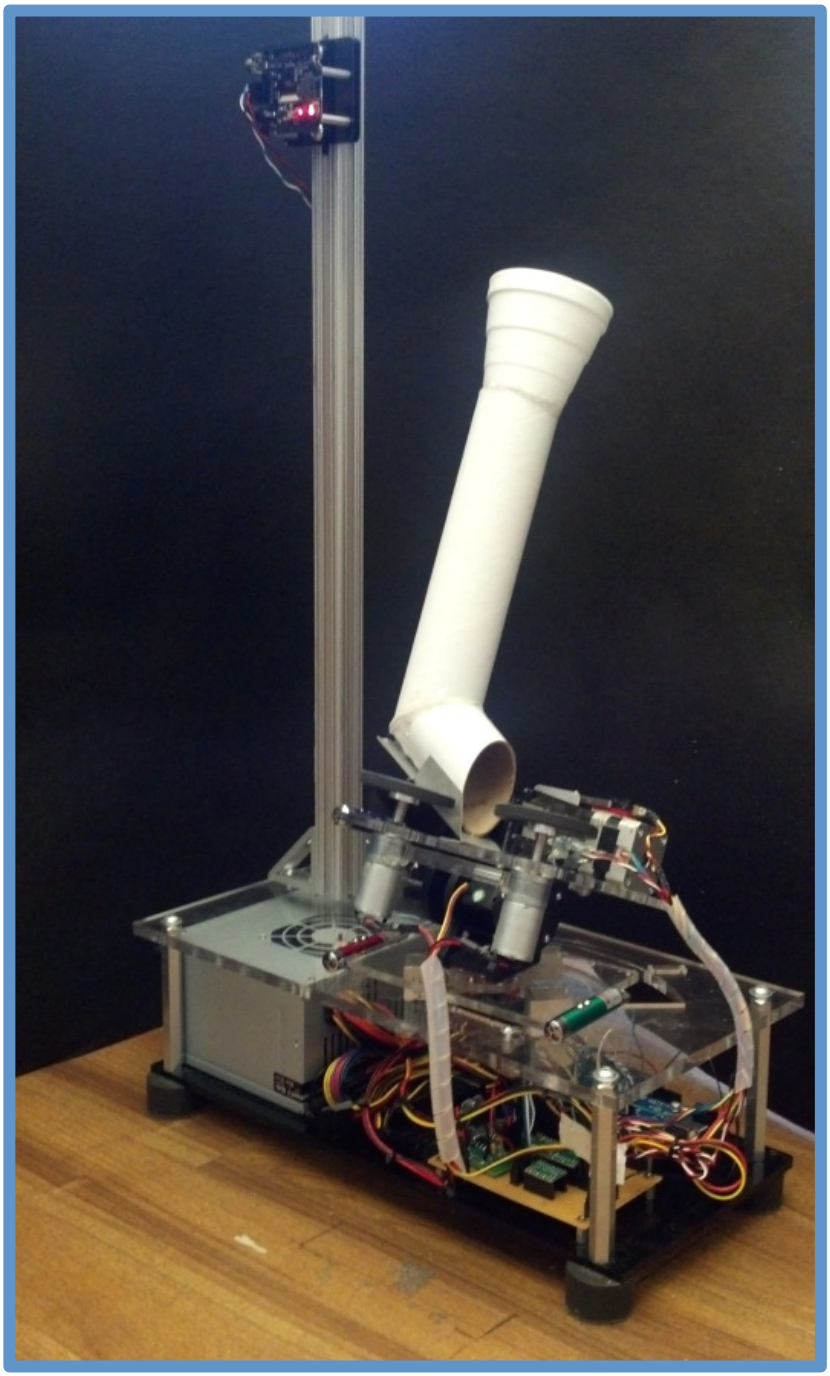
Final Report – Task 18

Team F: Richard Musgrave, Wesley Myers, SooHyun Jang, David Jia
18-578 Mechatronic Design
Spring 2012



Abstract

The overall goal of the design is to be able to engage eight inch wide stationary and moving targets with Nerf® Ballistic Balls from a distance of ten feet. To accomplish this goal, the team plans to design a mechatronic device, incorporating a vision system to detect the targets' location, a microcontroller for system control, and mechanisms for positioning and launching the balls toward the targets. The system in this work employed a dual flywheel shooting system with a linear feed mechanism and stacked hopper system. To aim at individual targets, the system used a PCB mounted camera for vision processing and had active control of panning (yaw) and tilting (pitch). All motors and sensors in the system are controlled by an Arduino microcontroller and has feedback on the important parts of the system. This project was designed for fulfillment of requirements for Mechatronics, a Carnegie Mellon capstone design course.

Table of Contents

Overview

Figure 1 - Soccer Ball Shooter Mechanism

The goal of this system was to accurately fire multiple shots at various targets from a specified distance. Our system accomplishes these goals by using a soccer ball style shooting system as shown in Figure 1. This basic design allowed us to greatly reduce the size of our system which you can see is much smaller than the other systems that have been designed. The overarching concept for our system was to create the simplest machine. This can be seen by our original design not having an active pitch degree of control, but later implementing it to be able to reach all targets on the board. Furthermore, the system was designed with a very small footprint making it very easy to move around the area as seen in Figure 2.

Our system was adapted over time to meet the requirements for the project. For instance, we had to completely redesign our system and implement active tilting so that we could hit all of the targets. This change was necessary and allowed us to adapt and learn over the semester. The project allowed everyone on the team to gain an understanding and work with mechanical, electrical and software systems in combination with each other. We found the project very rewarding and learned a lot throughout the semester.

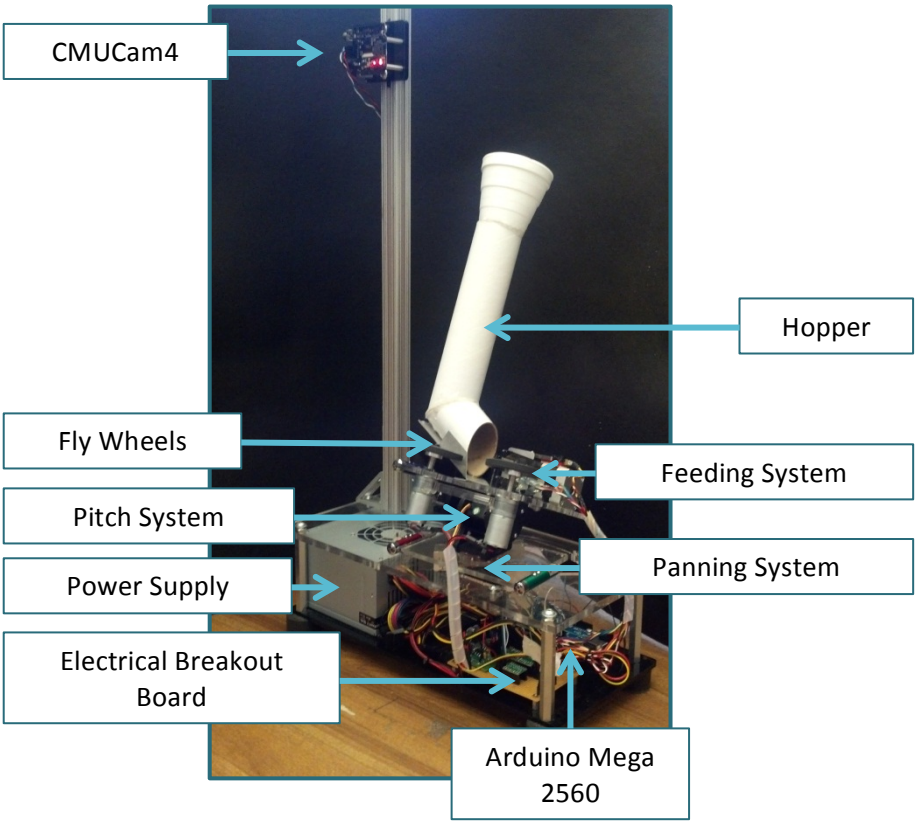


Figure 2 - System Overview

Design Requirements

Between the specifications of the class and our own analysis below is a list of requirements that we identified for the project below in Table 1. The description is what the requirement is, the type is either mechanical, electrical or software and the priority is how important it is relative to the overall system goals.

Table 1 - Design Requirements

Item	Description	Type	Priority
1	Shoot 8” targets in a predefined 3’ tall by 4’ wide area	Mechanical	High
2	Shoot static targets 6x under 20 sec	Mechanical	High
3	Shoot dynamic targets 6x under 40 sec	Mechanical	High
4	Use an embedded image processing system	Electrical	High
5	Ability to track multiple targets at once	Software	High
6	Sense if there is a ball to feed	Electrical	Medium
7	System fully embedded	All	High
8	Panning & tilting system resolution $\leq 1°$	Mechanical	High
9	Shoot targets from 10’ away	Mechanical	High

Design Concept

To meet the aforementioned specifications, our team planned to design a mechatronic device, incorporating a vision system to detect the targets’ location, a microcontroller for system control, and mechanisms for positioning and launching the balls toward the targets.

Our original concept seen in Figure 3 shows our planned system layout. A football launcher style mechanism was selected to shoot the balls. Two wheels rotating in different directions would grip and project the balls forward. One original consideration was to use one motor to control both wheels, ensuring that they would be spinning at the same speed. However, the complexity of the required gearing made this idea undesirable.

