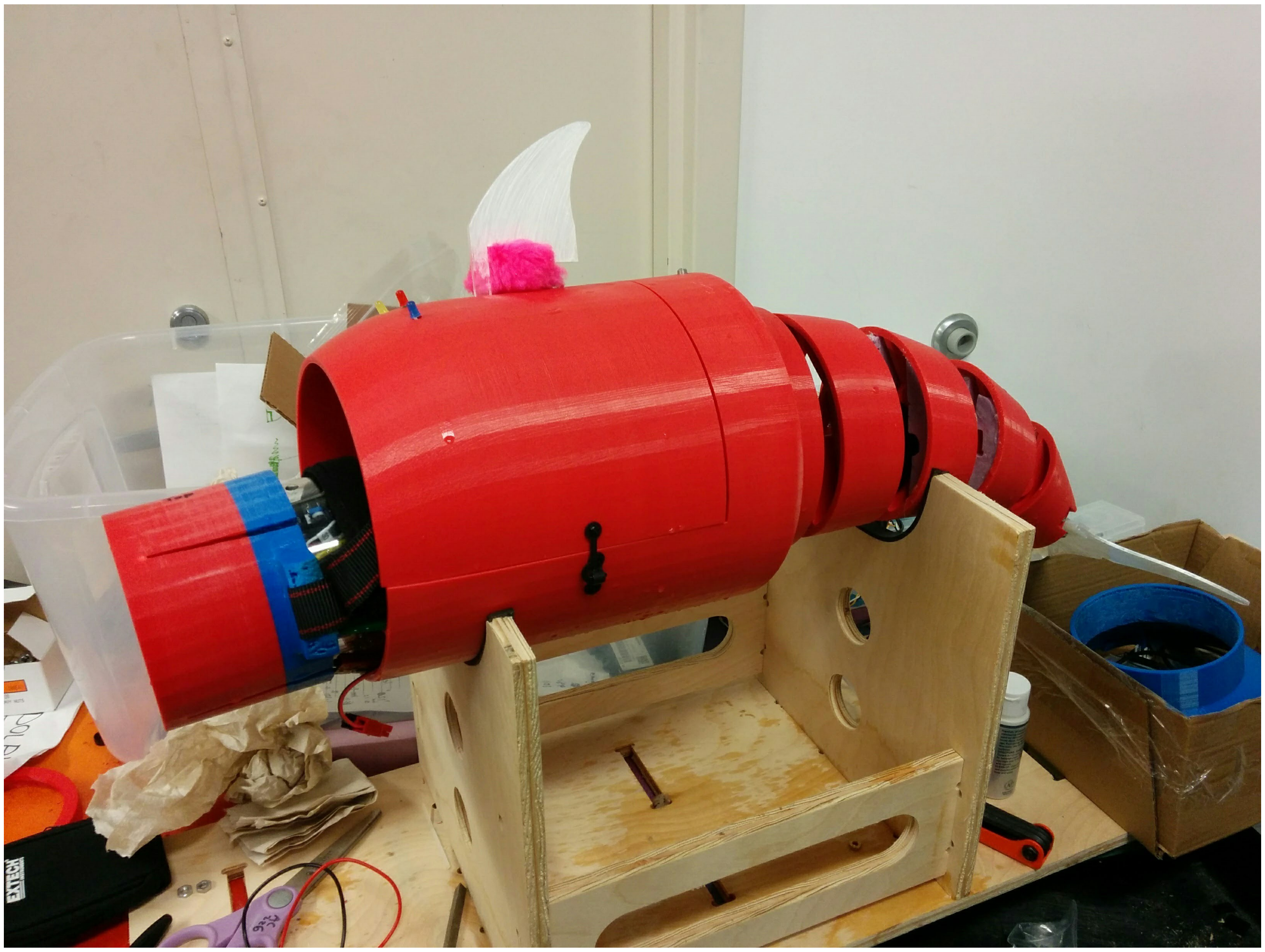
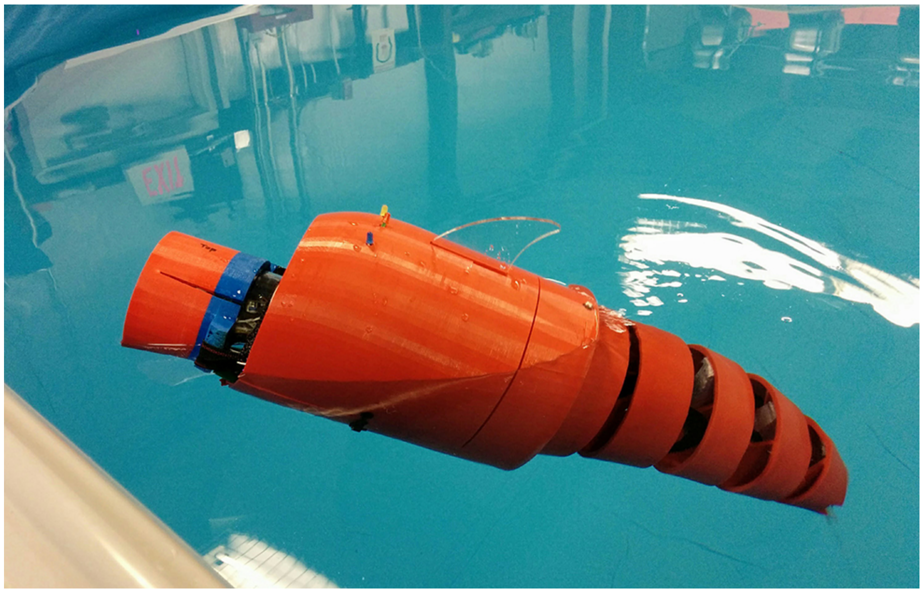
ENGR3390: Fundamentals of Robotics



