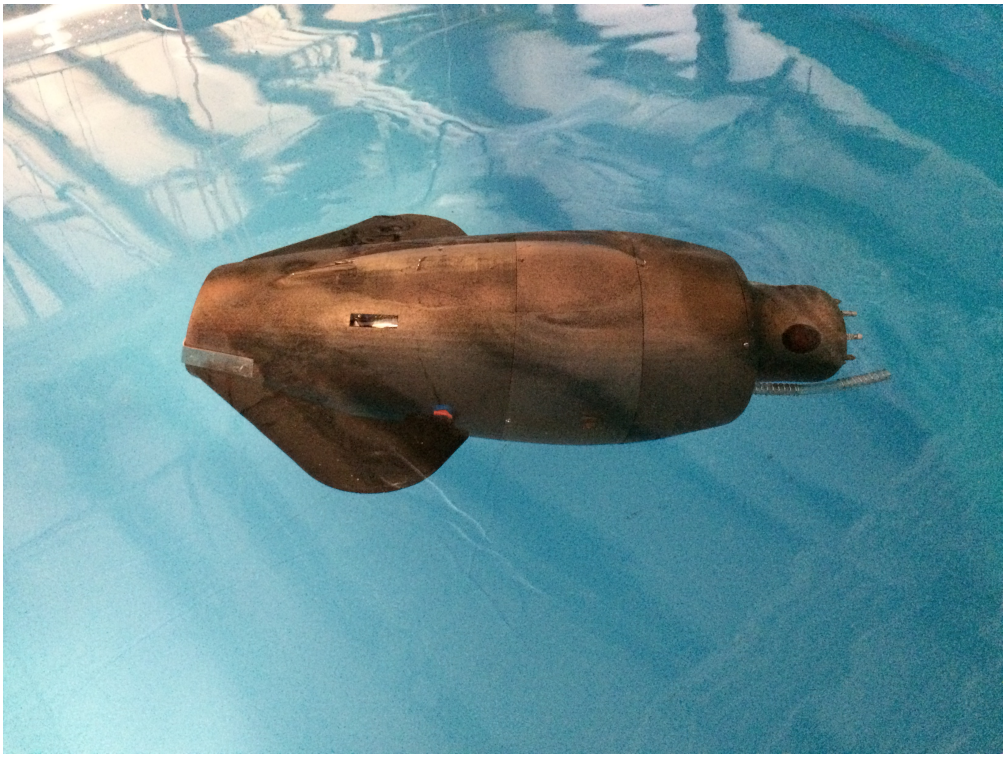
Fundamentals of Robotics
Spiral 2: A Robotic Squid

Paul Nadan

December 15, 2017



1 Executive Summary

Our team set out to design a robot capable of autonomously navigating between colored buoys on the surface of the pool. The core design requirements included floating with close to neutral buoyancy, locating and moving between colored targets, and sending missions wirelessly. Additionally, the robot's design had to be inspired by a marine animal and thus propellers could not be used. Our team chose to create a robotic squid with steerable jets and actuated fins for locomotion. A PixyCam allows the robot to identify colored targets and an XBee radio communicates with the operator to receive missions and send back data. This robot was intended to be the second of three iterations with the end goal of creating a bio-mimetic submersible capable of autonomously retrieving items and navigating to targets, but due to time constraints this will be the final iteration. While currently unable to submerge itself, the robot's almost neutral buoyancy still makes it possible to retrofit for diving in the future.

During the final demonstration, our robot was capable of successfully floating with close to neutral buoyancy, moving under its own power, sending missions wirelessly, and locating colored targets. The design also scored points for aesthetics and good engineering. However, due to issues with controlling the valves in the jet system, the robot was unable to operate one of its two jets. The uneven thrust prevented the robot from travelling in a straight line, making it impossible to maneuver towards the targets and complete the mission. As a result of this failure, our robot placed last out of four competing robots, with an overall score of 19/100 possible points.

2 Mechanical System

2.1 Overview

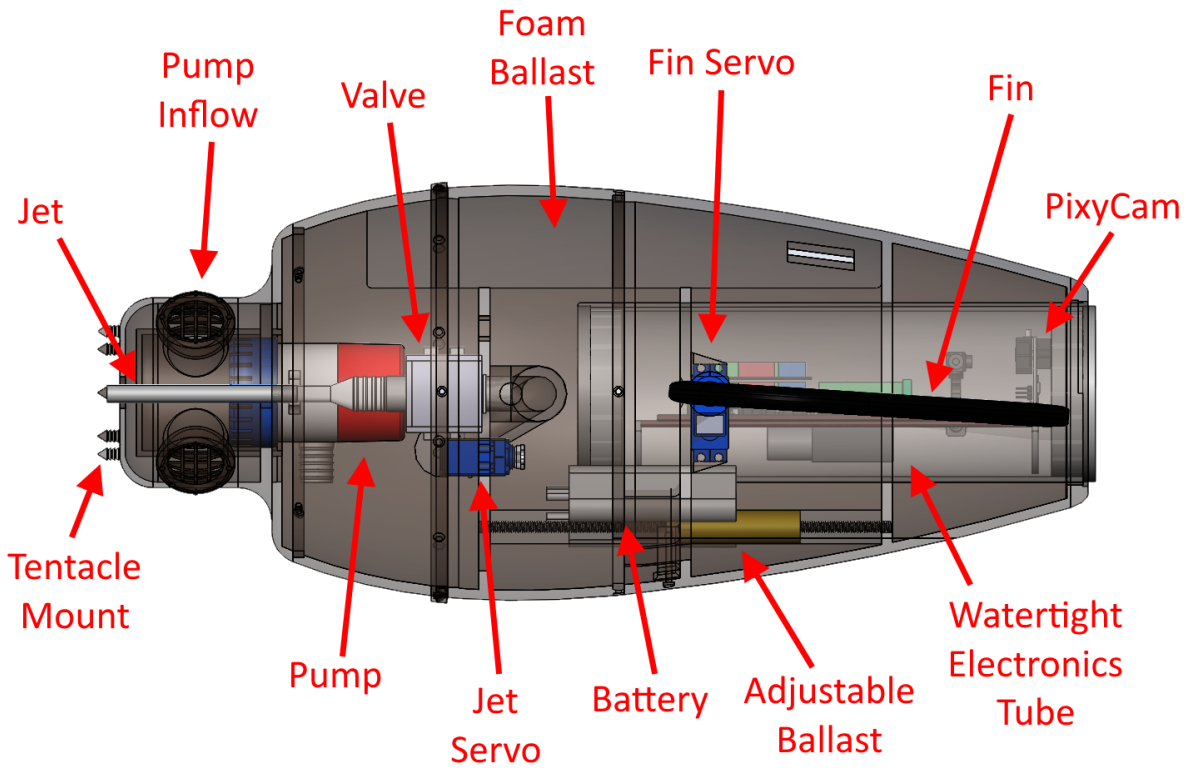


Figure 1: The positioning of the robot's key components.