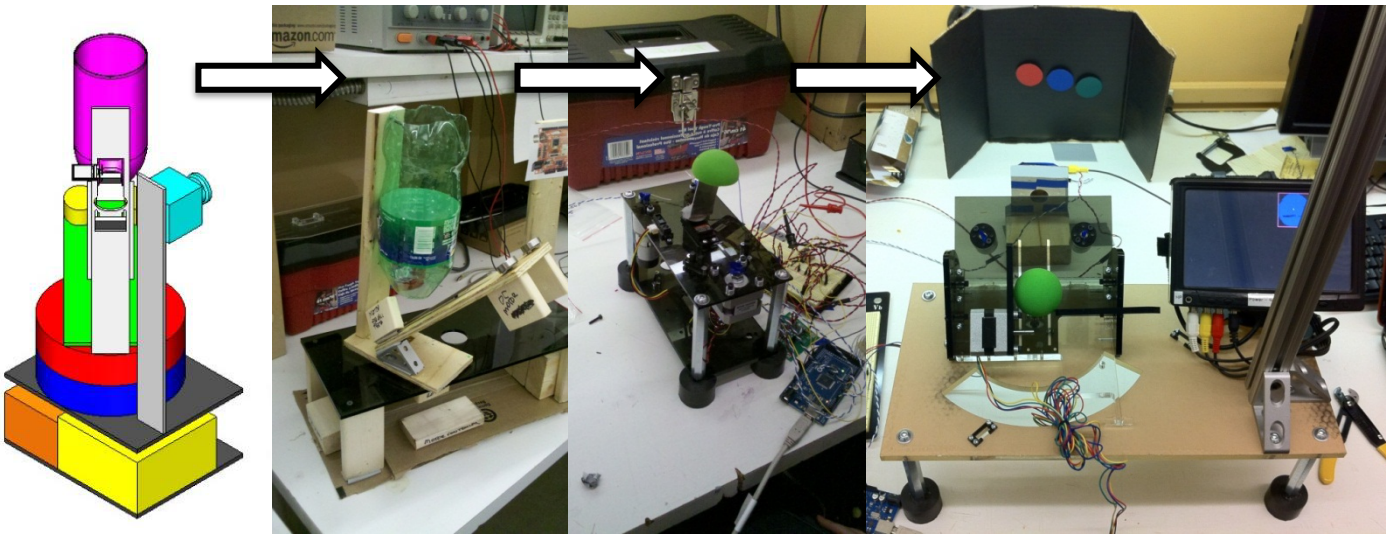


Figure 3 - Early Prototypes and Revisions of System

To enable the shooting mechanism to reach the edges of the shooting range, the shooter mechanism is attached to the panning platform, which is actively controlled by software. Most electronics, such as the microcontroller and power supply, were placed under this platform to keep all the electronics together. The shooting mechanism was also capable of active pitch control, which was required in order to shoot at all three targets, since the wheel speeds could not vary enough to shoot at all three targets.

A tube was used as the hopper subsystem because the original hopper system was too heavy for the panning actuator to handle if mounted on top of the shooting platform. Balls would fall through our hopper system into our rack and pinion feeding system—both of which are on the shooting platform—and the rack and pinion would push the balls towards the wheels in order to shoot. Our system could also actively detect whether a ball placed on the rack and pinion and was ready to shoot or not. By placing an infrared sensor on the rack and pinion mechanism in the Ball Feeder Subsystem, the system could recognize that there was no ball in the feeder and that the ball was probably stuck in the hopper system. The system would then shake back and forth until a ball rolled out. Once we knew the ball was in place, we pushed the ball into the flywheel and fired the ball.

There were several options to assess for the device's vision system, either using a camera or an infrared system based on the Wii-mote built-in infrared sensor. We chose the CMUcam4 given our knowledge that it was highly interface-able with the Arduino microcontroller. It is highly suitable for tracking stationary and moving color blobs, and vision rarely became a problem for our system.

Most of the design considerations made for this system were rationalized by the goal of creating a simplistic design. We believe that this is a critical reason for all of the successful aspects of our project. We chose things that we knew would work and were low risk. It is dangerous to try to be ambitious and choose something that you cannot complete in a semester. An example of something bolder was using a stepper motor for our panning mechanism. The issue is that this added an extra layer of complexity in our system since we need infrared limit switches to calibrate the position and a motor driver to precisely control the motor. To simplify our system, we switched to a servo, making the panning subsystem much simpler. Figure 4 shows a depiction of the system before the change to a servo was made.

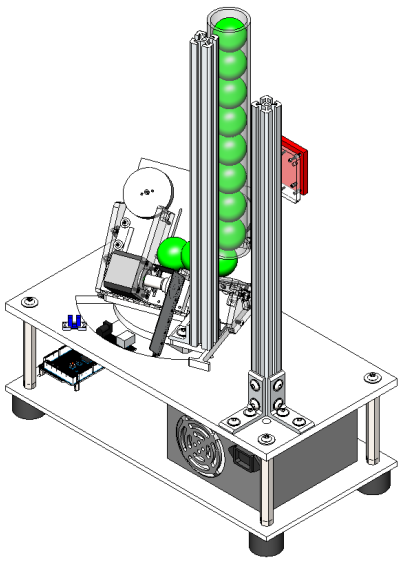


Figure 4 - CAD Model of System Before Active Pitch was Implemented

The only innovation we went after was ball detection. We wanted to make sure we were aware that there was a ball ready to be shot. This was accomplished with using an infrared sensor in the Ball Feeder Subsystem. This requirement was not in the original design specification and we were the only team that did this.

Functional Architecture

The goal of this architecture was to get a ball, locate a target, and shoot at the target. Figure 5 shows a diagram showing the functional architecture of our system at the subsystem level.

Essentially the Hopper Subsystem got the ball into place. Next, the CMUcam4 gathers data about the target's location and behavior. This information is passed to a look up table to tell the Pan/Tilt Subsystem how to aim. Next the Flywheel subsystem activates and drives the motors to shooting speed. Next, the Ball Feeding Subsystem drives the ball into the flywheels and fires the ball at the target. For more details on the subsystems and how they interact, please see the Subsystem Description section.

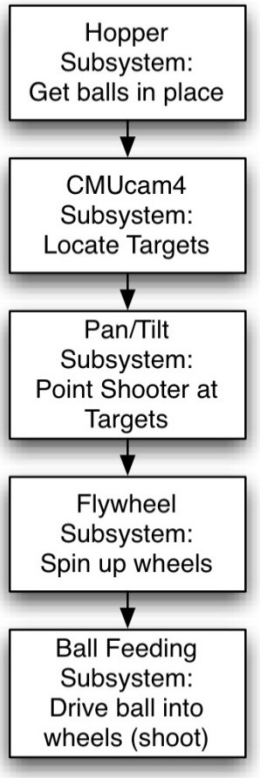


Figure 5 - Functional Architecture

Physical Architecture

Figure 6 shows our physical architecture. The arrows illustrate input and output lines on the Arduino. Numbers on the arrows indicate how many lines are going from the Arduino to the device.

This architecture is designed to solve our task. The only sensor that needed to

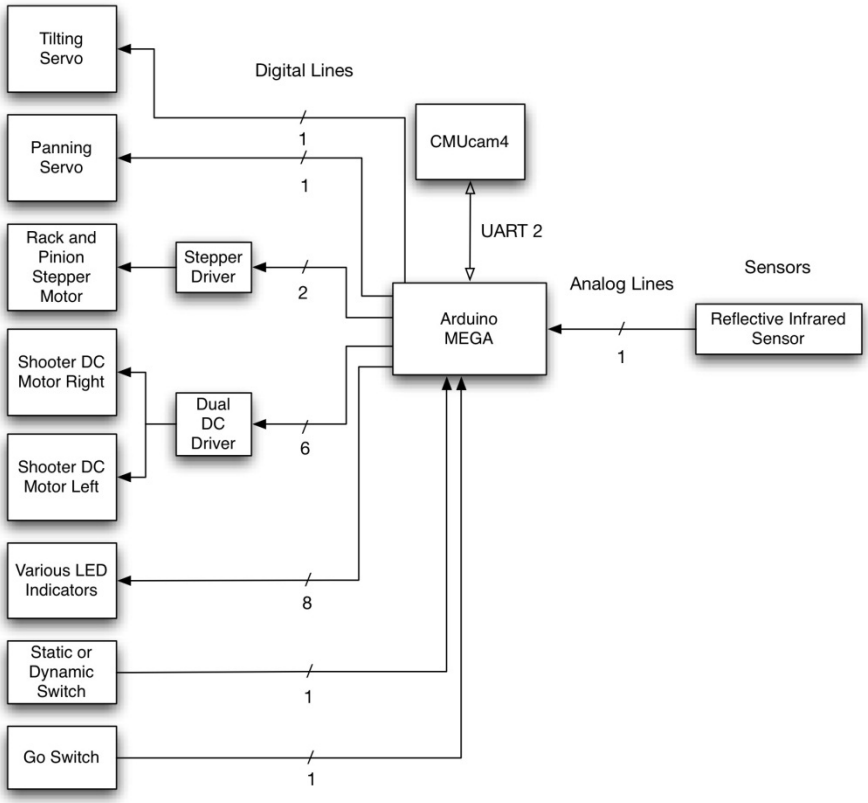


Figure 6 - Physical Architecture

be analog was our infrared reflective sensor. With this, we were able to get a wide range of possible values to indicate whether a ball was there or not. We could not work with a binary response because the ball is never quite in the same position.

Most of our pins ended up being digital lines. Input lines were switches to tell the system what to do. The rest were output lines to tell devices what to do.

The CMUcam4 was the only device that needed serial communication. We put it on UART 2 such that we could still print to the terminal.

Components in Relation to Subsystems

- Camera Subsystem: CMUcam4
- Pan/Tilt Subsystem: Tilting Servo and Panning Servo
- Ball Feeder Subsystem: Reflective Infrared Sensor, Rack and Pinion Stepper Motor
- Flywheel Subsystem: Shooter DC Motor Right, Shooter DC Motor Left

Subsystem Architecture

Below is a description of each of the subsystems that make up our system. The location and placement of these systems can be referenced in Figure 2.

Figure 7 - Window Division using CMUCam4

Vision

The CMUcam4 provides simple vision capabilities to small embedded systems in the form of an intelligent sensor. The CMUcam4 is an open source programmable embedded color vision sensors are low-cost, low-power sensors for mobile robots. We used the CMUcam4 for on-board, real-time vision processing tasks.

The CMUcam4 is connected to the Arduino MEGA serially over a UART port. With this, we can send commands to the CMUcam4 and receive tracking data back.

The strategy we used for this was very simple. We knew the relative y-coordinate position of the targets relative to our position. We also knew the maximum and minimum x-coordinate position of the targets relative to our position. So we constructed a window around the possible position of each target as shown in Figure 7. What this did was eliminate any outside noise. What this also allowed was the ability to really target the red, green, and blue models against the black background. We could accept a wide range of color values in light and darkness.

The data we got back from the CMUcam4 was the (x,y) position relative to the camera's view. We then passed this data to a look up table to find the servo positions we needed to aim the shooter.

Hopper and Feeding

This subsystem ended up being rather simple, but not to specification. We had originally designed a hopper that you could dump balls into a carousel that would feed the balls. The hopper caused our system to become too heavy and unbalanced, thus causing the servo to not be able to turn our shooter in the correct direction.