Katerina Soltan
Project 2: *Flipper the Dolphin*
Team Clicks and Whistles
December 14, 2017

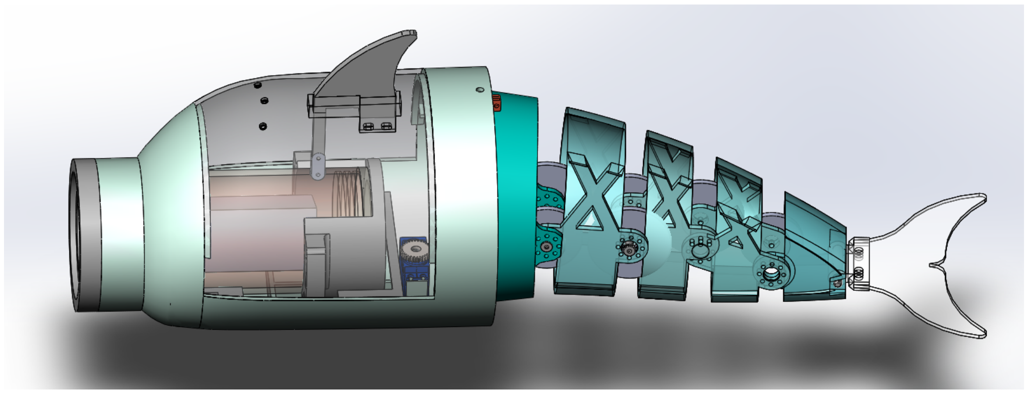
Overview

Flipper is an aquatic, dolphin-like robot designed to navigate to a sequence of different-colored buoys in a pool. The robot autonomously finds the buoy it must approach next in the sequence using a PixyCam and attempts to follow the straightest path to it. It's propulsion system is heavily inspired by that of a dolphin's tail. To swim forward, the tail oscillates smoothly up and down like a wave. To turn, the tail rotates in one plane and continues its oscillating motion.



Flipper very slowly moves the tail up and down to propel forwards.

This robot is the second project built in Fundamentals of Robotics, the ultimate goal of which is to create an autonomous underwater vehicle with biologically inspired propulsion to carry out specific tasks. The mission in the second iteration was to float just below the surface of the water and to complete a triangle path between a set of buoys in a real-time determined sequence. *Flipper* was able to traverse the pool in a straight path, exhibiting graceful, biomimetic movement. While all other subsystems pertinent to determining the location of the target and controlling the robot performed well when tested separately, despite our best efforts, we were unable to integrate them. Many of the design choices we made, especially in software, were found to have better alternatives, and we hope to include them in another iteration of the robot.

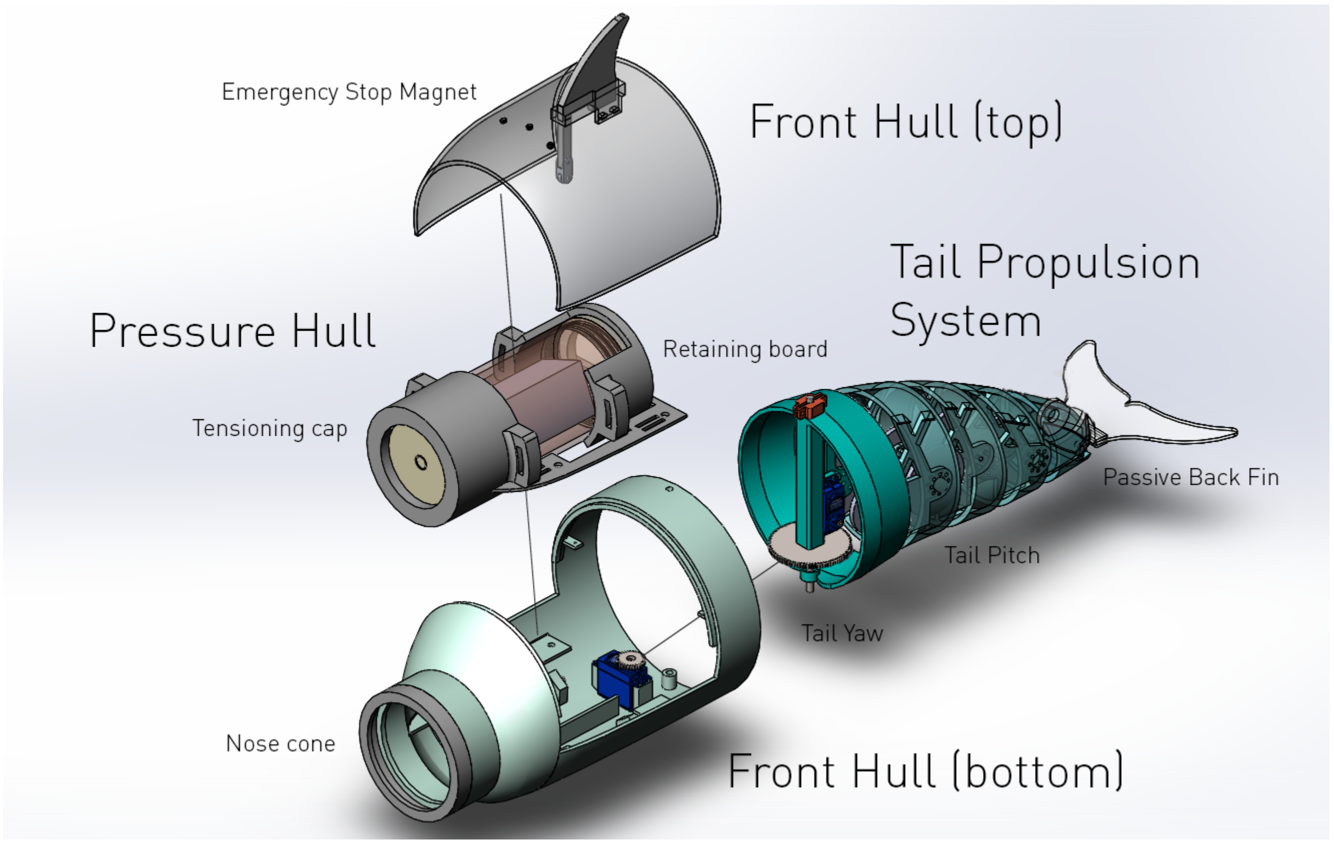


CAD assembly of *Flipper*.