Our design is modeled after a squid, with a sleek body, steerable jets, fins, and tentacles (removed during testing). The entire robot measured 22.3in long by 15.1in wide by 9.4in deep, including fins but excluding tentacles. Propulsion was provided by the two jets, which were pressurized using a bilge pump. The robot has three independent steering systems for redundancy: the fins can be angled, the jets can be steered, and a pair of valves can block the flow from of one jet or the other. A piece of foam near the top of the robot keeps it oriented upright and helps it float, while an adjustable piece of ballast in the base of the squid allows the center of mass to be aligned easily. A watertight tube protects the sensitive electronics, with a clear acrylic piece inset in the endcap for the PixyCam to look through.

2.2 Hull

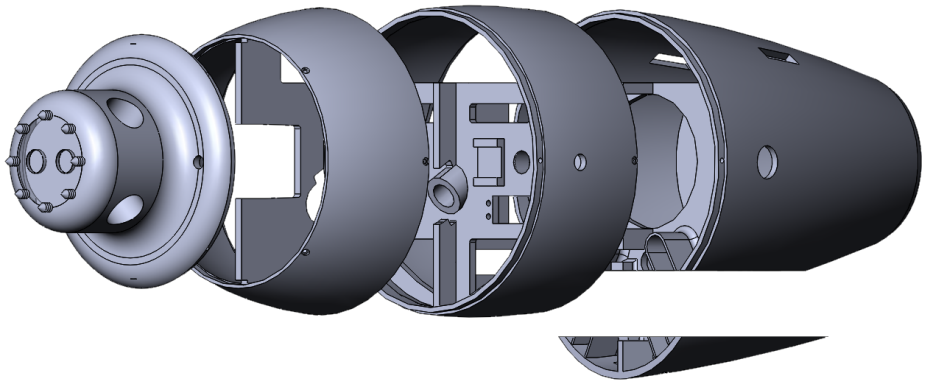


Figure 2: The robot's 3D printed hull, divided into four sections and a bottom hatch.

The hull was 3D printed and painted a bronze color to match the robot's steampunk theme. The profile of the hull was modeled after a squid, which has a very hydrodynamic shape. The hull was divided into four sections for ease of printing and joined together using threaded inserts. A removable hatch on the bottom makes swapping batteries easier. Several cross-sectional pieces were added to strengthen the structure, with holes left for fitting the various other components. Foam ballast at the top and a brass rod at the bottom of the robot were intended to make the robot float level, with near neutral buoyancy. During testing, additional ballast was added near the front of the robot to counterbalance the buoyancy of the watertight tube.

2.3 Jets

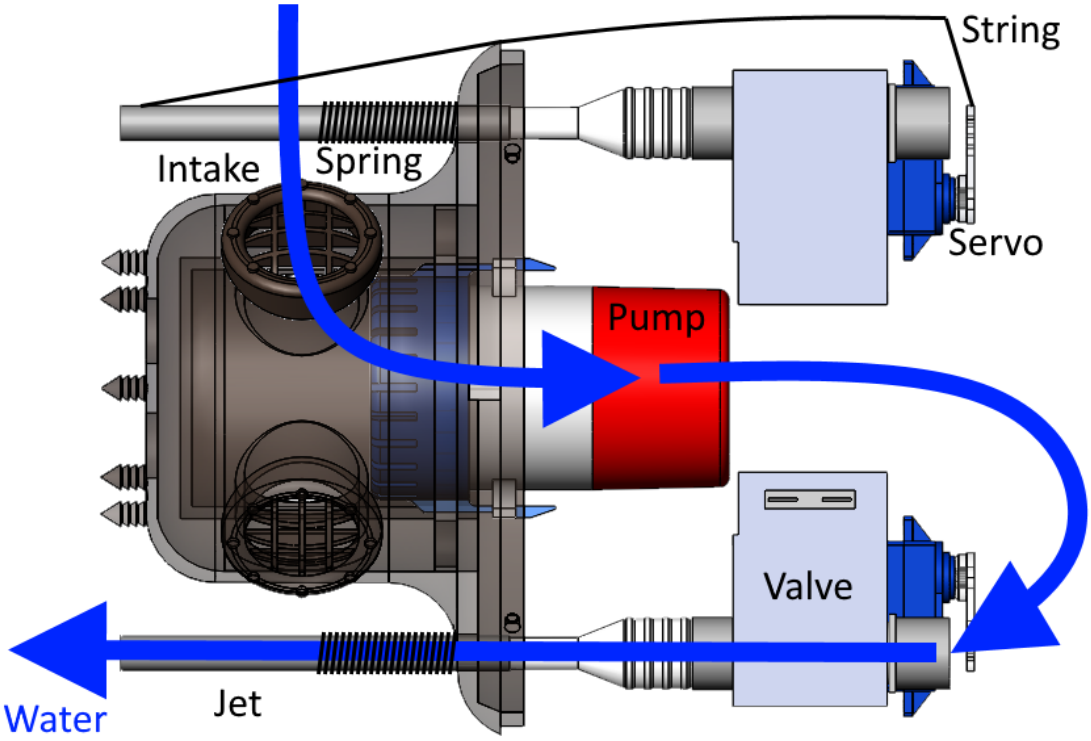


Figure 3: The jet propulsion system. Water is sucked through the intakes by the pump, and pressurizes a bladder (not shown). When a valve is opened, the water flows through and out of a jet. A string attaching the jet to a servo allows it to be pulled to point a different direction, and a spring wrapped around the jet restores it to its normal position.

To propel itself, the robot intakes water through vents near the back of the hull using a bilge pump. The pump pressurizes a bladder (we used a rubber glove), which can build up water pressure to release when one of the two valves is opened. The valves are each connected to a steerable jet, which can be pulled to point outwards or even forwards by pulling a fishing line string using a servo. A spring wrapped around each jet returns it to pointing straight when the servo releases tension in the string. Turning can be achieved by opening one valve and closing the other to create a one-sided thrust or by directing one jet forwards and the other backwards to apply a torque on either side of the robot. In practice, the system for directing the jet was never tested due to time constraints, although all of the requisite mechanical components were in place besides the string. Additionally, shortly before the demonstration, one valve closed and could not be reopened, rendering steering impossible. We have yet to diagnose the issue, but it could be caused by the valve's lack of waterproofing, or by wiring issues or code bugs. Even when the valves were both open, the pump was unable to provide very much pressure, and the squid moved at a fairly low velocity. This resulted from powering the pump off of seven volts rather than the full twelve volts that it is rated for.

2.4 Fins

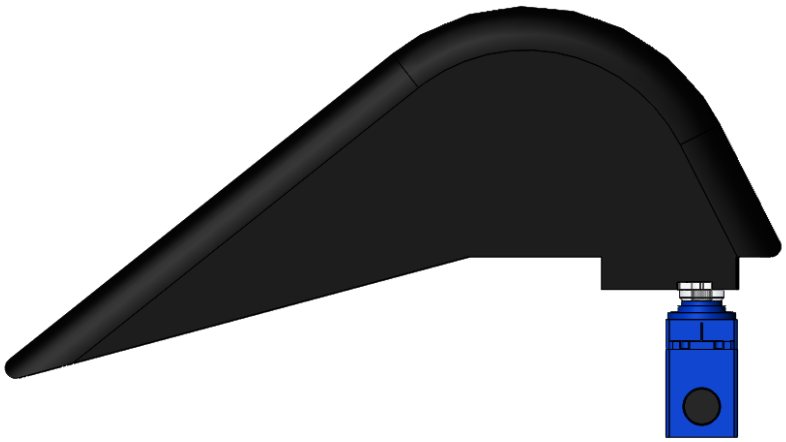


Figure 4: A fin mounted to a servo. Angling the fin creates hydrodynamic drag on that side of the robot, enabling it to turn.