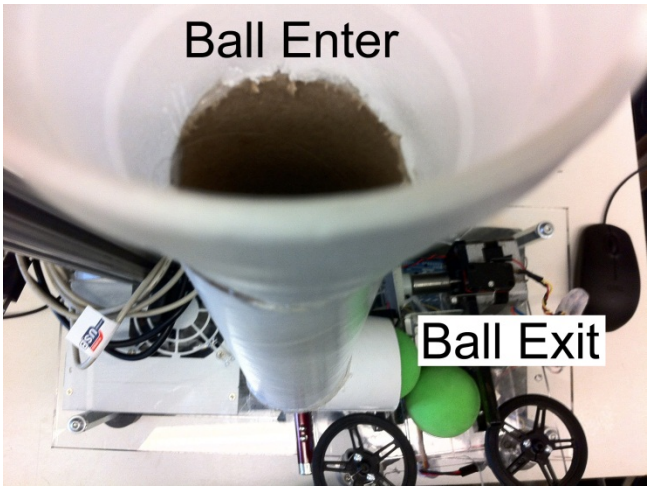


Figure 8 - Hopper Entrance and Exit

So the final feeding subsystem ended up being a cardboard tube with a Styrofoam cup attached at the top as seen in Figure 8. The cup was the right width such that no two balls could get stuck at entry, so this sufficed.

Panning & Tilting

This subsystem is designed for aiming at our shooter. Our original design called for only a panning motion since we felt our DC motors

could vary the speed enough to reach all three targets. However, initial testing showed we could only reach one of the three. So this called for a redesign to add a tilting mechanism.

We went with two servos for simplicity's sake as shown in Figure 9. This allowed fast and accurate control of the shooter to aim exactly where we wanted it to every time. We didn't have to rely on a stepper motor or an encoder to keep track of our position. We get the position of each servo from the look up table values we get from the (x,y) blob position.

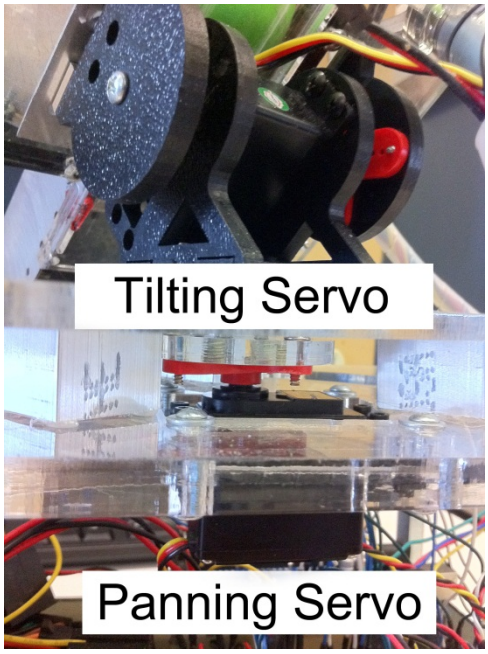


Figure 9 - Tilting and Panning Servo Systems

Shooting

This subsystem's primary goal was to launch the Nerf balls. The flywheel design is a tried and true concept for flinging balls. This system is used for launching baseballs, footballs, and even soccer balls, so why not Nerf balls? The flywheels keep at a constant speed for all shooting positions. This subsystem is depicted in Figure 11.

Ball Feeding

This subsystem is simply a rack and pinion. The subsystem is actuated by a stepper motor and two sensors govern its motion: a limit switch and an infrared reflective sensor. On startup, the rack and pinion goes to a start position governed by the limit switch. When it is time to drive the ball forward, the IR sensor says whether or not there is a ball. If not, then the pan/tilt subsystem gyrates to get the ball into position. The ball gets pushed towards the flywheel subsystem, but halts right before it. This is because the next ball in the hopper nudges the ball slightly, causing it to be off center right before getting into the flywheels. So we stop for a quarter of a second to let the ball settle down, and then we push it into the flywheels and then we retract to start position. This system is depicted in Figure 10.

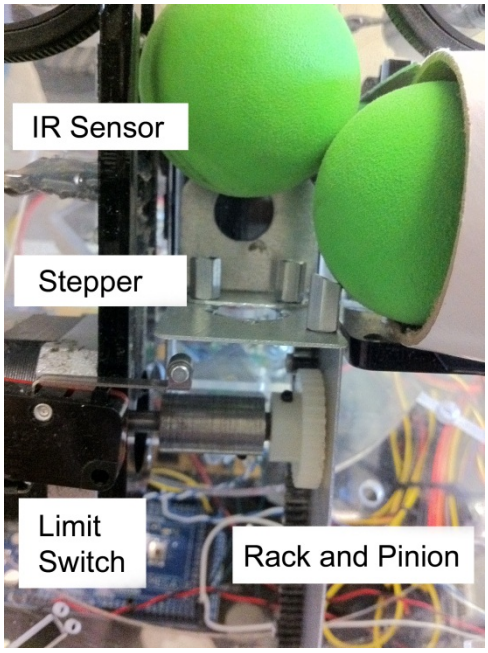


Figure 10 - Ball Feeding System

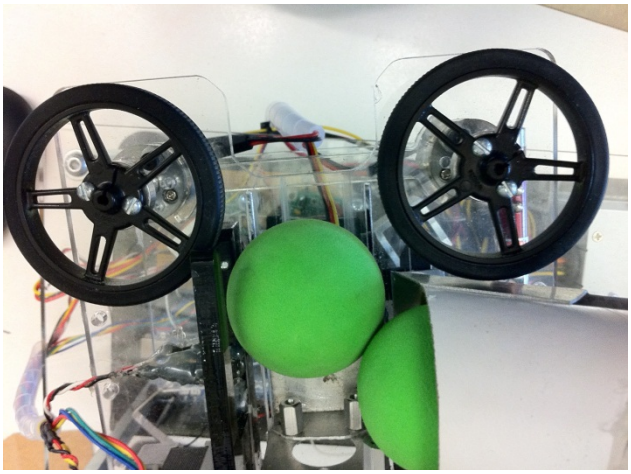


Figure 11 - Flywheels Mounted Directly on Output Shafts of Motors

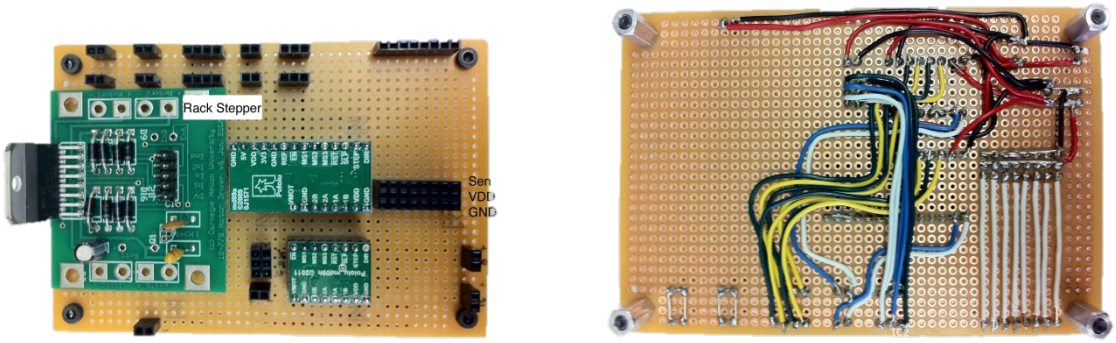


Figure 13 - Custom Motor Driver and Sensor Board

Motor Driver and Sensor Board

Wesley Myers created and designed a custom motor driver and sensor board for the system as shown in Figure 13. It allowed easy swapping of stepper or DC motor drivers. It also provided a sensor 5V rail for sensors, with the capability of supporting 8 sensors. The top part of the board was designed to be right alongside of the Arduino MEGA and have a short jumper wire between the MEGA and the board. This made the wiring much cleaner. Plus the electrical system looked much more elegant with this.

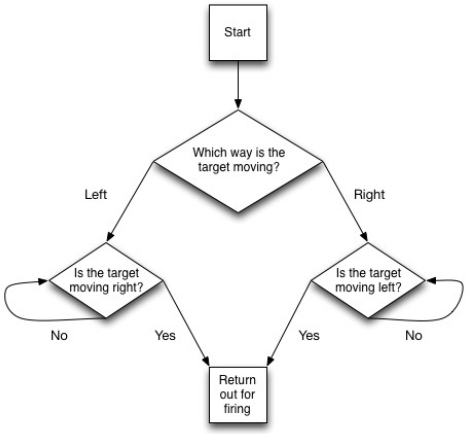


Figure 12 - Dynamic Target Algorithm

Pseudo Code

Figure 12 shows the pseudo code of the public demonstration competition dynamic target code. This basically describes how the code flows for our system. We wait for a button to tell us to start, then we collect target data until we find a moving target. We then go to a look up table to tell us where to put the servos. Next we move the servos and then drive the rack and pinion to shoot the ball at the target. We check to see if we can still

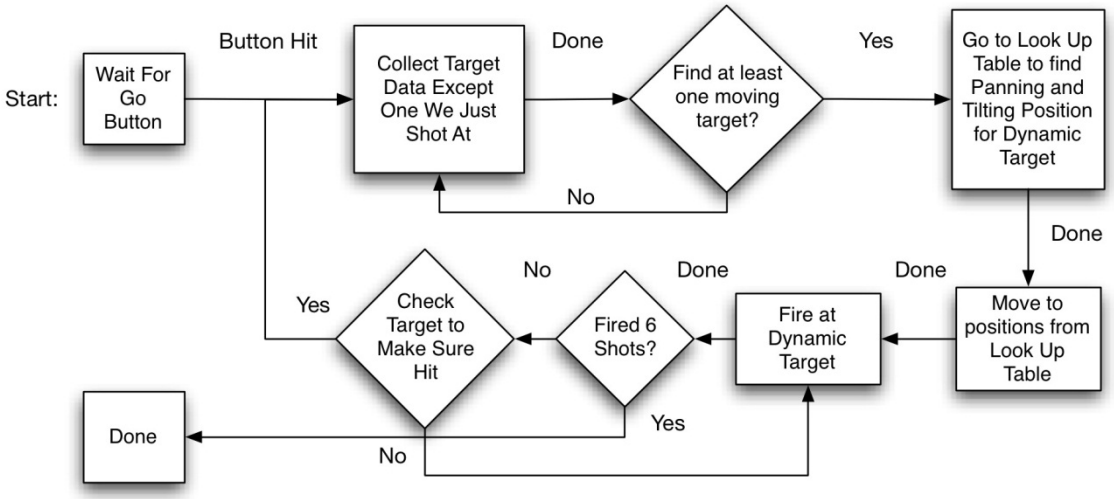


Figure 14 - Pseudo Code of Dynamic Firing System

shoot, and then check to see if we hit the target. We know if we hit the target because the dynamic target will become stationary. If we hit it, then collect target data on the other two targets to figure out which one is now moving and continue.