Mechanical

System Design

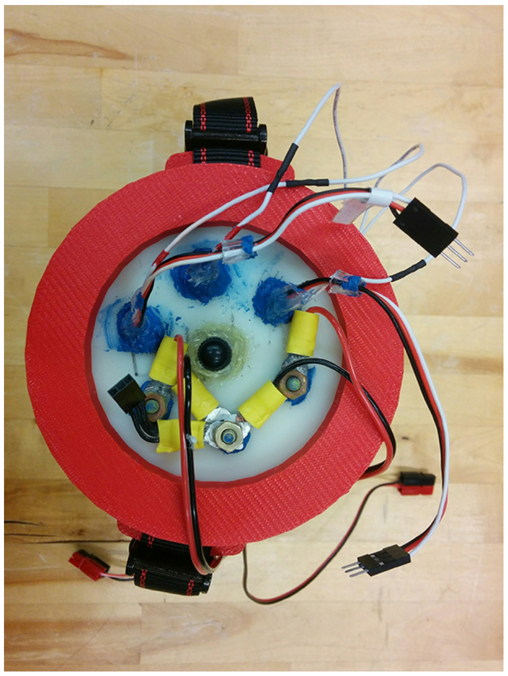
The mechanical system for *Flipper* is divided into three main parts: front hull, tail, and pressure hull. The front hull secures the pressure hull, which contains all water-sensitive electronics, and connects to the tail which provides propulsion. The robot is designed to mimic the playful shape and smooth swimming movement of a dolphin.

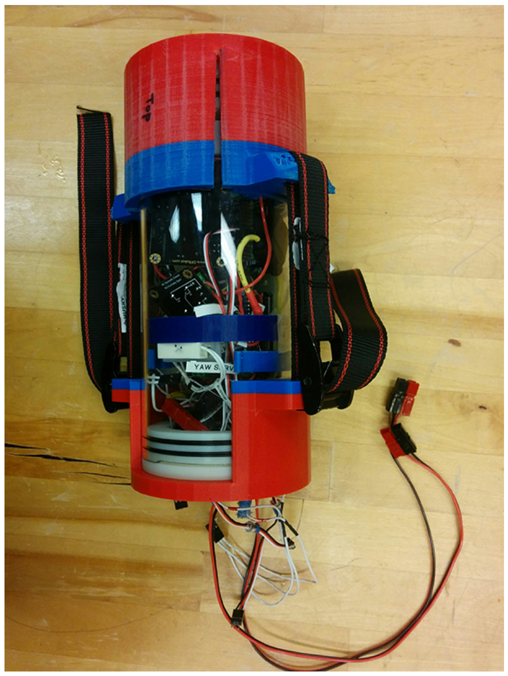


System Diagram of *Flipper*.

Pressure Hull

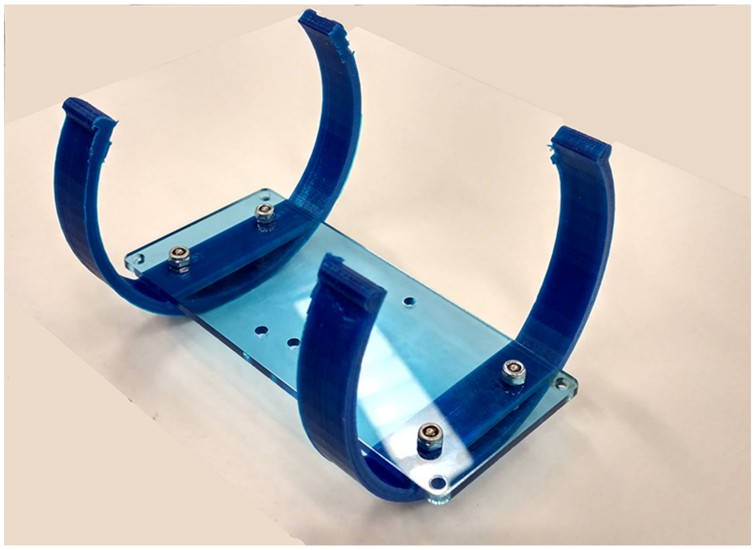
The cylindrical pressure hull contains all sensors and microcontrollers that require waterproofing. Two rubber end caps with double O-rings close the acrylic tube from both sides and prevent water leakage. A Schrader valve embedded through one of the end caps connects to a bike pump which fills the pressure hull with air. To maintain air pressure inside, the tube slides into two tensioning caps which connect with a ratchet and webbing mechanism, preventing the end caps from popping off due to the pressure. Battery power is routed to the inside of the tube through three brass screws, one for a common ground, and one for each of the power lines from the two batteries. Three threaded plastic valves route wires for two servos and signal wires for three LEDs. All valves and screws are waterproofed with aquarium sealant and silicone gasket glue. The air pressure inside of the tube helps to slow any potential leaking of water into the hull, as the air is pushing everything out of the tube.





Left: The end cap has three brass screws for battery connections, three plastic valves for servo and LED wires, and a Schrader valve for holding air pressure. Right: The pressure hull fits into two tensioning caps which are held together with ratchets and webbing to hold air pressure.