As a backup steering system, we included two fins, one on each side of the head. The fins can be angled using an attached servo motor. As the angle increases, the fin's cross-sectional area in the direction of motion increases as well, creating hydrodynamic drag. By only applying this drag on one side a torque is created, causing the robot to turn. The shape of the fin was chosen to aesthetically mimic that of a squid and to reduce water resistance when in the straight position. In practice, the robot's low velocity meant that drag was fairly insignificant, and the fins alone had insufficient authority to change the robot's direction. Additionally, the point where the valves connected to the motor was not very tight and so the fins often detached despite repeated application of superglue.

2.5 Watertight Tube

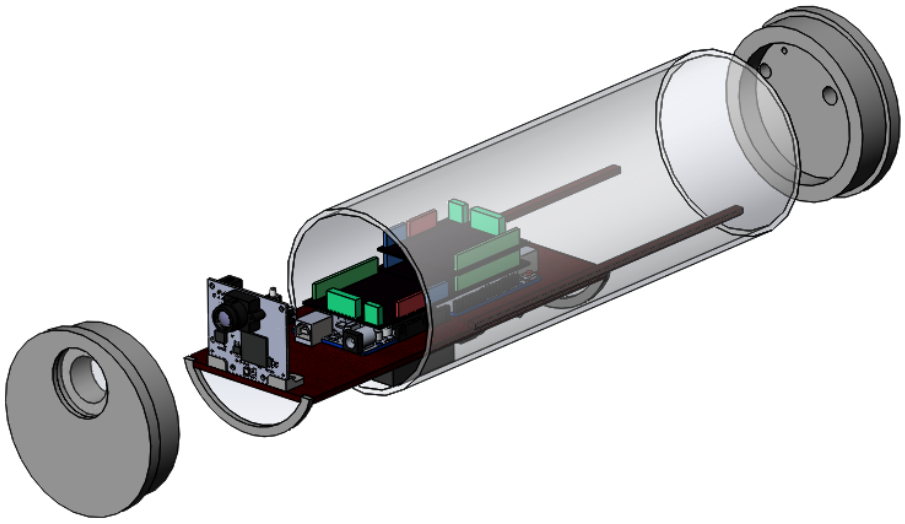


Figure 5: The watertight electronics tube. The electronics can be slid out for easy access.

To protect sensitive electrical components like the Arduinos, we sealed them inside of a watertight plastic tube. The endcaps are sealed in place using O-rings, and a cutout from one of the endcaps is covered with transparent acrylic, enabling the PixyCam to see out of the tube. To route power in and out of the tube, two conductive pass-throughs are screwed through the endcap, with gasket-maker used to seal any gaps. For the signal wires, a more difficult solution was needed. Three bolts with holes bored through the center using a lathe were screwed into the endcap using gasket maker. To fill the gap between the wires running through each bolt, we used sealant. However, during testing we discovered that the sealant was insufficient and water still got into the tube. We next tried filling the gaps with tool dip, but it did not seal around the wires tightly enough and water was still able to seep through. Finally we filled the entire inside surface of the endcap with large amounts of epoxy, which gave us a nice watertight seal. We also added a pressure fitting to the endcap that would allow us to pressurize the inside of the tube, so that in the event of a leak air would leak out rather than water coming in. To enable easier access to the electronics, we mounted them on a wooden board that could slide in and out of the tube. A metal band prevents the tube from sliding out of the robot.

3 Electrical System

3.1 Power Flow

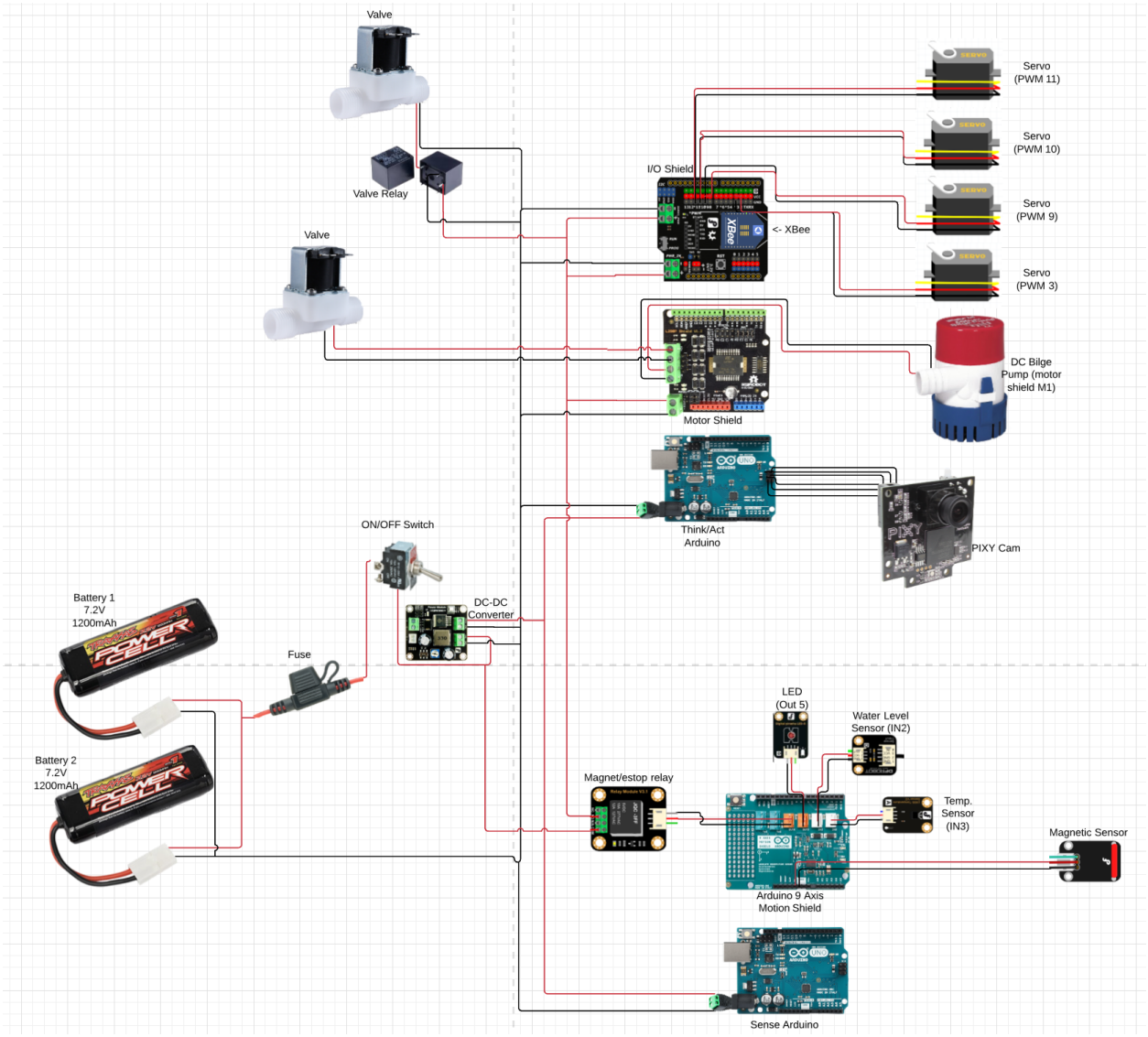


Figure 6: A diagram of the power flow through the robot. While the PixCam was initially attached to the Sense Arduino, it was moved to the Think/Act Arduino as shown.