Figure 12 is pseudo code from tracking the dynamic target. This is a very simple algorithm that keeps track of which way the target it moving and when it changes direction. Once it changes direction, we know we have roughly two seconds to make contact with the target.

System Modeling, Development and Performance

System Modeling

The first thing we did was create a mock up of our system to get a physical idea of what our system would look like. This is shown in the left picture of Figure 3 - Early Prototypes and Revisions of System. From there we went to a full CAD model shown in Figure 4 - CAD Model of System Before Active Pitch was Implemented We did some stress analysis on the acrylic plate above the electronics. This was important since we needed to check to make sure that our structure could be held up and the acrylic wouldn't warp under the stress. Another important calculation we did was the RPM required for our motors in order to fire the ball at the target. This helped us choose the motors we would need for our system.

The mechanical and electrical system integration wasn't too difficult. Our system was designed to be very simple. The next step was to integrate a finite state machine into the system. The state machine lab gave us a good foundation to start on. A large portion of the code was the CMUcam4 library, required to operate the CMUcam4. Integrating the code with the mechanical system was only a matter of sitting down, coding, testing, and repeating.

Development

The development of our system went very well over the semester. What really held us back was our consistency with shooting. The main issue was that we had large error on shooting. Sometimes we'd hit dead center, other times we'd be off by a foot, and other times the ball would go in a random direction. This took many weeks to debug. The first thing we noticed was that the ball would roll around when we pushed the ball into the wheels. This was solved with making a two-point cradle to hold the ball. Another thing we noticed was that the ball rubbed slightly on the feeding wall, thus causing the ball to rotate. This caused the ball to rotate as it was being fed into flywheels. The solution was to cut a slit into the wall so that the ball didn't rub against it. Another example of an iteration was that we noticed that the ball sometimes nose dived into the ground. We realized the issue was that the ball was getting topspin from rubbing onto the acrylic after the flywheels. We cut away the acrylic from the middle of the flywheels to the edge of the acrylic. This eliminated the problem for us. These were just a few examples of issues with this that we solved. The thing we learned from this is that iterations take time and we were happy that we got to the iteration phase weeks before demonstration time.

Performance

At our best, our system could shoot 80% for static targets and 60% for dynamic targets. We still had a +/- 6 inch radial deviation on our shooting. We were able to eliminate the issue of the ball moving before going into the flywheels. What we attribute the error to is our DC motors. They are very low torque motors, which means that the wheels slow down a significant amount instead of maintaining their speed as the balls go through the wheels.

One issue we had was that we didn't have a hopper for our system. We had designed and fabricated one, however when we added it to our system, we found that it caused the shooter to be unbalanced and too heavy. The panning servo didn't have enough torque to power the shooter to point in the correct direction due to the additional weight. We didn't have the money in our budget to buy the high torque servo we needed, so we decided we needed to forgo the hopper.

We didn't have any coding issues, except we did have a hiccup during our final system demonstration. This was due to the fact we made some code modifications the night before, and we couldn't thoroughly test the code. Luckily it was a simple fix. Our code was modular enough to quickly switch to the Public Demonstration Competition style.

Conclusions

From the start of this project, our system's design was meant to be simplistic and straight forward to implement. Our two-wheel shooting strategy has been used commercially in various ball shooting designs. From a mechanical perspective, there were not many components that were very difficult to assemble, since many parts of our system were laser cut from acrylic. This also allowed for quick redesigning and reimplementation of a component. We believe that we created a sleek mechanical design and chose a suitable vision system and microcontroller that regulated the functionality required by specifications.

However, despite this, the final performance of our system did not meet our expectations. Although most of the subsystems functioned fine on their own, there was too much overall error in our system, resulting in a mediocre accuracy during the public demonstration. During the demonstration, our average static target accuracy was about 50%, and our average dynamic target accuracy was about 33%. In a redesign of the system, there would be an important aspect that we would change.

We believe that a large portion of our error could be attributed to a sub-optimal motor selection for the DC motors that powered the fly wheel shooting mechanism. These motors were found to not spin at the same speed, causing all the ball's trajectories to curve to the left, and thus, when calibrating the panning angle look-up table, we always pointed the shooter to the right of the target. Furthermore, without any feedback loop for these motors, we could only assume that the ball would curve the exact same amount from day to day. In addition, the length of time these DC motors were powered on for seemed to affect the system's accuracy as well. For instance, the system would shoot higher than usual if left on for too long. To this end, we found that calibration of the look-up tables were only meaningful if the system had not been powered on for an extensive period of time. Finally, due to the low torque capabilities of these

motors, they would actually slow down when a ball was being pushed through between them, creating another source of error. The unreliability of these motors generated an error in accuracy that was considerably wider that the targets we were trying to strike. So, having a closed-loop feedback system for the DC motors and selecting motors with higher torques would have considerably reduced our system's error.

Another motor selection issue involved the servo motor used to pan the system. This motor was unable to handle the additional weight of a complex hopper system, which had not been fully designed when this servo motor was selected. The servo was unable to consistently pan to specific positions when the hopper was mounted onto the shooter plate, and eventually, we had to drastically redesign our hopper system. If we had had enough money available in our budget, we should have bought a higher torque servo motor.

As these two anecdotes show, an important lesson we learned was to properly calculate the motor requirements for our system and to leave room for error in case designs were changed. Despite wanting to make a simple, low-costing device, functionality must be prioritized and should not be sacrificed for cheaper actuators.

Lessons Learned for Future Work

If this project were to be designed for a start-up company, a specification that we would impose would be for the device to shoot at the targets for any position. All the robots made for this project could only accurately shoot if they were positioned very precisely in front of the targets. If this project were to be commercialized, this seems like a sensible requirement. For this robot to be practical, it must also be able to track targets of differing shapes and a larger variety of different colors—not just red, green and blue. In addition, we would make the size requirement stricter and enforce that the entire device be completely embedded. Shooting at these targets from such short distances is a completely solved problem in the real world and should not require a complex, bulky device to do so.

Our design for this new specification would be mechanically similar to our current design, except, of course, for certain motor selections, as previously mentioned. To be able to shoot successfully at the targets from varying angles and distances to the target, our device would need to perform a considerable amount of calculations, since hardcoding look-up tables would not work. Also, more complex computer vision software would be needed to track different shapes and more different colors. Thus, we would most likely require a microcontroller with higher processing capabilities, such as a Beagleboard. This project would be more software intensive, but would upgrade our current system to a much more versatile robot.

Parts and Budget Estimate

The team ran into several broken components throughout the semester that pushed us to limits of our budget, but overall we were very close to the goal budget as seen in the following table.

Item	Supplier	Part / Material	Type / Item # / Description	Quantity	Unit Price	Total
1	Sparkfun	Arduino Mega 2560 R3	Microcontroller	2	$ 58.95	$ 117.90
2	Pololu	A4988	Stepper Motor Driver	3	$ 19.95	$ 59.85
3	Pololu	1570	DC Motor	2	$ 21.95	$ 43.90
4	N/a	Hitec 485HB	Servo	2	$ 16.99	$ 33.98
5	McMaster	MMC 2572K25	RUBBER PAD, 1/4-20-1/2	4	$ 7.21	$ 28.84
6	ServoCity	DDT500H	Direct Drive Tilt System	1	$ 24.99	$ 24.99
7	Enco	327-6740	Acrylic Sheet, .375x12x24	1	$ 24.58	$ 24.58
8	McMaster	MMC 92230A350	1/4-20-4 HEX SPACER	5	$ 4.87	$ 24.35
9	Amazon	Diablotek DA Series 400-Watt ATX Power Supply PSDA400	Power supply	1	$ 19.99	$ 19.99
10	Pololu	#1420	Wheel 50x8	2	$ 7.95	$ 15.90
11	Pololu	# 1203	Aluminum Hub	2	$ 7.49	$ 14.98
12	McMaster	MMC 91375A101	Set Screws	1	$ 5.90	$ 5.90
13	McMaster	95947A039	Spacer, M3-12mm	8	$ 0.61	$ 4.88
14	McMaster	MMC 57655K62 MODIFIED	RACK	1	$ 4.63	$ 4.63
15	McMaster	MMC 57655K42	DRIVE GEAR, RACK & PINION	1	$ 2.68	$ 2.68
16	Sparkfun	COM-09340	White Arcade Button	1	$ 1.95	$ 1.95
17	Lab Parts	N/a	Stepper Motor 2ozin, 200 step/rev	1	$ 14.95	$ 14.95
18	Lab Parts	N/a	Misc. Electronics Parts (wires, connectors, etc)	1	$ 35.00	$ 35.00
19	Lab Parts	N/a	Stepper Motor Driver	1	$ 11.65	$ 11.65
20	N/a	N/a	Credit Microcontroller	1	$ (50.00)	$ (50.00)
21	N/a	N/a	Credit Motors	1	$ (60.00)	$ (60.00)
						$ 380.90