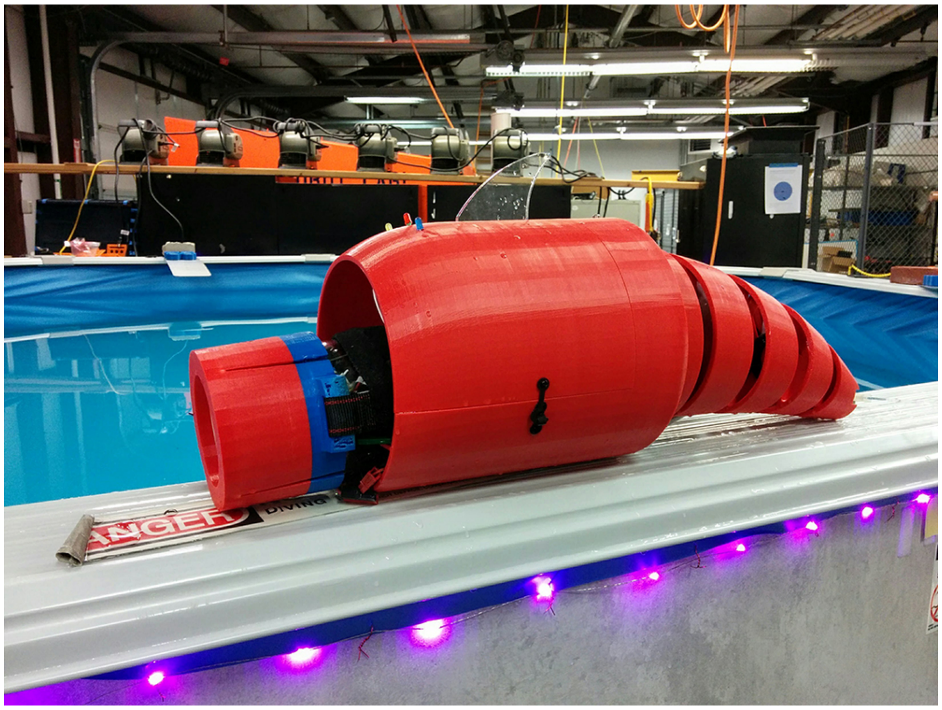
Electronics inside of the tube are mounted on a laser-cut acrylic sheet which is attached to two deformable 3D-printed arcs which snap into the tube. A separate arc attached a PixyCam, which sits against a clear acrylic window machined inside of the second end cap for an unobstructed view. Another arc holds a reed switch which aligns with the dorsal fin magnet and allows power to the servos. The pressure hull sits on a laser-cut retaining board that is bolted to platforms along the sides of the front hull. Velcro straps secure the pressure hull to the board and are easily removed to release it.



Deformable arcs snap into the inside of the pressure hull. A laser cut mounting board fits between them for attaching electronics.

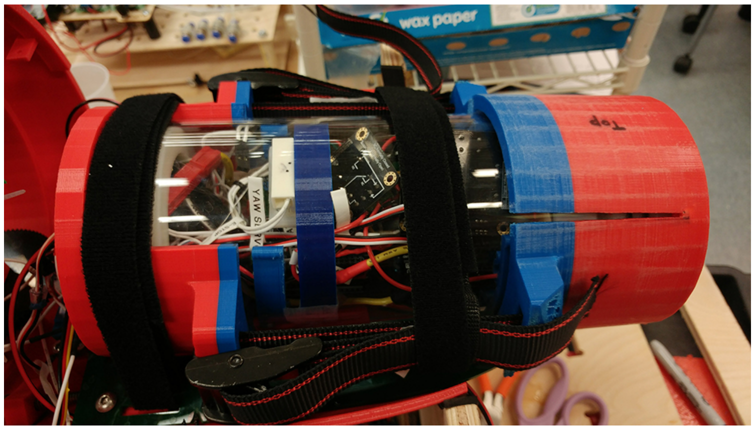
Front Hull

The front hull is inspired by the barrel shape of a dolphin. Our main design constraint was the length of the pressure hull which we wanted to minimize to accommodate the extensive tail. The pressure hull is situated below the centerline of the body to account for the large buoyant force that it exerts, as it is filled with air. Because we expected the robot's nose to be elevated with respect to the rest of the body, threaded rods are mounted on both sides of the bottom for the addition of heavy nuts for weight tuning. Battery holders are also located at the bottom of the hull.



The top and bottom half of the hull are held together with latches for swift disassembling. The dorsal fin contains a magnet which acts as an emergency stop.

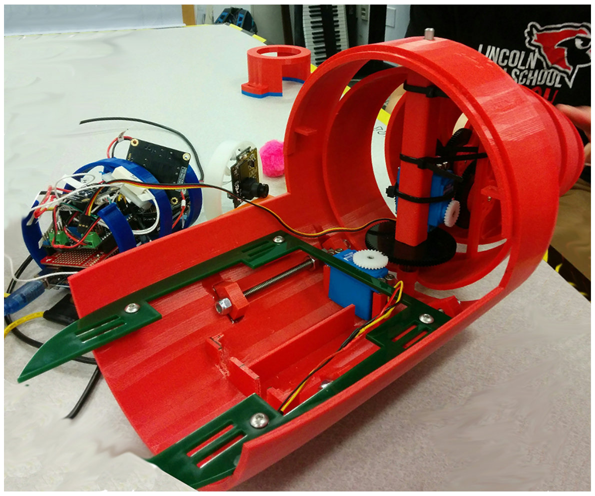
The hull is fully 3D printed with ABS filament; the top and bottom half are held together with metal latches, providing a quick-access point to the inside of the dolphin. A magnet is embedded in the dorsal fin and acts as an emergency stop; pulling the dorsal fin up and out stops the flow of current to the motors. The open nose cone provides an unobstructed view for the camera located inside of the pressure hull.



The pressure hull is strapped to the inside of the front hull with velcro straps.

