    int panRead = (int)PanTime_ReadCapture();
    pan = (float)(panRead) * 0.0967049f - 799.878f; /* Linear
       ↪ Interpolation from Wolfram */
    Pan_Reset_Write(1);
```

```
}

static float tilt = 0;

CY_ISR(tilt_isr)
{
    int tiltRead = (int)TiltTime_ReadCapture();
    tilt = 2.0f * ((float)(tiltRead) * 0.0367647f - 289.154f);
       ↪ /* Linear Interpolation from Wolfram */
    if(tilt > 40.0f) tilt = 40.0f;
    Tilt_Reset_Write(1);
}

/* Spherical */
static float r = 0.0f;
static float phi = 0.0f;
static float theta = 0.0f;

static float dr = 0.0f;
static float dphi = 0.0f;
static float dtheta = 0.0f;

CY_ISR(clk_isr)
{
    char tstr[17] = {0};
    static float rPrev = 0.0f;
    static float phiPrev = 0.0f;
    static float thetaPrev = 0.0f;

    watchdog++;

    if(dataRead) {
        float tanTheta, camTheta, heightPredicted;
        dataRead = 0;

        /* Calculate Spherical Radius */
        r = (320.0f * 7.0f) / (2.0f * (float)(width) * 0.767f);
           ↪  /* r = pix * diameter / (pixWidth * 2*tan(75/2))
           ↪  */
        dr = 50.0f * (r - rPrev);
        rPrev = r;
```

```
        /* Calculate Theta, include Camera Tilt and Y position
           ↪ */
        heightPredicted = (1481.14075f) / r;
        tanTheta = (7.0f * (yScreen - Y_OFFSET)) / (r *
           ↪ heightPredicted);
        camTheta = DEG(tanTheta*tanTheta*tanTheta / 3.0f -
           ↪ tanTheta);
        if ((camTheta < 2.0f) && (camTheta > -2.0f)) camTheta =
           ↪  0.0f;
        if (tilt < 0.0f)
            theta = 0.0f;
        else theta = tilt + camTheta;
        dtheta = 50.0f * (theta - thetaPrev);
        thetaPrev = theta;

        /* Calculate Phi, ignore X location, only use Camera
           ↪ Pan */
        phi = pan;
        dphi = (phi - phiPrev) * 50.0f;
        phiPrev = phi;

    }
}

void main()
{
    int finished = 0;
    char tstr[17] = {0};

    /* Start STUFFZ */
    DriveTrain_Start();
    LCD_Start();
    ADC_Start();
    ADC_StartConvert();

    /* Start Counters */
    PanTime_Start();
    TiltTime_Start();

    /* Interrupts */
    Pixy_int_Start();
    Pixy_int_SetVector(uart);
```

```
Pan_int_Start();
Pan_int_SetVector(pan_isr);
Tilt_int_Start();
Tilt_int_SetVector(tilt_isr);
Clk_int_Start();
Clk_int_SetVector(clk_isr);
Analog_int_Start();
Analog_int_SetVector(analog_isr);

/* Start UART */
Pixy_UART_Start();

CyGlobalIntEnable;
while(!finished)
{
    float z, dz, x, dx, y, dy, dxMeas, dyMeas, dzMeas,
      ↪ radical, tol, kpx;
    static float xTarget = 0.0f;
    static float yTarget = 0.0f;

    float gyro_err, gyro_p, gyro_i;
    static float gyro_sum = 0.0f;

    z = r * SIN(RAD(theta));
    dzMeas = dr * SIN(RAD(theta)) + r*COS(RAD(theta))*RAD(
      ↪ dtheta);

    x = r * COS(RAD(theta)) * COS(RAD(phi)) - 9.0f; /*
      ↪ Center on center of basket */
    dxMeas = dr * COS(RAD(theta)) * COS(RAD(phi)) - r * SIN
      ↪ (RAD(theta)) * COS(RAD(phi))*RAD(dtheta) - r *
      ↪ COS(RAD(theta)) * SIN(RAD(phi))*RAD(dphi);

    y = -r * COS(RAD(theta)) * SIN(RAD(phi));
    dyMeas = -dr * COS(RAD(theta)) * SIN(RAD(phi)) + r *
      ↪ SIN(RAD(theta)) * SIN(RAD(phi))*RAD(dtheta) - r *
      ↪  COS(RAD(theta)) * COS(RAD(phi))*RAD(dphi);

    /* Average Velocities */
    if (dyMeas > 60.0f) dyMeas = 0.0f;
    if (dyMeas < -60.0f) dyMeas = 0.0f;
    if (dxMeas > 60.0f) dxMeas = 0.0f;
```