Appendix A – Competition Code

```
/* Mechatronic Design Competition Code */

#include <CMUCam4.h>
#include <Time.h>

#include <Servo.h>
#include <stdio.h>
#include <CMUcam4.h>

void create_table();
void cmucamSetup();
int getCmucamData(int, int, boolean, boolean, boolean);
void resetTrackingData();

/************************PINS***********************/
// BALL SENSOR
#define ball_Sensor  2
#define sensor_limit 800

// FLIP SWITCH
#define switch_S  50              // Static switch; right
#define switch_D  52              // Dynamic switch; left

// HORIZONTAL PANNING ACTUATOR
Servo panS;
#define pan_servo_pin             22
#define PAN_CENTER                98

// PITCH ACTUATOR
Servo pitchS;
#define pitch_servo_pin           11
#define PITCH_CENTER              110
#define PITCH_LEVEL               93;

// RACK AND PINIONs
#define dir_pin_rack  13   // Rack and pinion stepper direction
#define step_pin_rack 12   // Rack and pinion stepper motor
#define FEED_COUNT    150 //125      // Step count for rack

// SHOOTER
#define left_dc_in1   5    // DC motor left input 1
#define left_dc_en    6    // DC motor left enable
#define left_dc_in2   7    // DC motor left input 2
#define right_dc_in1  2    // DC motor right input 1
#define right_dc_en   3    // DC motor right enable
#define right_dc_in2  4    // DC motor right input 2

// START/HALT SWITCH INTERRUPT
#define switchInput   18

// POSITION+SHOOT SWITCH
#define switch_pin    53   // Limit switch

// RACK AND PINION LIMITER
#define rack_limit    51   // Limit switch

/***********CMUCAM4 CONSTANTS/VARIABLES**********/
int x_blob; // X-coordinate returned by CMUcam4 (range, min and
max defined below)
int x_blob_10;   // For temporary input
int x_blob_1;    // For temporary input
#define camera_pixel_range 65 // Range of pixels on camera
#define camera_pixel_max   95 // Camera pixel range max
#define camera_pixel_min   35 // Camera pixel range min

#define LED_PIN 20
#define CYCLES_MAX 5

CMUcam4 cam(2);                   // Use port 2 to allow Serial
prints

int color;

#define RED (1)
#define GREEN (1 << 1)
#define BLUE (1 << 2)

#define redSwitch   3
#define greenSwitch 4
#define blueSwitch  5

#define LEFT_X 30
#define RIGHT_X 105  //prev 100
#define TOP_Y 30  //prev 30
#define MID1_Y (TOP_Y + 20)
#define MID2_Y (TOP_Y + 40)
#define BOT_Y (TOP_Y + 60)

int cmucamState;

int xData[30];
int yData[30];

struct circle {
  boolean moving;
  int xPos;
};

circle redTarget;
circle greenTarget;
circle blueTarget;

int targetTrackingColor = -1;

// Target RGB ranges
#define RED_RH 255
#define RED_RL 180
#define RED_GH 180
#define RED_GL 50
#define RED_BH 200
#define RED_BL 80
```

```
#define GREEN_RH 150
#define GREEN_RL 0
#define GREEN_GH 210
#define GREEN_GL 120
#define GREEN_BH 180
#define GREEN_BL 80

#define BLUE_RH 180
#define BLUE_RL 0
#define BLUE_GH 130
#define BLUE_GL 60
#define BLUE_BH 255
#define BLUE_BL 140

/*********SHOOTING PARAMETERS*****\***/
int dynamic_test = 0;              // 1 - dynamic, 0 - static
#define num_run_stat  1            // # of hits on static targets
#define num_run_dyna  1            // # of hits on dynamic targets
int runs;                          // Temporary variable
int to_moving_state;               // dynamic indicator for reset

// Variable indicating target to be shot
int target_to_shoot = -1;

/************DEBUG PRINTING*****************/
#define DEBUG
#ifdef DEBUG
#define DEBUG_PRINTLN(...) Serial.println( __VA_ARGS__ );
#else
#define DEBUG_PRINTLN(...) do {} while(false);
#endif

#ifdef DEBUG
#define DEBUG_PRINT(...) Serial.print( __VA_ARGS__ );
#else
#define DEBUG_PRINT(...) do {} while(false);
#endif

// Enables push button, otherwise runs automatically
//#define ENABLE_BUTTON

/************STATE MACHINE******************/
enum STATES {halt, start, getData, calcPos, getDir, feed, turn,
push, shoot, shootMoving, reset};
STATES state;

/******************INTERRUPT******************/
static unsigned long interrupt_prev = 0;
int debounce_flag = 1;   // debounce = 1: when enter interrupt,
start debounce
                // debounce = 0: debouncing...
//unsigned long debounce = 200;  // ms

/***************OTHER VARIABLES*************/
int switchHit = 0;        // 1 = start, 0 halt
int interrupt_call = 0;  // Variable for interrupt
int counter;              // Temp variable; counter for stepper
int counter_rack;       // Temp variable; for rack and pinion
int dir;                  // Temporary variable; direction for stepper
int i;                    // For loop variable

int target_position;    // Target (panning) position
int target_pitch;       // Target (pitching) position
int target;               // Target variables

/****LOOK UP TABLE CONSTANTS/VARIABLES FOR MOTORS***/
#define resolution      1          // Resolution in pixels
#define num_loc         (round(camera_pixel_range/resolution)+1)
// # of locations per target
#define num_target      3          // Number of targets

// DC motor speed
int motor_speed;

// Servo motor panning angle table
int servo_table[num_target][num_loc+2];
int target_vert;                   // Look up table index
int target_hori;                   // Look up table index

// Servo motor pitch angle table
int pitch_table[num_target][5];
int pitch_hori;                                  // Look up table index

// Other servo variables
int current_pan = 0;
int new_pan = 0;
int dir_pan = 0;
void move_pan();
void move_pitch();

// Variable for counting number of balls shot
int count_shot = 0;

/************SETUP PINS***************/
void setup(){
 DEBUG_PRINTLN("Setuping up");
 // Interrupt
 pinMode(switch_pin, INPUT);
 //attachInterrupt(5, switchInterrupt, RISING); // int 5 (pin 18)

 // DC motors
 pinMode(left_dc_in1, OUTPUT);
 pinMode(left_dc_in2, OUTPUT);
 pinMode(left_dc_en, OUTPUT);
 pinMode(right_dc_in1, OUTPUT);
 pinMode(right_dc_in2, OUTPUT);
 pinMode(right_dc_en, OUTPUT);

 // Rack and Pinion
 pinMode(dir_pin_rack, OUTPUT);
 pinMode(step_pin_rack, OUTPUT);

 // Pitching motor
 pitchS.attach(pitch_servo_pin);
```

```
 pitchS.write(PITCH_CENTER+4); // Point straight

 // Panning motor
 panS.attach(pan_servo_pin);
 panS.write(PAN_CENTER); // Go in middle

 // Create lookup table
 create_table();

 // Next state
 state = halt;
 Serial.begin(19200);

 // Set up CMUcam4
 cmucamSetup();
}

void loop(){
 switch(state) {
  /**************HALT STATE*********************/
  // Return state after shooting is complete. Waits for button
push if enabled
  case halt: {
   DEBUG_PRINTLN("State: Halt");

   // Disable DC motors not in use
   digitalWrite(left_dc_en, LOW);
   digitalWrite(right_dc_en, LOW);

   // Reset counter for rack and pinion
   counter_rack = 0;

   // Reset count for shot balls
   count_shot = 0;

   DEBUG_PRINTLN("Press button to start");
   // Wait for button
   #ifdef ENABLE_BUTTON
   while(digitalRead(switch_pin) == LOW){
    if(digitalRead(switch_pin) == HIGH){
     while(digitalRead(switch_pin) == HIGH);
     break;
    }
   }
   #endif

   // Next state
   state = start;

   break;
  }

  //*************START SYSTEM STATE*****************/
  case start: {
   DEBUG_PRINTLN("Start State");

   // Disable DC motors
   digitalWrite(left_dc_en, LOW);
   digitalWrite(right_dc_en, LOW);

   // Reset counter for rack and pinion
   counter_rack = 0;

   /* Read flip switch to determine dynamic or static mode, and
set number of runs per target. For this demonstration, it is 1 per
target for both modes. If the mode has been flipped (from last
   run), reset target state to allow full target tracking.  */
   if(digitalRead(switch_D)) {

    if(dynamic_test == 0){
     target_to_shoot = -1;
     resetTrackingData();
    }
    DEBUG_PRINTLN("DYNAMIC");
    dynamic_test = 1;
    runs = num_run_dyna;
   }
   else if (digitalRead(switch_S)){
    if(dynamic_test == 1){
     target_to_shoot = -1;
     resetTrackingData();
    }
    DEBUG_PRINTLN("STATIC");
    dynamic_test = 0;
    runs = num_run_stat;
   }

   // Reset CMUcam4 tracking data. Set all targets to "moving"
for static mode, and "not moving" for dynamic mode
   resetTrackingData();

   // Next state
   state = getData;
   break;
  }


  //************GET TARGET DATA STATE**********/
  // Get data for CMUcam4. Should track all targets only once per
demonstration (static or dynamic)
  case getData: {
   DEBUG_PRINTLN("Get Target Data State");

   #ifdef ENABLE_BUTTON
   DEBUG_PRINTLN("Please choose static/dynamic with switch,
then push button");
   // Wait for button to start tracking target
   while(digitalRead(switch_pin) == LOW){
    if(digitalRead(switch_pin) == HIGH) {
     while(digitalRead(switch_pin) == HIGH);
     break;
    }
   }
   #endif
```

```
  // 0, 1, 2 for each target (bottom to top)
  target_vert = 0;

  /* ------------- CMUCAM4 ----------------- */
  // Gets information from CMUcam4

  // If first time shooting for particular test, scan all targets