Tail Propulsion

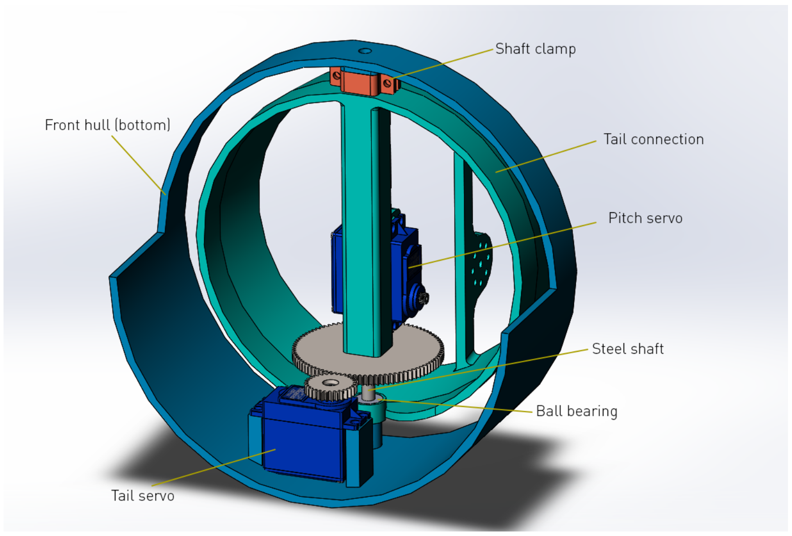
The propulsion system consists of two servos which independently drive two parts: pitch and yaw. An intricate gear train replicated a sinusoidal motion in the tail, pitching up and down. The yaw portion of the tail rotated the entire tail right and left, mimicking the natural rigid motion of a dolphin's tail when turning.



The yaw servo is mounted behind the tube retaining board, connecting to a gear on a tail connection link which holds the pitch servo.

Tail Yaw

The yaw servo sits on the bottom of the front hull and drives a tail connection link at a 3:1 reduction ratio. The connection link slides onto a steel shaft which fits into cylindrical stopper, also at the bottom of the hull. A small squeeze clamp located between the top of the hull and the connection link prevents the shaft from moving up and out of the hull. To ensure the alignment of the shaft, the hull was not split directly down the center and, instead, was printed with a full circular cross section at the very back where the shaft connects. The connection link rotates around the shaft and contains two ball bearing (one on top, one on the bottom) embedded in it for smooth rotation. The tail servo is supported by the connection link, allowing the entire tail system to rotate together.



The yaw subsystem consists of a servo driving a tail connection link about a rigid shaft.

Tail Pitch

To replicate the the smooth oscillatory motion of the tail, a gear train powered by one servo extends across the sectioned tail. Eleven gears are used to achieve the desired movement across five sections. One driving gear on the servo translates rotation to a fixed gear on the first link. Another fixed gear on the opposite side of the link provides a reduction to the fixed gear on the next link which moves along it. The last link connects to a passive tail fin which provides thrust on forwards and upwards path, pushing against the water with its large cross-sectional area. Gears were fixed using small pins and hot glue to prevent slipping.

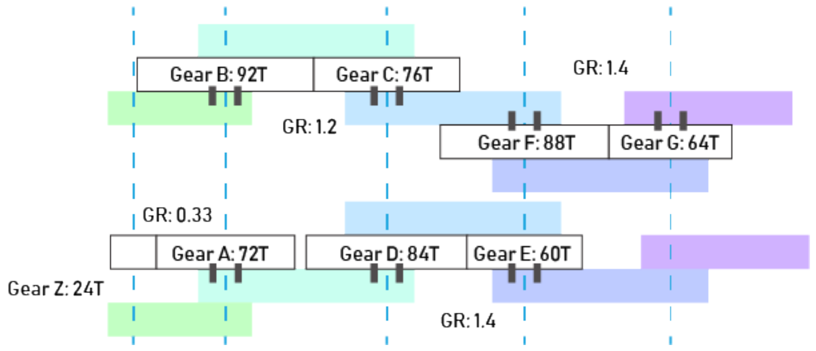
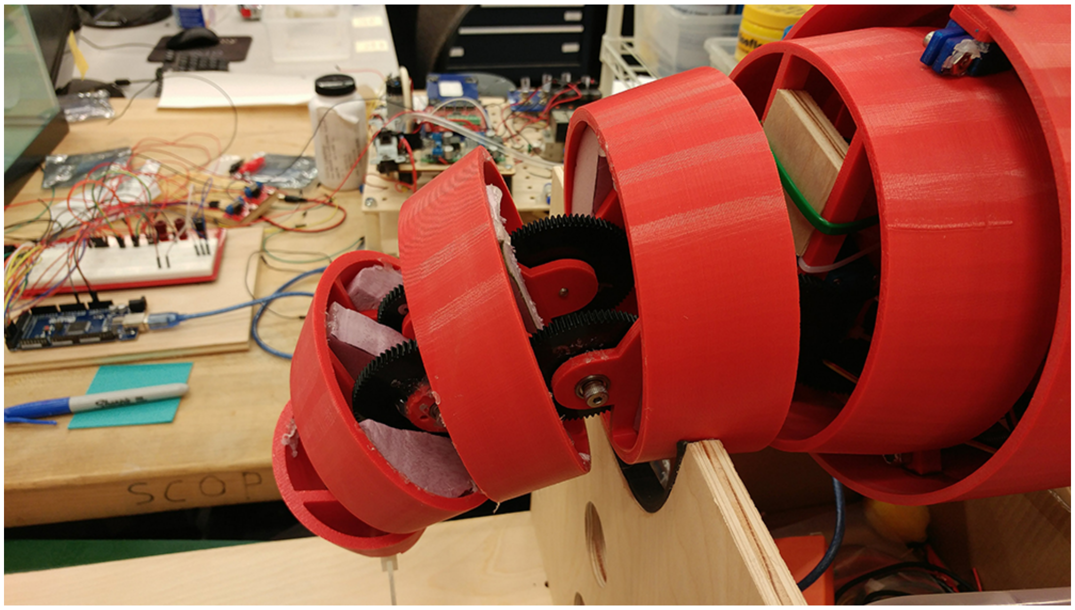


Diagram of the 11 gears and their positions relative to the tail links (represented by colors rectangles). Black pins represent gears fixed to a link.