Our robot is powered by a pair of rechargeable, 7.5V lithium batteries wired together in parallel. The positive terminal is connected to a fuse to prevent damage in the event of a short circuit and a switch to turn off the robot. From there, a DC/DC converter regulates the voltage so that the Arduinos receives a consistent voltage input even as the battery voltage drops and avoids spikes in current caused by the motors. The power from the regulator feeds into two Arduinos, one designated for "Sense" code, i.e. operating the PixyCam and other sensors, and one designated for "Think" and "Act" code, i.e. making decisions, receiving missions, and operating actuators. A motor shield on the Think/Act Arduino is powered off of the unregulated current for running the servos, and an I/O shield on top of the motor shield allows for clean wiring and easy repositioning of servo ports. The power for the motor shield runs through a relay designed to act as an E-stop by blocking current when triggered by the Sense Arduino. A magnetic switch hooked

up to the Sense Arduino allows operators to disable the robot without opening the tube, even if the wireless communication stops working.

The Sense Arduino provides power to several sensors, including the magnetic switch as well as a flood sensor and temperature sensor, designed to detect critical problems during a mission. The PixyCam initially drew power from the Sense Arduino, but during testing we moved the PixyCam to the Think/Act Arduino instead. The Think/Act Arduino provides power to four servos (two for the jets and two for the fins), the bilge pump, and the two valves. One of the valves is actually triggered by a relay, due to a shortage of outputs from the motor shield. An XBee attached to the I/O shield on the Think/Act Arduino also draws power.

According to our power calculations, our maximum current draw is around 11A. The batteries can hold 1200mAh each or 2400mAh combined, giving the robot a battery life of 13 minutes under full load. However, because the servos move fairly infrequently, the actual battery life is significantly longer in practice.

Table 1: Power Usage

Component	**Voltage**	**Current**	**Quantity**
Servo	7.2V	2A (load), 330mA (no load)	4
Pump	7.2V	2A	1
Arduino Uno	5V	47mA	2
DC Motor Shield	5V	36mA	1
PixyCam	5V	140mA	1
Valve	7.2V	400mA	2

3.2 Data Flow

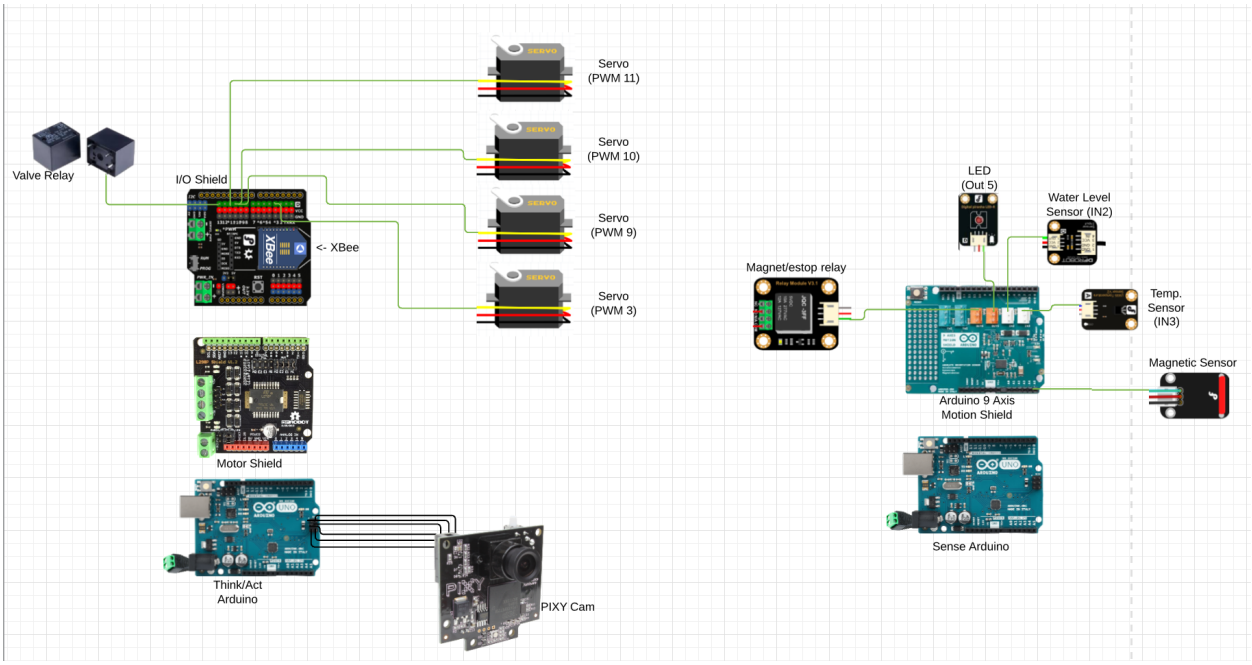


Figure 7: A diagram of the data flow through the robot. Note that the two Arduinos are no longer communicating.

The data diagram centers around the two Arduinos, one for Sense and one for Think and Act code. They were originally in communication with each other using a protocol called SoftwareSerial, but during testing we switched the PixyCam to the ThinkAct Arduino to avoid conflicts with the servo library and severed all communication between the two. Currently, the Sense Arduino is only used to E-Stop the robot and potentially control LEDs added to the tentacles. The Sense Arduino receives data from the magnetic switch and the temperature and flood sensors, which it can use to determine whether to trigger the E-stop relay. The Sense Arduino is also equipped with a 9-axis motion shield, intended for possible use in a future mission.

The Think/Act Arduino receives object detection data from the PixyCam and operator commands through an XBee. It also sends status information back to the operator through the XBee connection. Initially, we were using a pair of XBees without antenna, which were unable to communicate while the robot was in the water. Switching out our XBees solved the problem and greatly extended our communication range. The Arduino sends PWM signals to the four servos using the motor shield, and also sends an voltage output to one valve through the motor shield and the other through a relay.