  if(target_to_shoot == -1) {
   if(dynamic_test){
    target_to_shoot = getCmucamData(3, 0, false, false, false);
    DEBUG_PRINT("(Dynamic) Target to shoot:");
    DEBUG_PRINTLN(target_to_shoot);
   }
   else{
    target_to_shoot = getCmucamData(4, 0, false, false, false);
    DEBUG_PRINT("(Static) Target to shoot:");
    DEBUG_PRINTLN(target_to_shoot);
   }
  }

  // Check if the aimed target was hit last time or not
  else {
   DEBUG_PRINTLN("Checking whether the target was hit...");

   // Check Red target
   if(target_to_shoot == RED) {
    DEBUG_PRINTLN("Current Target Red");
    if(dynamic_test == 1)
     getCmucamData(3, 0, false, true, true);
    else
     getCmucamData(4, 0, false, true, true);

    if(dynamic_test == 1 && !redTarget.moving){
     DEBUG_PRINTLN("Dynamic red hit");
     greenTarget.moving = true;
     greenTarget.xPos = 0;
     target_to_shoot = GREEN; // getCmucamData(3, 0, true,
false, false);
     DEBUG_PRINT("Next target: ");
     DEBUG_PRINTLN(target_to_shoot);
    }
    else if(dynamic_test == 1 && redTarget.moving){
     DEBUG_PRINTLN("Dynamic red miss");
    }
    else if(dynamic_test == 0 && redTarget.moving){
     DEBUG_PRINTLN("Static red hit");
     // Have to check where next static target is
     target_to_shoot = getCmucamData(4, 0, true, false, true);
     DEBUG_PRINT("Next target: ");
     DEBUG_PRINTLN(target_to_shoot);
    }
   }
   else if(dynamic_test == 0 && !redTarget.moving){
     DEBUG_PRINTLN("Static red miss");
   }

   // Check Green Target
   else if(target_to_shoot == GREEN) {
    DEBUG_PRINTLN("Current Target Green");
    if(dynamic_test == 1)
     getCmucamData(3, 0, true, false, true);
    else
     getCmucamData(4, 0, true, false, true);

    if(dynamic_test == 1 && !greenTarget.moving){
     DEBUG_PRINTLN("Dynamic green hit");
     blueTarget.moving = true;
     blueTarget.xPos = 0;
     target_to_shoot = BLUE; // getCmucamData(3, 0, false,
true, false);
     DEBUG_PRINT("Next target: ");
     DEBUG_PRINTLN(target_to_shoot);
    }
    else if(dynamic_test == 1 && greenTarget.moving){
     DEBUG_PRINTLN("Dynamic green miss");
    }
    else if(dynamic_test == 0 && greenTarget.moving){
     DEBUG_PRINTLN("Static green hit");
     // Have to check where next static target is
     target_to_shoot = getCmucamData(4, 0, true, true, false);
     DEBUG_PRINT("Next target: ");
     DEBUG_PRINTLN(target_to_shoot);
    }
    else if(dynamic_test == 0 && !greenTarget.moving){
     DEBUG_PRINTLN("Static green miss");
    }
   }

   // Check Blue Target
   else {
    DEBUG_PRINTLN("Current Target Blue");
    if(dynamic_test == 1)
     getCmucamData(3, 0, true, true, false);
    else
     getCmucamData(4, 0, true, true, false);

    if(dynamic_test == 1 && !blueTarget.moving){
     DEBUG_PRINTLN("Dynamic blue hit");
     redTarget.moving = true;
     redTarget.xPos = 0;
     target_to_shoot = RED; // getCmucamData(3, 0, false, false,
true);
     DEBUG_PRINT("Next target: ");
     DEBUG_PRINTLN(target_to_shoot);
    }
    else if(dynamic_test == 1 && blueTarget.moving){
     DEBUG_PRINTLN("Dynamic blue miss");
    }
    else if(dynamic_test == 0 && blueTarget.moving){
     DEBUG_PRINTLN("Static blue hit");
     // Have to check where next static target is
     target_to_shoot = getCmucamData(4, 0, false, true, true);
     DEBUG_PRINT("Next target: ");
```

```
      DEBUG_PRINTLN(target_to_shoot);
    }
    else if(dynamic_test == 0 && !blueTarget.moving){
      DEBUG_PRINTLN("Static blue miss");
    }
   }
  }

  DEBUG_PRINT("Next target to shoot: ");
  DEBUG_PRINTLN(target_to_shoot);

  // Next state
  state = calcPos;

  break;
}

//*********CALCULATE POSITION DATA STATE*******/
// Calculate lookup table indexes for panning and pitching
case calcPos: {
  DEBUG_PRINTLN("State: Calculate datas");

  // Next state. It will skip this next state if the target is dynamic
  state = getDir;

  // Reset number of runs per target
  if(dynamic_test){
   runs = num_run_dyna;
  }
  else {
   runs = num_run_stat;
  }

  // Get datas from CMUcam4. If dynamic, it will skip panning
  if(target_to_shoot == RED)
  {
   if(redTarget.moving == true)
   {
    targetTrackingColor = 0;
    state = shootMoving;
   }
   x_blob = redTarget.xPos;
   target_vert = 0;   // red (top)
  }
  else if(target_to_shoot == GREEN)
  {
   if(greenTarget.moving == true)
   {
    targetTrackingColor = 1;
    state = shootMoving;
   }
   x_blob = greenTarget.xPos;
   target_vert = 1;   // green (middle)
  }
  else if(target_to_shoot == BLUE)
  {
   if(blueTarget.moving == true)
   {
    targetTrackingColor = 2;
    state = shootMoving;
   }
   x_blob = blueTarget.xPos;
   target_vert = 2;   // blue (bottom)
  }

  DEBUG_PRINTLN("CMUcam4 complete. Please press button
when ready to shoot");

  // Wait for button to start tracking target
  #ifdef ENABLE_BUTTON
  while(digitalRead(switch_pin) == LOW){
   if(dynamic_test)
    break;
   if(digitalRead(switch_pin) == HIGH) {
    while(digitalRead(switch_pin) == HIGH);
    break;
   }
  }
  #endif

  // *****Calculate look up table indexes *************
  // Get horizontal position index (for Panning servo)
  target_hori = (x_blob-camera_pixel_min)/resolution;

  // Hardcoded motor speed
  motor_speed = 250;

  // Get pitching "section" index
  // The testbed is divided into "sections" as they require slightly
different pitching angle
  pitch_hori = target_hori/(camera_pixel_range/5);

  /* Saturate "section" index to prevent out of bounds, but it
rarely happens when the  range of CMUcam4 data fluctuates once
in a while. */
  if(pitch_hori > 4){
   pitch_hori = 4;
  }
  // If faulty index, return to "center" pitch
  else if(pitch_hori <0){
   pitch_hori= 2;
  }

  DEBUG_PRINTLN("Panning information:");
  DEBUG_PRINT("- Target Hori: ");
  DEBUG_PRINTLN(target_hori);
  DEBUG_PRINT("- Target Vert: ");
  DEBUG_PRINTLN(target_vert);

  delay(250);

  break;
 }
```

```
  //***********GET TARGET POSITION STATE***********/
  // Get target's horizontal location with table. For static targets
  case getDir: {
   DEBUG_PRINTLN("STATE: Get target location");

   target_position =
servo_table[target_vert][target_hori]+PAN_CENTER;

   // Next state
   state = turn;

   break;
  }

  /********ADJUST PITCH AND YAW TO AIM AT TARGET******/
  case turn: { // Servos for pitch and yaw are adjusted
   DEBUG_PRINTLN("STATE: Adjust Pitch & Yaw");

   // Next state. Return to "start" if pan/pitch data is faulty
   state = shoot;

   // Checks if location data is faulty (index out of bound).
   // Panning
   if (target_hori < 0){
    DEBUG_PRINTLN("Bad data for panning!");
    panS.write(PAN_CENTER-10);
    delay(500);
    panS.write(PAN_CENTER+10);
    delay(500);
    panS.write(PAN_CENTER-10);
    delay(500);
    panS.write(PAN_CENTER);
    motor_speed = 0;
    DEBUG_PRINTLN("Resetting to start");
    state = start;

   } else
    move_pan(target_position);

   // Pitching
   if(target_vert < 0 || target_vert > 2) {
    DEBUG_PRINTLN("Bad data for pitching!");
    pitchS.write(PITCH_CENTER-5);
    delay(500);
    pitchS.write(PITCH_CENTER+5);
    delay(500);
    pitchS.write(PITCH_CENTER-5);
    delay(500);
    pitchS.write(PAN_CENTER);
    motor_speed = 0;
    DEBUG_PRINTLN("Resetting to start");
    state = start;

   } else{
    target_pitch = pitch_table[target_vert][pitch_hori]+
PITCH_LEVEL;
    move_pitch(target_pitch);
   }

   // Debug statements
   /*
   DEBUG_PRINT("Panning Servo: ");
   DEBUG_PRINTLN(target_position);
   DEBUG_PRINT("Pitch Servo: ");
   DEBUG_PRINTLN(target_pitch);
   DEBUG_PRINT("Pitch vert: ");
   DEBUG_PRINTLN(target_vert);
   DEBUG_PRINT("Pitch area: ");
   DEBUG_PRINTLN(pitch_hori+1);
   */

   break;
  }

  /*******SHOOTING AT TARGET STATE**************/
  // Speed up DC motors
  case shoot: {
   DEBUG_PRINTLN("Speed Up Shooter State");

   // Start DC motor pins, opposite directions
   digitalWrite(left_dc_in1, HIGH);
   digitalWrite(left_dc_in2, LOW);
   digitalWrite(right_dc_in1, LOW);
   digitalWrite(right_dc_in2, HIGH);

   /* Only speed up DC motors if running for the first time (per
demonstration). The wheels will then continue to spin until the
end of demo */
   if(count_shot==0){
    DEBUG_PRINTLN("***Initial motor speedup***");
    for(int i = 90; i < motor_speed; i+=5) {
     analogWrite(left_dc_en, i);
     analogWrite(right_dc_en, i);
     delay(50);
    }
   }

   // Next state
   state = push;

   break;
  }

  /**********PUSH BALL**********************/
  // Push the ball to the shooter with rack and pinoin
  case push:  {
   DEBUG_PRINTLN("Pushing ball state");

   // Initialize timer
   int shake_timer;

   // Reset counter for rack and pinion
   counter_rack = 0;
```

```
    // Next state. Return to "start" state if ball is not found
    state = reset;

    // Note: Direction low = retrack (go backwards)
    // Make sure rack and pinion is all the way back
    digitalWrite(dir_pin_rack, LOW);