The tail system contains many bearings and shoulder screws for gear rotation, making it too heavy for the servo to move, even in water. The positively buoyant pressure hull floats to the surface, causing the tail to sink downwards. To alleviate the stress on the servo and to level the dolphin body, large pieces of pink foam were cut to fit inside of the first three links. The servo was able to successfully move the tail in this configuration, although the nose of the robot still pointed upwards.



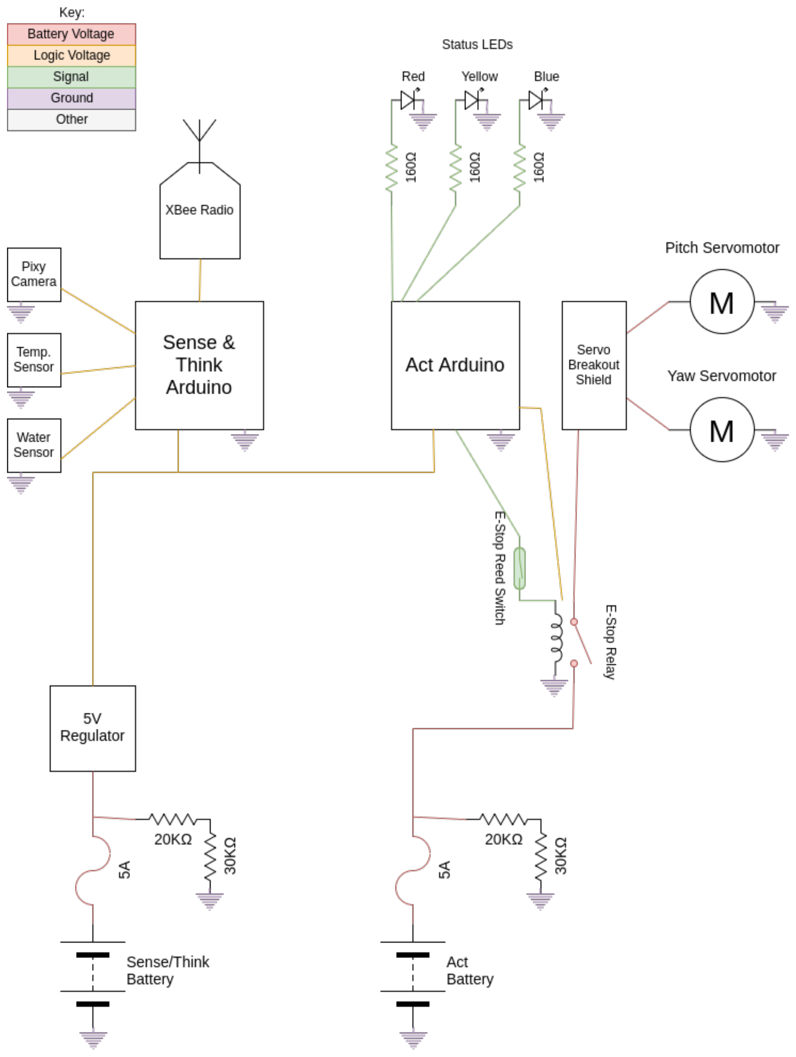
Foam was placed inside of the tail to make more neautrally buoyant and easier to move.

Electrical System Design

Flipper's electrical system consists of two Arduino, representing the thinking and acting aspect of the brain, a couple of sensors to monitor the environment inside of the pressure hull (overheating or water presence), a PixyCam camera for visual feedback from the pool, two servos, one for yaw and one for pitch, and two batteries.

Power

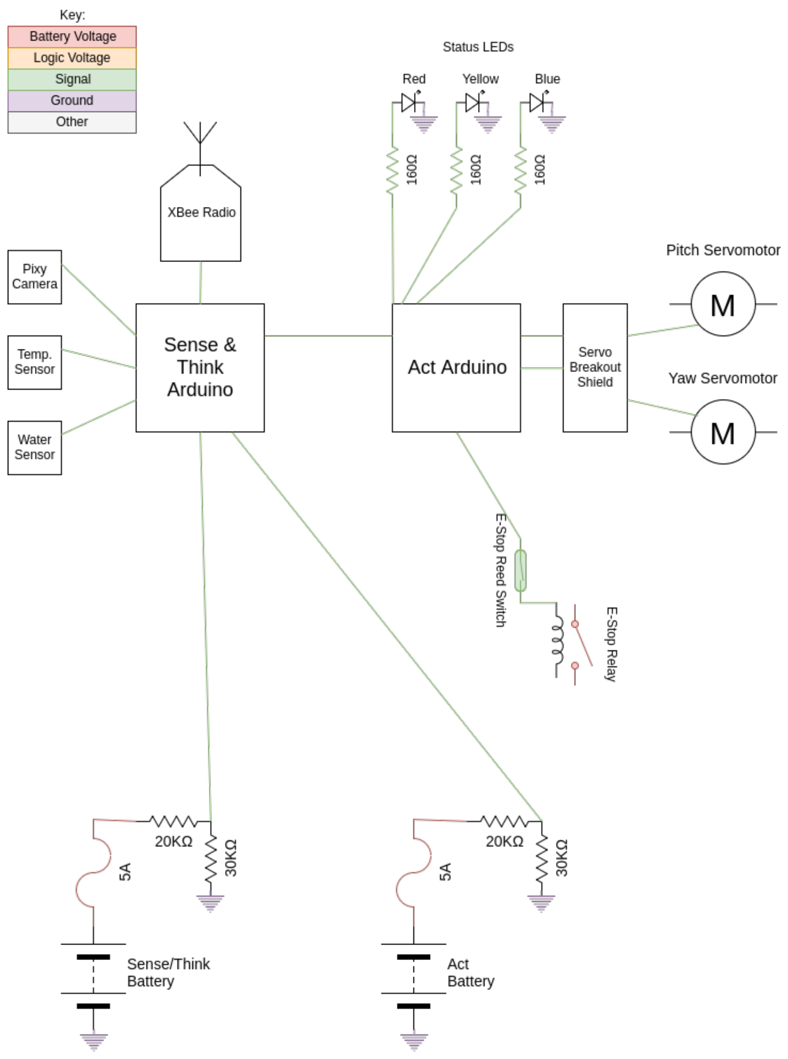
Two 7.2V NiMH batteries power the robot; one is a dedicated supply for the microcontrollers and sensors while the second is reserved for servos. Fuses (5A) are installed in line with the batteries to prevent current overdraw damaging the rest of the system. The ground wires of the batteries are coupled to create one mother ground line. The batteries are places outside of the pressure hull and connect to the electronics inside via brass screws. The yaw and pitch servos as well as the LEDs attach to the signal wires routed through plastic valves.