3.3 Electrical Implementation

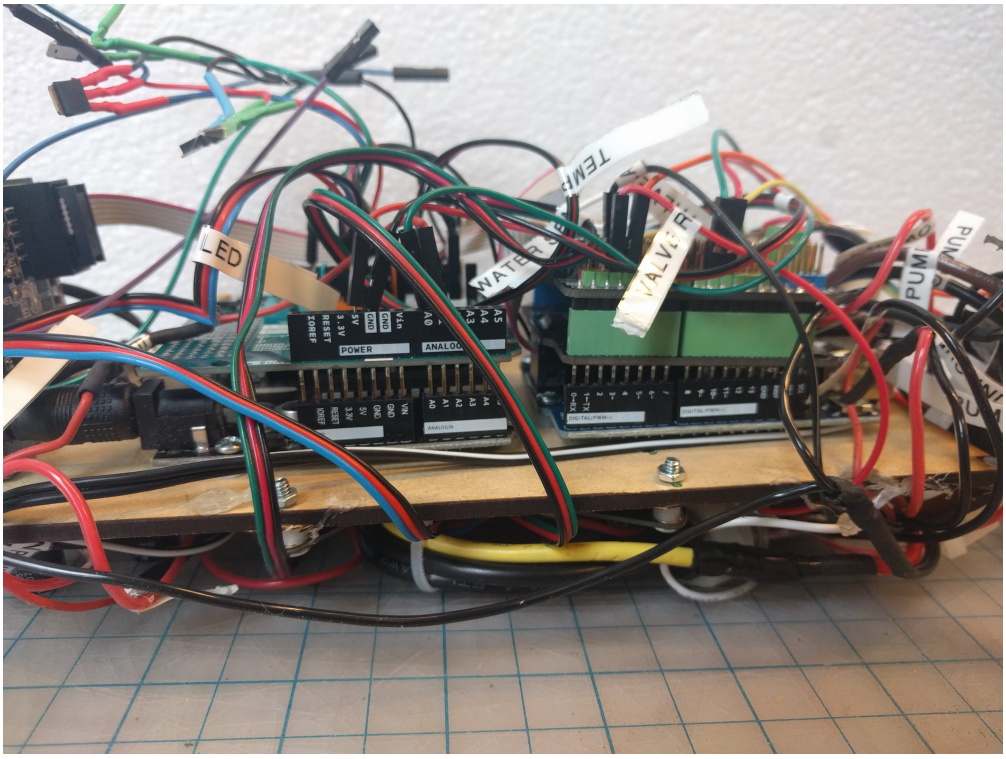


Figure 8: The physical electronics on the robot.

When designing the electrical system, the primary concern was waterproofing. While sealing the watertight tube was covered in the mechanical design section above, by necessity the batteries, valves, servos, and pump were all mounted outside of the tube. The batteries were coated in tool-dip for protection, while the pump and servos were already waterproof. The valves were not water-proof, and so they were encased in a sealed enclosure. However, the enclosure was not watertight and it was judged that the valves would probably work fine in water anyway, so further attempts to waterproof them were abandoned.

To minimize opportunities for leaks, as few wires were routed out of the tube as possible. The external components all shared common power and ground terminals that passed through the endcap. Additionally, connectors were used so that the tube could be detached from the rest of the robot without the need to open it. Wires were carefully labeled so that attaching and reattaching the tube to the robot could be done more easily.

While the electronics manifested several bugs, for the most part problems were identified and treated fairly fast. Problems encountered included the Arduino requiring seven volts rather than five from the DC converter to operate certain components at five volts, issues with one of the relays breaking off a lead and requiring replacement, and accidentally using Arduino ports that were already in use by either the shields, the PixCam, or the XBee. By demo day, most of our electrical issues had been worked out, aside from an issue with controlling the valves that may have had an electrical cause.

4 Control Software

4.1 Sense Setup

```
/**
 * Sprint 2 Code
 * Sense
 * SquidBot
 * Mission: Drive straight to buoy, turn in circle, drive to next buoy, etc.,
 *          then back home
 * Team Squid: Aubrey, Diego, Gretchen, Jon, MJ, Paul
 * 12/12/2017
 * Version 3
 */
//library for serial communication
//#include <EasyTransfer.h>
//#include <SoftwareSerial.h> //need this library to run Software Serial

//libraries included to use PixyCam
#include <SPI.h>
#include <Wire.h>
//#include <PixyI2C.h>
#include <Pixy.h>

//#include "PixyUART.h"


//library included to use servos
#include<Servo.h>

//libraries included to use motor and motion shield
//#include "NineAxesMotion.h"

//Constants and Global Variables
//Pixy pixy; //creates PixyCam object to use
//EasyTransfer ETin, ETout; //creates serial structures to transfer data

//flood True if hull flooding
//temp true if electronics overheating
boolean flood, temp, estop = false;

// Pins
const int FLOODPIN = A3;
const int MAX_BLOCKS = 7;
const int STOP = A0; // Magnetic sensor pin to determine eStop
const int TEMP = A2;
const int RELAY = 5;

//struct SEND_DATA_STRUCTURE{
//  //put your variable definitions here for the data you want to receive
//  //THIS MUST BE EXACTLY THE SAME ON THE OTHER ARDUINO
//  float widths[MAX_BLOCKS];
//  int16_t signatures[MAX_BLOCKS];
//  float positions[MAX_BLOCKS];
//  boolean estop;
//};
//
////give a name to the group of data
//SEND_DATA_STRUCTURE txdata;


//SETUP ROBOT CODE (RUN ONCE)
void setup() {
  Serial.begin(9600);

  //pixy.init();
  //Arduino.begin(4800);
```

```
64
65    //ETout.begin(details(txdata), &Serial);
66
67    pinMode(STOP, INPUT);
68    pinMode(FLOODPIN, INPUT);
69    pinMode(RELAY, OUTPUT);
70
71    delay(100);
72  }
```

In the setup function, the Arduino initializes the pinmodes of the sensors. The commented out code was used to establish SoftwareSerial communication between the two Arduinos, but was abandoned due to conflicts with the Think/Act Arduino's Servo library. There is also commented code that initialized the PixyCam before it was moved to the other Arduino.

4.2 Sense Loop

```
1  //ROBOT CONTROL LOOP (RUNS UNTIL STOP)
2  void loop() {
3  //  ETout.sendData();
4    delay(20);
5    //checkFlood();
6    //checkTemp();
7  }
8
9  //CONTROL FUNCTIONS
10
11 // Check the flood sensor for problems
12 void checkFlood(){
13   int liquidLevel = digitalRead(FLOODPIN);
14   if(liquidLevel == HIGH){
15     flood = true;
16   }
17 }
18
19 // Check the temperature sensor for problems
20 void checkTemp(){//temp 150F
21   int val=analogRead(TEMP);//Connect LM35 on Analog 0
22   float dat = (double) val * (5/10.24);
23   if(dat >= 65.5){
24     temp = true;
25   }
26 }
27
28 // E-stop the robot using the relay
29 void eStop(){
30   estop = digitalRead(STOP);
31   if(eStop){
32     digitalWrite(RELAY, HIGH);
33   }
34 }
```

The loop function is called iteratively while the robot is running. Currently, the Sense Arduino actually does nothing during each loop. As mentioned earlier, the PixyCam was moved to the other Arduino. Additionally, there was not enough time to test out the temperature and flood sensors, so their values are currently being ignored. However, helper functions exist for reading their values and triggering the E-stop relay.