    while(digitalRead(rack_limit)==LOW){
     digitalWrite(step_pin_rack, LOW);
     delay(10);
     digitalWrite(step_pin_rack, HIGH);
     delay(10);
    }

    // DELAY: Give time for ball to roll into pusher
    delay(100);


    // ********Detecting ball on shooter*********
    /* If the ball is not properly positioned on the pusher, the
shooter will shake to either  reposition the ball or to indicate a
jam. If it cannot find a ball within 10 seconds, the system will
    return to state "start". */
    if(analogRead(ball_Sensor) > sensor_limit){
     shake_timer = second();
     while(analogRead(ball_Sensor) > sensor_limit){
      digitalWrite(left_dc_en, LOW);
      digitalWrite(right_dc_en, LOW);
      panS.write(PAN_CENTER+5);
      delay(200);
      panS.write(PAN_CENTER-5);
      delay(200);
      pitchS.write(PITCH_CENTER+5);
      delay(200);
      pitchS.write(PITCH_CENTER-5);
      delay(200);
      if(second()-shake_timer >= 10){
       DEBUG_PRINTLN("Couldn't find ball. Resetting to start");
       state = start;
       break;
      }
     }

     // If ball found, first set shooter in to "zero" position.
     move_pan(PAN_CENTER);
     move_pitch(PITCH_CENTER);

     // If indexes are out of bounds, don't bother returning to
position and return to "start"
     if(target_hori < 0 || (target_vert < 0 || target_vert > 2)){
      DEBUG_PRINTLN("Target hori or target vert out of bounds");
      DEBUG_PRINTLN("Resetting to start");
      state = start;
      break;
     }

     // Aim towards the target yet again
     move_pan(target_position);
     move_pitch(target_pitch);

     // Speed up the motor
     for(int i = 90; i < motor_speed; i+=5) {
      analogWrite(left_dc_en, i);
      analogWrite(right_dc_en, i);
      delay(50);
     }
    }
    // ********END************


    // Note: dir: High = Move forward
    digitalWrite(dir_pin_rack, HIGH);

    // The feeder approaches the shooter and waits for a bit to
calm the ball from jittering
    while(counter_rack < (FEED_COUNT*2)/3-15){
     digitalWrite(step_pin_rack, LOW);
     //delay(5);
     digitalWrite(step_pin_rack, HIGH);
     delay(5);
     counter_rack++;
    }

    delay(850);

    while(counter_rack < FEED_COUNT){
     digitalWrite(step_pin_rack, LOW);
     //delay(5);
     digitalWrite(step_pin_rack, HIGH);
     delay(5);
     counter_rack++;
    }

    // Set direction for rack and pinion to retract
    digitalWrite(dir_pin_rack, LOW);

    delay(250);

    break;
   }

   /****PUSH & SHOOTING STATE FOR DYNAMIC TARGETS***/
   case shootMoving:  {
    DEBUG_PRINTLN("Pushing ball & Shooting state for dynamic
targets");

    // Initialize a timer
    int shake_timer;

    // Reset counter for rack and pinion
    counter_rack = 0;

    // Calculate table index for center position (x_blob ~= 69)
    int dynamic_hori = (69-camera_pixel_min)/resolution;
```

```
    int dynamic_center =
servo_table[target_vert][dynamic_hori]+PAN_CENTER;

    // Move panning towards center

move_pan(servo_table[target_vert][dynamic_hori]+PAN_CENTER)
;

    DEBUG_PRINT("Dynamic hori: ");
    DEBUG_PRINTLN(dynamic_hori);
    DEBUG_PRINT("Dynamic middle: ");
    DEBUG_PRINTLN(servo_table[target_vert][dynamic_hori]);

    // Target will always be in middle, therefore use section [2],
where [2] region indicate middle area
    target_pitch = pitch_table[target_vert][2]+PITCH_LEVEL;
    move_pitch(target_pitch);

    // Next state
    state = reset;

    motor_speed = 250;

    // Start DC motor pins, opposite directions
    digitalWrite(left_dc_in1, HIGH);
    digitalWrite(left_dc_in2, LOW);
    digitalWrite(right_dc_in1, LOW);
    digitalWrite(right_dc_in2, HIGH);

    // Speed up DC motors
    /* Only speed up DC motors if running for the first time (per
demonstration). The wheels will then continue to spin until the
end of demo  */
    if(count_shot==0){
     DEBUG_PRINTLN("***Initial motor speedup***");
     for(int i = 90; i < motor_speed; i+=5) {
       analogWrite(left_dc_en, i);
       analogWrite(right_dc_en, i);
       delay(50);
     }
    }

    // Note: Direction low = retract (go backwards)
    // Make sure rack and pinion is all the way back
    digitalWrite(dir_pin_rack, LOW);
    while(digitalRead(rack_limit)==LOW){
     digitalWrite(step_pin_rack, LOW);
     delay(10);
     digitalWrite(step_pin_rack, HIGH);
     delay(10);
    }


     // ********Detecting ball on shooter************
        /* If the ball is not properly positioned on the pusher, the
shooter will shake to either reposition the ball or to indicate a
jam. If it cannot find a ball within 10 seconds, the system will
return to state "start".*/
    if(analogRead(ball_Sensor) > sensor_limit){
     shake_timer = second();
     while(analogRead(ball_Sensor) > sensor_limit){
       digitalWrite(left_dc_en, LOW);
       digitalWrite(right_dc_en, LOW);
       panS.write(PAN_CENTER+5);
       delay(200);
       panS.write(PAN_CENTER-5);
       delay(200);
       pitchS.write(PITCH_CENTER+5);
       delay(200);
       pitchS.write(PITCH_CENTER-5);
       delay(200);
       if(second()-shake_timer >= 10){
         DEBUG_PRINTLN("Couldn't find ball. Resetting to start");
         state = start;
         break;
       }
     }

     // Return pan and pitch to its position
     move_pan(PAN_CENTER);
     move_pitch(PITCH_CENTER);

     // If indexes are out of bounds, don't bother returning to
position and return to "start"
     if((!dynamic_test && target_hori < 0) || (target_vert < 0 ||
target_vert > 2)){
       DEBUG_PRINTLN("Indexes out of bounds. Resetting to
start");
       state = start;
       break;
     }

     // Aim towards target yet again
     move_pan(dynamic_center);
     move_pitch(target_pitch);

     // Speed up motors
     for(int i = 90; i < motor_speed; i+=5) {
       analogWrite(left_dc_en, i);
       analogWrite(right_dc_en, i);
       delay(50);
     }
    }
    // ********END************

    // Note: dir: High = Move forward
    digitalWrite(dir_pin_rack, HIGH);

    // Push ball to the wheels and wait
    while(counter_rack < (FEED_COUNT*2)/3-15){
     digitalWrite(step_pin_rack, LOW);
     //delay(5);
     digitalWrite(step_pin_rack, HIGH);
     delay(5);
```

```
    counter_rack++;
   }

   // Wait target to change direction
   DEBUG_PRINTLN("Tracking moving target");
   trackTarget();
   DEBUG_PRINTLN("Target hit edge");

   // Add delays to hit moving targets in time.
   if(target_to_shoot == RED){
    delay(650);
   }
   else if(target_to_shoot == GREEN){
    delay(500);
   }
   else if(target_to_shoot == BLUE){
    delay(400);
   }

   // Push ball to shooter
   while(counter_rack < FEED_COUNT){
    digitalWrite(step_pin_rack, LOW);
    //delay(5);
    digitalWrite(step_pin_rack, HIGH);
    delay(5);
    counter_rack++;
   }

   // Set direction for rack and pinion to retract
   digitalWrite(dir_pin_rack, LOW);

   counter_rack = 0;
   to_moving_state = 1;    // Indicate that it's in dynamic state
(used for reset state)

   delay(250);
   break;
  }

  /******RESET RACK AND PINION STATE************/
  case reset:  {
   DEBUG_PRINTLN("Rack and Pinion reset state");

   // Retrack rack and pinion, limited by limit switch
   while(digitalRead(rack_limit)==LOW){
    digitalWrite(step_pin_rack, LOW);
    //delay(5);
    digitalWrite(step_pin_rack, HIGH);
    delay(5);
   }

   // Next state
   state = getData;

   // Count number of balls shot, max 6
   count_shot++;
   if(count_shot == 6){
    state = halt;
   }
   break;
  }
 }
}

// Setup CMUCAM4
void cmucamSetup()
{
 pinMode(LED_PIN, OUTPUT);

 cam.begin();
 cam.LEDOn(10);

 delay(5000);

 cam.autoGainControl(0);
 cam.autoWhiteBalance(0);

 cam.LEDOff();

 int result = cam.getButtonState();

 if(result >= 0)
  digitalWrite(LED_PIN, result ? HIGH : LOW);

 resetTrackingData();
}

// Reset target data
void resetTrackingData()
{
 //reset tracking data
 if(dynamic_test){
  DEBUG_PRINTLN("Reset structs at dynamic");
  redTarget.moving = false;
  redTarget.xPos = 0;
  greenTarget.moving = false;
  greenTarget.xPos = 0;
  blueTarget.moving = false;
  blueTarget.xPos = 0;
 }
 else {
  DEBUG_PRINTLN("Reset structs at static");
  redTarget.moving = true;
  redTarget.xPos = 0;
  greenTarget.moving = true;
  greenTarget.xPos = 0;
  blueTarget.moving = true;
  blueTarget.xPos = 0;
 }
}

/*
mode: 0 - normal
    1 - check specific color (trackLastColor) if dynamic
```

```
    2 - check specific color (trackLastColor) if static
    3 - look for dynamic target
    4 - look for static target

if mode == 1 or mode == 2
 if returned value > 0, bad miss
 if returned value == 0, good confirm hit

if mode == 3 or mode == 4
 returned value is color needed to hit
*/

// Track targets
int getCmucamData(int mode, int trackLastColor, boolean
skipRed, boolean skipGreen, boolean skipBlue)
{
 cmucamState = 0;


 if(mode == 0)
 {
  skipRed = false;
  skipGreen = false;