Power flow diagram.

Sense/Think Arduino

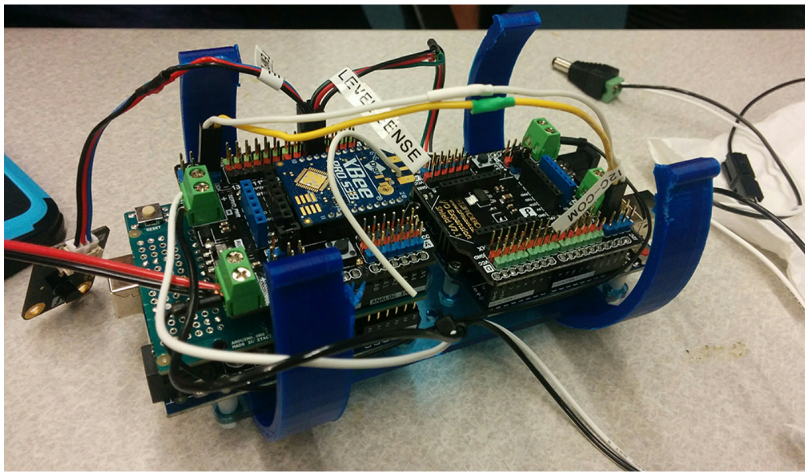
The sense/think Arduino represent the sensing and decision making part of a robot brain. A temperature sensor, water level sensor, a PixyCam camera, and the emergency stop are connected to this Arduino for quickly taking readings and incorporating them into the control loop. The sensors and emergency stop (a magnetically activated reed switch) attach to an I/O breakout shield. The Arduino communicates with a ground control laptop through an XBee radio module. The XBee receiver is mounted on a Sparkfun breakout board on top of the sense/think Arduino, while the transmitter is connected to the laptop. One of the batteries is dedicated to powering the sense/think portion of the electronics, using a 5V DC/DC converter for a smooth input.



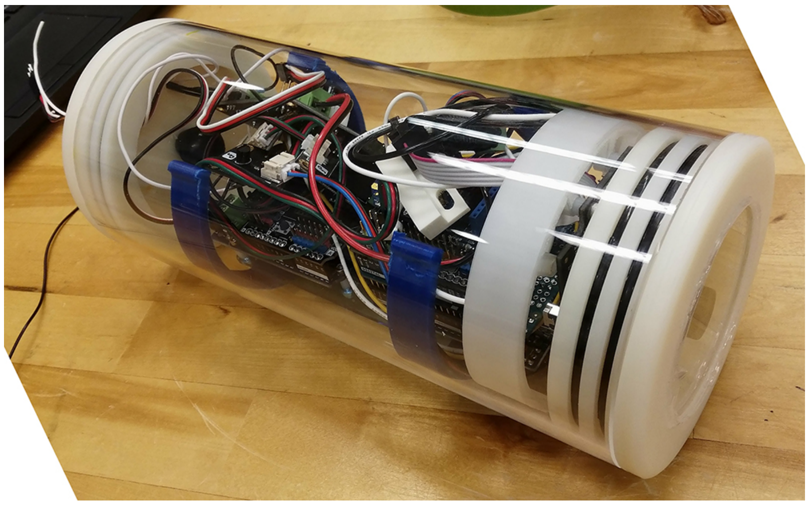
Data flow diagram.

Act Arduino

The yaw and pitch servos are connected directly to it through another I/O board. For safety, servo power is separated from the act Arduino power by cutting the diode between input servo and logic power. Servo power is routed from the second battery through a normally open relay controlled programatically by the sense/think Arduino; if there is no power on this line, the relay will remain open, and no power will flow to the servos. The reed switch provides a mechanical emergency stop capability. It is wired in-line between the sense/think Arduino pin and the relay; thus, if the magnet is not closing the reed switch, even if the Arduino is programatically allowing the relay to close, it will remain open until the dorsal fin with the magnet is lowered.



Arduinos and their shields were mounted to the flexible arcs for easy placement inside of the pressure hull.



All of the electronics were fit inside of the pressure hull and sealed.

Software System Design

The sense/think Arduino's job is broken up into five tasks. The first is to communicate with the ground control laptop via XBee to receive a mission or user commands. The second is to read the PixyCam output and to determine whether it sees the buoy that it is looking for. Next, the dolphin robot's state must be updated to reflect whether the robot is in searching mode or approaching mode. Based on this mode, the motion parameters are calculated. Finally, these parameters are sent to the act Arduino through I2C. The act Arduino simply receives this transmission and writes the given values to the servos.