4.3 Think/Act Setup

```
1    /**
2   * Sprint 2 Code
3   * Think/Act
```

```
 * SquidBot
 * Mission: Drive straight to buoy, turn in circle, drive to next buoy, etc.,
 *          then back home
 * Team Squid: Aubrey, Diego, Gretchen, Jon, MJ, Paul
 * 12/12/2017
 * Version 3
 */

// Library for Serial Transfer
//#include <EasyTransfer.h>
//#include <SoftwareSerial.h>
//#include <AltSoftSerial.h>

// Libraries included to use PixyCam
#include <SPI.h>
#include <Pixy.h>

// Library included to use servos
//#include<Servo.h>
#include<ServoTimer2.h>

// Libraries included to use motor and motion shield
#include <Wire.h>

Pixy pixy; //creates PixyCam object to use

// CONSTANTS AND GLOBAL VARIABLES
// Constants
enum {RIGHT=-1, NONE=0, LEFT=1, STRAIGHT=2}; // Directions
enum {GREEN=3, YELLOW=4, RED=5, HOME=6, DANCE=7, LOOP=8}; // Targets
const int APPROACH_DIST = 100; // Distance from target to start turning (inches)
const float K_P = 1.0; // Proportional constant for feedback control
const int FORWARD_VELOCITY = 255; // Pump output for normal swimming
const int TURNING_VELOCITY = 255; // Pump output for turning
const int MAX_MISSION_LENGTH = 10; // Maximum number of targets in a mission
const int CAMERA_RATIO = 1; // Distance from buoy divided by pixel width of buoy (inches/
    pixel)
const int SERVO_MIN_POSITION = 1000;//0; // Minimum angle that servos can output
const int SERVO_MAX_POSITION = 2000;//170; // Maximum angle that servos can output
const int FIN_FORWARD_ANGLE = 500; //85; // Left fin servo value for going forward (right is
     reversed)
const int FIN_TURN_ANGLE = 100; //120; // Left fin servo value for turning (right is
    reversed)
const int TUBE_ZERO_ANGLE = 1500; // Left tube servo default position for going forward (
    right is reversed)
const int TURNING_ANGLE = 2000; //170; // Angle output for initiating a turn, cutoff for
    applying valves and fins
const int MAX_BLOCKS = 7; // Maximum number of blocks sent from pixycam
const bool TURN_TUBES = true; // Whether to use tube servos for steering
const bool TURN_VALVES = true; // Whether to use valves for steering
const bool TURN_FINS = true; // Whether to use fins for steering

// Pins
const int FIN1 = 10; // Right fin
const int FIN2 = 3; // Left fin
const int TUBE1 = 9; // Right tube pull
const int TUBE2 = 5; // Left tube pull
const int VALVE2 = 2; // Left valve through relay
const int PUMPE = 4; // Pump PLL speed control pin
const int PUMPM = 5; // Pump motor plug
const int VALVE1E = 7; // Valve PLL speed control pin
const int VALVE1M = 6; // Valve motor plug

// Objects
ServoTimer2 rightFin, leftFin, leftTube, rightTube;
//AltSoftSerial Arduino(12,13); //communicate with sense Arduino RX TX
//EasyTransfer ETin, ETout;

```

```
// State variables
int direction = NONE; // Computed direction to travel
int mission[MAX_MISSION_LENGTH]; // Ordered array of targets, e.g. {RED, YELLOW, WHITE, HOME
    , NONE}
int target = 0; // Current target index
int distance = 0; // Distance from target in inches
int angle = 0; // Angle towards target in degrees CCW
long previousMillis = 0; // Previous loop time in milliseconds
boolean flood, temp = false; // E-Stop activated, hull flooding, electronics overheating
int loops = 0; // Number of times loop function is called

float widths[MAX_BLOCKS]; // Widths of detected blocks
int16_t signatures[MAX_BLOCKS]; // Colors of detected blocks
float positions[MAX_BLOCKS]; // X-positions of detected blocks
boolean estop = false; // Whether to disable the robot

//
//// Serial send/recieve structures
//struct RECEIVE_DATA_STRUCTURE{
//  //put your variable definitions here for the data you want to receive
//  //THIS MUST BE EXACTLY THE SAME ON THE OTHER ARDUINO
//  float widths[MAX_BLOCKS];
//  int16_t signatures[MAX_BLOCKS];
//  float positions[MAX_BLOCKS];
//  boolean estop;
//};
//
//
//// Give a name to the group of data
//RECEIVE_DATA_STRUCTURE rxdata;

// SETUP ROBOT CODE (RUN ONCE)
void setup() {
  // Serial transfer initialization
  Serial.begin(9600);
  pixy.init();
  //Arduino.begin(4800);
  //ETin.begin(details(rxdata), &Arduino);

  //Serial.println("In setup");

  // Pin initialization
  rightTube.attach(TUBE1);
  leftTube.attach(TUBE2);
  leftFin.attach(FIN2);
  rightFin.attach(FIN1);
  pinMode(PUMPM, OUTPUT);
  pinMode(VALVE1M, OUTPUT); // Right
  pinMode(VALVE2, OUTPUT); // Left

  Serial.println("About to system check");
  systemCheck();
  Serial.println("System check done");
}
```

The code for the Think/Act Arduino begins by declaring several constants for how to interpret missions, read the PixyCam, and output values to the Servos. A set of enums defines the different robot missions and behavior states. There is also commented out code for initializing SoftwareSerial and importing a Servo library that we tried unsuccessfully to get working before finding a better one. A few state variables are also declared to track the mission, current robot state, and observed objects. In the setup function, we initialize the Arduino pins and servos, start up the PixyCam, begin Serial communication through the XBee, and perform a system check to ensure the robot is fully functional.

4.4 Think/Act Loop

```
// ROBOT CONTROL LOOP (RUNS UNTIL STOP)
void loop() {
  delay(20);
  loops++;

  digitalWrite(PUMPM, HIGH);
  analogWrite(PUMPE, 255);

  downloadMission();
  readSenseArduino();
  think();
  act();
  if(loops%10==0) debug();
}
```

Each loop iteration, the robot tells the pump to turn on, then proceeds to check for a new mission, look for data from the Sense Arduino, decide on a course of action, and then output actuator values accordingly. Finally, the debug function is called to print data over the XBee for the operator once every ten loops.

4.5 Downloading a Mission

```
// Check for new mission over Serial in the format of a string of characters
void downloadMission() {
  int n = Serial.available();
  if(n<1) { // No message available
    return;
  }
  for (int i=0;i<n; i++) {
    // Map input characters to desired targets
    switch(Serial.read()) {
      case '>': return; // Debug message
      case '2': mission[i] = STRAIGHT; break;
      case '1': mission[i] = LEFT; break;
      case '3': mission[i] = RIGHT; break;
      case 'r': mission[i] = RED; break;
      case 'y': mission[i] = YELLOW; break;
      case 'g': mission[i] = GREEN; break;
      case 'h': mission[i] = HOME; break;
      case 'd': mission[i] = DANCE; break;
      case 'l': mission[i] = LOOP; break;
      default: mission[i] = NONE;
    }
  }
  for (int i=n;i<MAX_MISSION_LENGTH;i++){
    mission[i] = NONE;
  }
  target = 0;
}
```

This function checks if Serial has any new data, and if so parses it into a set of actions to take in order, overwriting its current mission.