```
if (dxMeas < -60.0f) dxMeas = 0.0f;
if(z < 3.0f) {
    dx = dxMeas;
    dy = dyMeas;
} else {
    dx = ALPHA*dxMeas + (1.0f-ALPHA)*dx;
    dy = ALPHA*dyMeas + (1.0f-ALPHA)*dy;
}
dz = ALPHAZ*dzMeas + (1.0f-ALPHAZ)*dz;

radical = dz * dz + 2.0f*GRAVITY*z;
if(radical < 0)
    tol = 0.0f;
else
    tol = (dz + (float)sqrt((double)radical)) / GRAVITY
      ↪ ;

xTarget = x + tol*dx;
yTarget = y + tol*dy;

sprintf(tstr, "x: %+2.2f  ", xTarget);

LCD_Position(0, 0);
LCD_PrintString(tstr);


sprintf(tstr, "y: %+2.2f  ", yTarget);

LCD_Position(1, 0);
LCD_PrintString(tstr);

/* Gyro PID Loop */
gyro_err = gyro - GYRO_SET;
if ((gyro_err < GYRO_DEADBAND) && (gyro_err > -
  ↪ GYRO_DEADBAND))
    gyro_err = 0;
gyro_p = 0.5f * gyro_err;

gyro_sum += gyro_err;
/* Clamp Sum */
if (gyro_sum > 10000) gyro_sum = 10000;
else if (gyro_sum < -10000) gyro_sum = -10000;
```

```
            gyro_i = 0.01f * gyro_sum;

            if (xTarget < 0)
                kpx = 100.0f;
            else kpx = 35.0f;

            if ((tilt > 10.0f) && (watchdog < 10)) {
                DriveTrain_Set(-45.0f * yTarget, kpx * xTarget, (
                    ↪ int)(gyro_p + gyro_i));
            } else DriveTrain_Stop();

        }

        /* Stop Drive Train */
        DriveTrain_Stop();

        for(;;){}
    }

    /* [] END OF FILE */

    /* Saved Junk
    static float xLast = 0.0f;
        static float yLast = 0.0f;
        static float zLast = 0.0f;
        if (width != 0 && height != 0)
        {
            float tanPhi, tanTheta, xDotTest, yDotTest, zDotTest, r
                ↪ , phi, theta;
            float tof;
            r = (320.0f * 7.0f) / (2.0f * (float)(width) * 0.767f);
            tanPhi = (7.0f * (x - X_OFFSET)) / (r * width);
            tanTheta = (7.0f * (y - Y_OFFSET)) / (r * height);
            phi = pan + (tanPhi - tanPhi*tanPhi*tanPhi / 3) *
                ↪ (180.0f / PI);
            theta = tilt - (tanTheta - tanTheta*tanTheta*tanTheta /
                ↪  3) * (180.0f / PI);


            xBall = r * cos(theta * PI / 180.0f) * cos(phi * PI /
                ↪ 180.0f);
            yBall = r * cos(theta * PI / 180.0f) * sin(phi * PI /
```

```
            ↪ 180.0f);
        zBall = r * sin(theta * PI / 180.0f);

        if ((xBall - xLast > 100) || (xBall - xLast < -100))
            ↪ xBall = xLast;
        if ((yBall - yLast > 100) || (yBall - yLast < -100))
            ↪ yBall = yLast;
        if ((zBall - zLast > 50) || (zBall - zLast < -50))
            ↪ zBall = zLast;


        xDotTest = (xBall - xLast) * 50.0f;
        yDotTest = (yBall - yLast) * 50.0f;
        zDotTest = (zBall - zLast) * 50.0f;

        xDot = ALPHA*xDotTest + (1 - ALPHA)*xDot;
        yDot = ALPHA*yDotTest + (1 - ALPHA)*yDot;
        zDot = ALPHA*zDotTest + (1 - ALPHA)*zDot;

        xLast = xBall;
        yLast = yBall;
        zLast = zBall;


        tof = (zDot + sqrt(zDot*zDot + 2*GRAVITY*zBall)) /
            ↪ GRAVITY;
        xTarget = xBall + tof*xDot;
        yTarget = yBall + tof*yDot;
    }
    */
```

6.4 Old Control Loop

```
/* =======================================
 *
 * Copyright YOUR COMPANY, THE YEAR
 * All Rights Reserved
 * UNPUBLISHED, LICENSED SOFTWARE.
 *
 * CONFIDENTIAL AND PROPRIETARY INFORMATION
 * WHICH IS THE PROPERTY OF your company.
 *
 * =======================================
```

```
*/
#include <device.h>
#include <stdio.h>
#include "drivetrain.h"