  skipBlue = false;
 }
 else if(mode == 1 || mode == 2)
 {
  if(trackLastColor == RED)
  {
   skipGreen = true;
   skipBlue = true;
  }
  else if(trackLastColor == GREEN)
  {
   skipRed = true;
   skipBlue = true;
  }
  else if(trackLastColor == BLUE)
  {
   skipGreen = true;
   skipRed = true;
  }
 }

 int redMax, redMin, greenMax, greenMin, blueMax, blueMin;

 boolean error;

 while(1)
 {
  CMUCAM4_track_data_t data;

  error = false;

  if(cmucamState == 0 && skipRed == false)          // red
  {
   cam.setTrackingParameters(RED_RL, RED_RH, RED_GL,
RED_GH, RED_BL, RED_BH);
   cam.setTrackingWindow(LEFT_X, TOP_Y, RIGHT_X, MID1_Y);
  }
  else if(cmucamState == 1 && skipGreen == false)       // Green
  {
   cam.setTrackingParameters(GREEN_RL, GREEN_RH,
GREEN_GL, GREEN_GH, GREEN_BL, GREEN_BH);
   cam.setTrackingWindow(LEFT_X, MID1_Y, RIGHT_X, MID2_Y);
  }
  else if(cmucamState == 2 && skipBlue == false)        // Blue
  {
   cam.setTrackingParameters(BLUE_RL, BLUE_RH, BLUE_GL,
BLUE_GH, BLUE_BL, BLUE_BH);
   cam.setTrackingWindow(LEFT_X, MID2_Y, RIGHT_X, BOT_Y);
  }
  else if(cmucamState == 4)
   return -1;            // Failed?

  if(cmucamState < 3)
  {
   cam.trackColor(); // Start streaming

   int cycles = 0;

   //get data from camera
   do
   {
    cam.waitStream();
    cam.parseTPacket(&data);

    xData[cmucamState*CYCLES_MAX + cycles] = (int)
data.centroidX;
    yData[cmucamState*CYCLES_MAX + cycles] = (int)
data.centroidY;

    cycles++;
   }
   while(cycles < CYCLES_MAX);

   if(cmucamState == 0 && skipRed == false)
   {
    int redSum = 0;
    int redZero = 0;
    int redDiv = CYCLES_MAX;

    redMin = xData[0];
    redMax = xData[0];

    //analyze red
    DEBUG_PRINT("RED DATA: ");
    for(int i = 0; i < CYCLES_MAX; i++)
    {
     if(xData[i] != 0)
     {
      redSum += xData[i];
```

```
      DEBUG_PRINT(xData[i]);
      DEBUG_PRINT(" ");

      if(xData[i] > redMax)
        redMax = xData[i];

      if(xData[i] < redMin)
        redMin = xData[i];
     }
     else if(xData[i] == 0)
      redZero++;
    }

    DEBUG_PRINTLN("");

    if(abs(redMax - redMin) > 20)
     error = true;


    if(redZero >= 2)
    {
     redTarget.moving = false;
     redTarget.xPos = 0;
     DEBUG_PRINTLN("Red Multiple not seen");
    }
    else if(abs(redSum/redDiv - xData[0]) > 2 || abs(redMax -
redMin) > 2)
    {
     DEBUG_PRINTLN("Red Moving");

     redTarget.xPos = 0;
     redTarget.moving = true;

     if(mode == 1 || mode == 3) //tracking dynamic
      return RED;

     else if(mode == 2)
      return 0;  //good

    }
    else
    {
     DEBUG_PRINTLN("Red Not Moving");
     DEBUG_PRINT("Final X Position: ");
     DEBUG_PRINTLN(redSum/redDiv);

     redTarget.moving = false;
     redTarget.xPos = redSum/redDiv;

     if(mode == 2 || mode == 4)  //tracking static
      return RED;

     else if(mode == 1)
      return 0; //good

    }
   }
   else if(cmucamState == 1 && skipGreen == false)
   {
    int greenSum = 0;
    int greenZero = 0;
    int greenDiv = CYCLES_MAX;

    greenMin = xData[CYCLES_MAX];
    greenMax = xData[CYCLES_MAX];

    DEBUG_PRINT("GREEN DATA: ");
    for(int i = 0; i < CYCLES_MAX; i++)
    {

     if(xData[CYCLES_MAX + i] != 0)
     {
      greenSum += xData[CYCLES_MAX + i];

      DEBUG_PRINT(xData[CYCLES_MAX + i]);
      DEBUG_PRINT(" ");

      if(xData[CYCLES_MAX + i] > greenMax)
       greenMax = xData[CYCLES_MAX + i];

      if(xData[CYCLES_MAX + i] < greenMin)
       greenMin = xData[CYCLES_MAX + i];

     }
     else if(xData[CYCLES_MAX + i] == 0)
      greenZero++;
    }
    DEBUG_PRINTLN("");

    if(abs(greenMax - greenMin) > 20)
     error = true;


    if(greenZero >= 2)
    {
     greenTarget.moving = false;
     greenTarget.xPos = 0;
     DEBUG_PRINTLN("Green Multiple not seen");
    }
    else if(abs(greenSum/greenDiv - xData[CYCLES_MAX]) > 2 ||
abs(greenMax - greenMin) > 2)
    {
     DEBUG_PRINTLN("Green Moving");

     greenTarget.xPos = 0;
     greenTarget.moving = true;

     if(mode == 1 || mode == 3) //tracking dynamic
      return GREEN;

     else if(mode == 2)
       return 0;  //good

    }
```

```
 else
 {
  DEBUG_PRINTLN("Green Not Moving");
  DEBUG_PRINT("Final X Position: ");
  DEBUG_PRINTLN(greenSum/CYCLES_MAX);

  greenTarget.xPos = greenSum/CYCLES_MAX;
  greenTarget.moving = false;

  if(mode == 2 || mode == 4)  //tracking static
    return GREEN;

  else if(mode == 1)
    return 0; //good
  }
 }
}
else if(cmucamState == 2 && skipBlue == false)
{
 int blueSum = 0;
 int blueZero = 0;
 int blueDiv = CYCLES_MAX;

 blueMin = xData[CYCLES_MAX*2];
 blueMax = xData[CYCLES_MAX*2];

 DEBUG_PRINT("BLUE DATA: ");

 for(int i = 0; i < CYCLES_MAX; i++)
 {

  if(xData[2*CYCLES_MAX + i] != 0)
  {
    blueSum += xData[2*CYCLES_MAX + i];
    DEBUG_PRINT(xData[2*CYCLES_MAX + i]);
    DEBUG_PRINT(" ");
    if(xData[CYCLES_MAX*2 + i] > blueMax)
      blueMax = xData[CYCLES_MAX*2 + i];

    if(xData[CYCLES_MAX + i] < blueMin)
      blueMin = xData[CYCLES_MAX*2 + i];
  }
  else if(xData[2*CYCLES_MAX + i] == 0)
    blueZero++;

 }
 DEBUG_PRINTLN("");

 if(abs(blueMax - blueMin) > 20)
  error = true;

 if(blueZero >= 2)
 {
  blueTarget.moving = false;
  blueTarget.xPos = 0;
  DEBUG_PRINTLN("Blue Multiple not seen");
 }
     else if(abs(blueSum/blueDiv - xData[2*CYCLES_MAX]) > 2 ||
abs(blueMax - blueMin) > 2)
     {
      DEBUG_PRINTLN("Blue Moving");

      blueTarget.xPos = 0;
      blueTarget.moving = true;

      if(mode == 1 || mode == 3) //tracking dynamic
        return BLUE;

      else if(mode == 2) {
      {
        return 0;  //good
      }
     }
     else
     {
      DEBUG_PRINTLN("Blue Not Moving");
      DEBUG_PRINT("Final X Position: ");
      DEBUG_PRINTLN(blueSum/CYCLES_MAX);

      blueTarget.xPos = blueSum/CYCLES_MAX;
      blueTarget.moving = false;

      if(mode == 2 || mode == 4)  //tracking static
        return BLUE;

      else if(mode == 1)
        return 0; //good

     }
    }
    cam.stopStream(); // Stop streaming
   }

   if(error != true)
    cmucamState++;
   else
    DEBUG_PRINTLN("ERROR DETECTED IN COLOR DETECTION");
 }
}

void trackTarget()
{
 CMUCAM4_track_data_t data;

 DEBUG_PRINT("Tracking Target: ");
 DEBUG_PRINTLN(target_to_shoot);

 cmucamState = 0;
 int iter = 0;
 int trackingX[180];

 enum dir {left, right, none};
 dir d = none;
```

```
 if(target_to_shoot == RED)                    // Red
 {
  cam.setTrackingParameters(180, 255, 50, 180, 80, 200);
  cam.setTrackingWindow(LEFT_X, TOP_Y, RIGHT_X, MID1_Y);
 }
 else if(target_to_shoot == GREEN)             // Green
 {
  cam.setTrackingParameters(0, 150, 120, 210, 80, 180);
  cam.setTrackingWindow(LEFT_X, MID1_Y, RIGHT_X, MID2_Y);
 }
 else if(target_to_shoot == BLUE)              // Blue
 {
  cam.setTrackingParameters(0, 180, 60, 130, 140, 255);
  cam.setTrackingWindow(LEFT_X, MID2_Y, RIGHT_X, BOT_Y);
 }

 while(1)
 {
  if(cmucamState == 0)
  {
   cam.trackColor(); // Start streaming

   while(1)
   {
    cam.waitStream();
    cam.parseTPacket(&data);

    trackingX[iter] = (int) data.centroidX;
    DEBUG_PRINTLN(trackingX[iter]);

    if(iter > 6)
    {
     if(d == none)
     {
      if(trackingX[iter] > trackingX[iter - 4])  // target moving
right?
       d = right;

      else if(trackingX[iter] < trackingX[iter - 4]) //target moving
left?
       d = left;

     }
     else if(d == right)
     {
      if(trackingX[iter] < trackingX[iter - 4]) // target moving left?
       break;

     }
     else if(d == left)
     {
      if(trackingX[iter] > trackingX[iter - 4])  // target moving
right?
       break;

     }
    }
    iter++;
   }

   cam.stopStream(); // Stop streaming

   return; //target has been found
  }
  else if(cmucamState == 1)
  {
   if(d == right)
   {
    DEBUG_PRINTLN("Target is moving to the center from the
right");
   }
   else if(d == left)
   {
    DEBUG_PRINTLN("Target is moving to the center from the
left");
   }
  }
  cmucamState++;
 }
}

// Move panning motor tick by tick to reduce stress
void move_pan(int desired_pan){
 int i;
 current_pan = panS.read();
 new_pan = desired_pan;
 // Determine direction of movement
 if(new_pan - current_pan  < 0)
  dir_pan = -1;
 else
  dir_pan = 1;

 for(i = current_pan; i != new_pan; i+= dir_pan){
  panS.write(i);
  delay(100);
 }
}
// Move pitching motor tick by tick to reduce stress
void move_pitch(int desired_pan){
 int i;
 current_pan = pitchS.read();
 new_pan = desired_pan;
 // Determine direction of movement
 if(new_pan - current_pan  < 0)
  dir_pan = -1;
 else
  dir_pan = 1;

 for(i = current_pan; i != new_pan; i+= dir_pan){
  pitchS.write(i);
  delay(100);
 }
}
void create_table(){
```

```
  // Tables cropped to shorten length of code
  // PITCH TABLE INSTANTIATION

  // PAN TABLE INSTANTIATION
  //********************RED TARGET*****************//
  //*****************GREEN TARGET******************//
//**************BLUE TARGET*********************//
}
```