4.6 Reading Sense Arduino

```
// Compute distance and direction from sense Arduino input
void readSenseArduino() {
  int n = pixy.getBlocks();
  for (int i=0; i<MAX_BLOCKS; i++) {
    signatures[i] = 0;
  }
  for (int i=0; i<min(n,MAX_BLOCKS); i++) {
    widths[i] = pixy.blocks[i].width;
    positions[i] = pixy.blocks[i].x;
    signatures[i] = pixy.blocks[i].signature;
```

```
  }
  distance = -1;
  angle = 0;
  for (int i=0; i<MAX_BLOCKS; i++) {
    if (signatures[i]==mission[target]-2) { // G,Y,R,H = 1,2,3,4
      distance = CAMERA_RATIO*widths[i];
      angle = positions[i]-159; // 159 = center of screen
    }
  }

//  if(ETin.receiveData()){ //recieves data: n, blocks
//    delay(20);
//    distance = -1;
//    angle = 0;
//    if(rxdata.estop){
//      eStop();
//  }
//
//}
}
```

This function was initially intended to obtain PixyCam data from the Sense Arduino over SoftwareSerial, but now instead reads the PixyCam values directly. Based on the size of the detected object and its x-position in the camera field, it computes the current direction and distance to the target.

4.7 Think

```
// THINK
void think() {
  if (mission[target]<=2) { // Manual override
    direction = mission[target];
  } else if (mission[target]==DANCE) { // Dance code
    direction = DANCE;
  } else if (distance<0) { // Target not visible
    direction = LEFT;
  } else if (distance>APPROACH_DIST) { // Reached target
    target++;
    if(mission[target]==LOOP) { // Restart mission
      target=0;
    }
    direction = NONE;
  } else { // Target visible
    direction = STRAIGHT;
  }
}
```

The robot's behavior varies based on its current mission. For manual control commands (turn left, turn right, go straight, stop) the desired direction is set directly to the command. However, when seeking a target the robot moves straight if it sees that target, and turns otherwise. When it gets close enough to the target, it proceeds to advance to the next target in its mission. If the mission is set to loop, the code resets back to the first target upon completing the mission.

4.8 Act

```
void act() {
  if(estop) {
    move(0,0);
    return;
  }
  switch (direction) {
    case STRAIGHT: // Swim straight using proportional feedback control
      move(FORWARD_VELOCITY, int(K_P*angle));
      break;
```

```
    case LEFT: // Turn left
      move(TURNING_VELOCITY, TURNING_ANGLE);
      break;
    case RIGHT: // Turn right
      move(TURNING_VELOCITY, -TURNING_ANGLE);
      break;
    case DANCE: // Show off your moves
      break;
    default: // Stop
      move(0, 0);
      break;
  }
}
```

Based on the direction computed by the think function, the code calls the move function with the appropriate parameters. A simple proportional feedback loop is used to steer straight by turning in proportion to the angle to the current target.

4.9 Helper Functions

```
// Delay loop
void wait(int t){
  previousMillis = millis();
  while(millis() - previousMillis <= t) {}
}

//Check all systems
void systemCheck(){
  wait(1000);
  move(0, TURNING_ANGLE);
  wait(1000);
  move(0, -TURNING_ANGLE);
  wait(1000);
  move(0, 0);
}

//eStop function to shut off all motors
void eStop(){
//  rightFin.write(0);
//  leftFin.write(0);
//  leftTube.write(0);
//  rightTube.write(0);
  digitalWrite(VALVE2, LOW);
  digitalWrite(PUMPM, LOW);
  analogWrite(PUMPE, 0);
  digitalWrite(VALVE1M, LOW);
  analogWrite(VALVE1E, 0);
}

// Output current state over Xbee
void debug() {
  Serial.print(">>> Mission: ");
  for (int i=0; i<MAX_MISSION_LENGTH; i++) {
    Serial.print(mission[i]);
  }
  Serial.print(", Blocks: ");
  Serial.print(signatures[0]);
  Serial.print(signatures[1]);
  Serial.print(signatures[2]);
  Serial.print(signatures[3]);
  Serial.print(signatures[4]);
  Serial.print(signatures[5]);
  Serial.print(signatures[6]);
  Serial.print(", Target: ");
  Serial.print(target);
  Serial.print(", Direction: ");
```

```
  Serial.print(direction);
  Serial.print(", Distance: ");
  Serial.print(distance);
  Serial.print(", Angle: ");
  Serial.print(angle);
  Serial.print(", Flood: ");
  Serial.print(flood);
  Serial.print(", Temp: ");
  Serial.print(temp);
  Serial.print(", E-Stop: ");
  Serial.println(estop);
}

// Output motor values
void move(int vel, int ang){
  // Set pump output
//  if(vel>0) {
//    digitalWrite(PUMPM, HIGH);
//    analogWrite(PUMPE, vel);
//  } else {
//    digitalWrite(PUMPM, LOW);
//    analogWrite(PUMPE, 0);
//  }

  digitalWrite(PUMPM, HIGH);
  analogWrite(PUMPE, 255);
  // Set tube angles
  if(TURN_TUBES) {
    int leftTubeAngle = min(max(TUBE_ZERO_ANGLE+ang, TUBE_ZERO_ANGLE), TUBE_ZERO_ANGLE+
        TURNING_ANGLE);
    int rightTubeAngle = min(max(TUBE_ZERO_ANGLE-ang, TUBE_ZERO_ANGLE), TUBE_ZERO_ANGLE+
        TURNING_ANGLE);
    leftTube.write(SERVO_MIN_POSITION + leftTubeAngle);
    rightTube.write(SERVO_MAX_POSITION-rightTubeAngle);
  } else {
    leftTube.write(SERVO_MIN_POSITION + TUBE_ZERO_ANGLE);
    rightTube.write(SERVO_MAX_POSITION-TUBE_ZERO_ANGLE);
  }
  // Set fin angles
  if(TURN_FINS && ang>=TURNING_ANGLE) { // Left
    leftFin.write(SERVO_MIN_POSITION + FIN_TURN_ANGLE-200);
    rightFin.write(SERVO_MAX_POSITION - FIN_FORWARD_ANGLE-100);
  } else if(TURN_FINS && ang<=-TURNING_ANGLE) { // Right
    leftFin.write(SERVO_MIN_POSITION + FIN_FORWARD_ANGLE-200);
    rightFin.write(SERVO_MAX_POSITION - FIN_TURN_ANGLE-100);
  } else { // Straight
    leftFin.write(SERVO_MIN_POSITION + FIN_FORWARD_ANGLE-200);
    rightFin.write(SERVO_MAX_POSITION - FIN_FORWARD_ANGLE-100);
  }
  // Set valve states
  if(TURN_VALVES && ang>=TURNING_ANGLE) { // Left
    digitalWrite(VALVE1M, LOW);
    analogWrite(VALVE1E, 0);
    digitalWrite(VALVE2, HIGH);
  } else if(TURN_VALVES && ang<=-TURNING_ANGLE) { // Right
    digitalWrite(VALVE1M, HIGH);
    analogWrite(VALVE1E, 255);
    digitalWrite(VALVE2, LOW);
  } else { // Straight
    digitalWrite(VALVE1M, HIGH);    analogWrite(VALVE1E, 255);
    digitalWrite(VALVE2, HIGH);
  }
}
```

To organize the code, several subtasks were split off into helper functions. These include a function that delays for a number of milliseconds, a function that runs a system check by trying to turn one way and then the other, a function to E-stop the robot, a function to print debugging information over the XBee, and a

function to output values for the jets, fins, valves, and pump based on the given velocity and turning angle. The pump control code was commented out and replaced so that the pump would always run for testing purposes.