#define PAN_OFFSET 8320
#define TILT_OFFSET 8180
#define GYRO_OFFSET 0

#define RANGE_CUTOFF 1900
#define GYRO_DEADBAND 30
#define RANGE_ALPHA 0.2f

#define ALPHA 0.9f

#define GYRO_SEL 0
#define RANGE_SEL 1

/* PID Params */
#define PAN_SET 0
#define RANGE_SET 53000
#define GYRO_SET 24420

static short pan = 0;

CY_ISR(pan_isr)
{
    pan = (int)(ALPHA * (float)((int)PanTime_ReadCapture() - (
       ↪ int)PAN_OFFSET) + (1.0f - ALPHA)*(float)(pan));
    Pan_Reset_Write(1);
}

static short tilt = 0;

CY_ISR(tilt_isr)
{
    tilt = (int)(ALPHA * (float)((int)TiltTime_ReadCapture() -
       ↪ (int)TILT_OFFSET) + (1.0f - ALPHA)*(float)(tilt));
    Tilt_Reset_Write(1);
}

static int gyro = GYRO_SET;
```

```
static unsigned int range = 45000;
static int flag = GYRO_SEL;
CY_ISR(analog_isr)
{
    if (flag == RANGE_SEL) {
        if (Detection_Status_Read()) {
            range = ADC_GetResult16();
        } else range = 0;
    } else {
        int gyro_tst = ADC_GetResult16();
        gyro = (gyro_tst > 23000) ? ((gyro_tst < 26000) ?
          ↪ gyro_tst : gyro) : gyro;
    }
}

static int startDelay = 600;
CY_ISR(gyro_isr)
{
    if (startDelay > 0) startDelay--;
    flag = GYRO_SEL;
}

CY_ISR(range_isr)
{
    flag = RANGE_SEL;
}

void main()
{
    int finished = 0;
    char tstr[17] = {0};
    unsigned int range_end = 56000;

    /* Loop Vars */
    float gyro_sum = 0.0f;
    float pan_prev = 0.0f;
    float range_prev = 0.0f;

    /* Start LCD Screen */
    LCD_Start();

    /* Start PWM Signals */
```

```
DriveTrain_Start();

/* Start Counters */
PanTime_Start();
TiltTime_Start();
AMux_Sel_Start();
ADC_Start();
ADC_StartConvert();

/* Start Interrupts */
Pan_int_Start();
Pan_int_SetVector(pan_isr);
Tilt_int_Start();
Tilt_int_SetVector(tilt_isr);
Gyro_int_Start();
Gyro_int_SetVector(gyro_isr);
Range_int_Start();
Range_int_SetVector(range_isr);
Analog_int_Start();
Analog_int_SetVector(analog_isr);

CyGlobalIntEnable;
while(finished < 2)
{
    float pan_err, range_err, gyro_err;
    float gyro_p, gyro_i;
    float pan_p, pan_d;
    float range_p, range_d;

    sprintf(tstr, "Tilt: %5d", tilt);

    LCD_Position(0, 0);
    LCD_PrintString(tstr);

    pan_err = pan - PAN_SET;
    range_err = range - RANGE_SET;
    gyro_err = gyro - GYRO_SET;
    if ((gyro_err < GYRO_DEADBAND) && (gyro_err > -
       ↪ GYRO_DEADBAND))
        gyro_err = 0;

    /* Gyro PID Loop */
```

```
        gyro_p = 0.3 * gyro_err;

        gyro_sum += gyro_err;
        /* Clamp Sum */
        if (gyro_sum > 10000) gyro_sum = 20000;
        else if (gyro_sum < -10000) gyro_sum = -20000;
        gyro_i = 0.00 * gyro_sum;

        /* Pan PID Loop */
        pan_p = 11.0f * pan_err;
        pan_d = 3.0f *(pan_err - pan_prev);
        pan_prev = pan_err;

        /* Tilt PD Loop */
        range_p = 3.0f * (range_err / 30.0f);
        range_d = 2.5f *(range_err - range_prev);
        range_prev = range_err;

        /* Don't do dangerous things before start delay */
        if (startDelay <= 0) {
            if(Detection_Status_Read()) {
                    if (tilt < -150) {
                        DriveTrain_Stop();
                        LCD_Position(1, 0);
                        LCD_PrintString("Stopped.....");
                    } else {
                        DriveTrain_Set((int)(pan_p + pan_d), -(
                            ↪ int)(range_p + range_d), (int)(
                            ↪ gyro_p + gyro_i));
                        LCD_Position(1, 0);
                        LCD_PrintString("Running.....");
                    }
            } else {
                DriveTrain_Stop();
                LCD_Position(1, 0);
                LCD_PrintString("Stopped.....");
            }
        }
    }

    DriveTrain_Stop();
```

```
    for(;;){}
}

/* [] END OF FILE */
```