4.10 Testing

Most of our code was initially tested on the electronics before they were put into the robot, and then again after assembly was complete. We trained the PixyCam on vision targets without the rest of the electronics setup using the bright LED targets with dim lighting for maximum reliability. However, we encountered significant issues with conflicts between the many different devices that were all trying to communicate using Serial, namely the two Arduinos, the PixyCam, and the XBee. As a result, we ended up removing large portions of the code the night before the demonstration, as is made evident the amount of commented out code. While we were eventually able to resolve these issues, we were set back a significant amount of time, and unable to work out other issues like the valves in time for the demonstration.

5 Conclusion

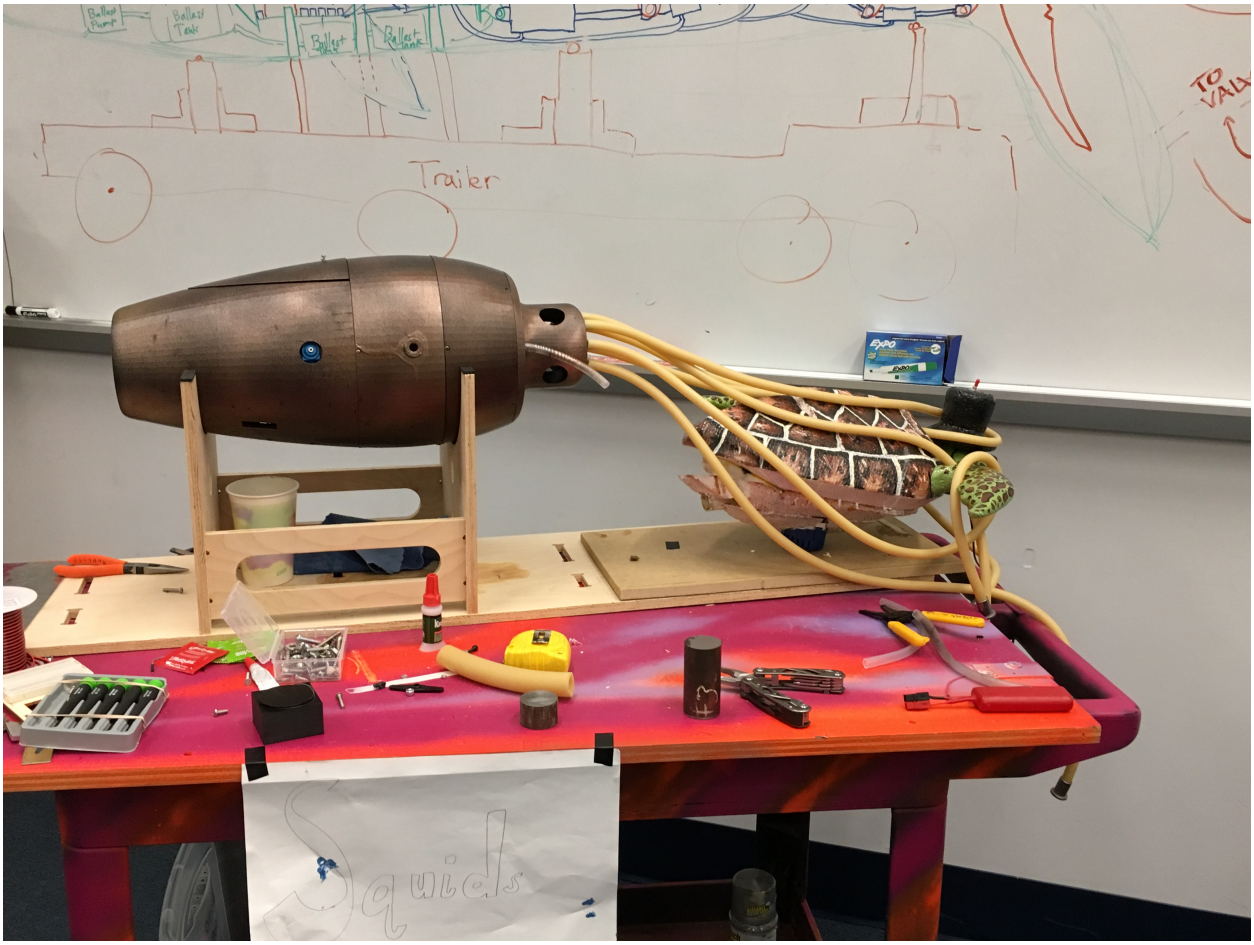


Figure 9: An image of our finished robot with tentacles attached. It is shown strangling a robot from the previous spiral.

Overall, the robot met most of the design requirements, with the exception of being able to steer properly. Due to problems actuating the valves, we were unable to steer, or even to move straight, because one of the jets was always closed. While this obviously rendered it impossible to complete the mission, with a little more time for debugging the robot has the potential to complete the mission without any significant changes to its design. Our robot did score some points for floating properly and good engineering, and was awarded extra points for aesthetics. We tried a couple of different approaches to solve the valve problem, including testing the electronics and code without the valves attached using a multimeter and powering the valves directly from batteries. The valves were getting the correct voltage, and the batteries were unable to open the valves, which points towards the problem being with the valves themselves.

The valve issue, while most likely not too hard to remedy on its own, was a symptom of insufficient time being left for testing with the finished robot. Our team finished the mechanical design fairly early in the process, but 3D printing the robot took much longer than anticipated and set us behind schedule. We also encountered significant issues interfacing with multiple Arduinos, an XBee, a PixyCam, and several different Arduino shields. While we successfully got each component working individually, we had major problems when integrating them all together, which ultimately led us to disconnect the Arduinos from each other shortly before the demo. We also experienced many difficulties sealing the electronics tube, which prevented us from testing our electronics until days before the demonstration. While all of these problems

were ultimately resolved, we did not leave sufficient time to iron out last minute issues with the electrical and software integration.

There are a number of ways we could have better implemented our robot. While we added three steering systems in the hope that at least one would work, in actuality each one just added to the amount of time we had to spend on debugging, and ultimately none of them functioned well enough to complete the mission. Additionally, we waited until the robot was fully fabricated to begin testing the servos and valves, rather than working out any issues in parallel with the mechanical fabrication. Finally, we invested a significant effort into aesthetics rather than using the time for testing. Admittedly, there was a surplus of mechanical skills on our team compared to electrical and software, and it is also hard for multiple people to test the control system at a time, so there may not have been a better use for the time spent on painting and decoration.

Despite the failure of our robot on demo day, I think our whole team found the project to be a valuable learning experience. Personally, I improved at mechanical design and creating CAD as part of a team, which requires careful planning and organization to make everything integrate easily. Additionally, I learned more about communication between embedded controllers and debugging complex systems with integrated mechanical, electrical, and software components. Finally, I learned the importance of planning ahead, sticking with a schedule, and dividing up work effectively when working on a long-term, complex team project.