Sense/Think

Mission Communication

The XBee is used to communicate between the ground station laptop and the sense/think Arduino. It is setup on pins 2 and 3 with SoftwareSerial (the Sparkfun XBee shield uses these), leaving the pins Serial Monitor uses open for debugging.

```
/**
 * DolphinVariables.h
 * Purpose: Defines state and mission variables, as well as XBee port which are used across
     multiple header files in the project.
 * Notes: Include this file before any other header files to avoid compilation errors of
     undefined variables.
 */

// Define XBee: Sparkfun XBee board uses pins 2 (DOUT - RX) and 3 (DIN - TX).
// Use Software Serial to leave Serial Monitor open for debugging.
#include <SoftwareSerial.h>
SoftwareSerial XBee(2, 3); // RX, TX

// Defines 5 robot states of the dolphin.
// Enum states can be used as integers, with Standby = 0...Helpme = 6.
enum robotState {
  STANDBY, // Waiting for mission
  SEARCH, // Searching for next target
  APPROACH, // Approaching target
  VICTORY, // Transitional state to update mission, or wait for new one.
  HELPME // Sensors picked up problem in robot, such as flooding, overheating, or e-stop.
};

// declare dolphinState as a robotState data structure
enum robotState dolphinState;

// Mission definition variables
bool hasMission = false; // The mission comes from the computer
String mission = ""; //r for red, y for yellow, w for white
int lengthMission = 0;
int current_mission_step = 0; // 0 is the first step of the mission, increment until
    lengthMission - 1

// Approach Timing: From different angles, buoys look different sizes due to LED
    brightnesses and dispersion. Allow robot to approach buoy for a set amount of time
    before comparing size of buoy to target size.
boolean approaching_timer_set = false; // Did the robot go into approach state for this buoy
long approach_start_time = 0; // time robot entered approaching mode for current buoy
long min_approach_time = 10 * 1000L; // Approach for 10s before considering buoy's proximity
    .

// Reset mission variables to be ready for new mission.
void resetMission(){
  hasMission = false;
  mission = "";
  lengthMission = 0;
  current_mission_step = 0;
}
```

PixyCam Logic

The PixyCam is sampled once every 500ms. Sampling it continuously (as fast the main loop executes) does not allow the camera enough time to process the image and it will often switch rapidly between seeing and not seeing an object that has not moved from its position in front of it. The PixyCam pre-processes images to find the positions rectangular blocks of colors which it has been trained to recognize. For this mission, the camera was trained on submerged red, green, and blue LED arrays. During testing, red, blue, and yellow balls were used (thus the naming of the color codes).

```
/**
 * PixyLogic.h
 * Purpose: Defines the PixyCam object. Reads the PixyCam input, determines whether it can
     see the correct buoy and where it is. Decides whether the buoy is close enough.
 * Notes: The area of the block the PixyCam must see to decide that the buoy is close enough
     varies hugely with where the robot is in the pool. The value currently set may need
     calibration.
 */
#include <SPI.h> // Serial Peripheral Interface: the way we communicate with the our PixyCam
#include <Pixy.h>

Pixy pixy; // This object handles the pixy cam

// Pixycam vision variables
bool canSeeMissionBuoy = false;
int buoyX = -1, buoyY = -1; // Position of the buoy: X is 0 to 319, Y is 0 to 199.
                            //-1 indicates we don't know.
int CLOSE_BUOY_AREA = 1000; // Change to tune how close robot comes to buoy before turning
bool missionBuoyIsClose = false; // True: reached the buoy and can switch to a new target.

long lastTimePixySampled = 0; // Do not sample the pixy too fast, to avoid switching between
     Search to Approach rapidly
int pixySampleTime = 500; // Read pixycam every 500 ms.

// From Arduino API section in http://www.cmucam.org/projects/cmucam5/
// wiki/Hooking_up_Pixy_to_a_Microcontroller_(like_an_Arduino)
#define X_MAX 319 // maximum horizontal position on pixycam. Min is 0.
#define X_CENTER X_MAX / 2 // horizontal center of the pixycam
#define Y_MAX 199 // maximum vertical position on pixycam. Min is 0.

void setupPixy() {
  pixy.init();
}

int getCharPixyColor(char c){
  switch(c){
    case 'r':
      return 1; // Pixy trained on red as signature 1
    case 'y':
      return 2; // Pixy trained on yellow as signature 2
    case 'w':
      return 3; // Pixy trained on white as signature 3
    default:
      return -1; // This color doesn't exist
  }
}
```

The sense/think Arduino processes the output of the PixyCam by determining if any of the blocks it has detected are of the color that it is currently searching for. If there are such blocks, it chooses the one with the largest area and sets its position to be the target. It also changes the variable canSeeMissionBuoy to true which can trigger a state change from Search to Approach. To determine whether the buoy is close enough to consider victory, a maximum area threshold is set; if the area of the block with the correct color is greater, victory is achieved.

```
bool readPixyCam() {
  int blockCount = pixy.getBlocks(); // Number of blocks detected
  missionBuoyIsClose = false;
  canSeeMissionBuoy = (blockCount > 0);
  int maxIndex = -1; // Define as -1 in case no detected blocks have the correct color
  int maxArea = 0;
  if (canSeeMissionBuoy) { // If there are potential buoys
    for (int i = 0; i < blockCount; i++) {
      // Only check blocks that are of the color that is being searched for
      if(pixy.blocks[i].signature == getCharPixyColor(mission[current_mission_step])){
        // Choose the block with the largest area to be the supposed buoy
        int area = pixy.blocks[i].width * pixy.blocks[i].height;
        if (area > maxArea) {
          maxArea = area;
          maxIndex = i;
        }
      }
    }
  }
  // If we have detected a buoy, decide whether we are close enough to the buoy as victory
  if(maxIndex > -1){
    buoyX = pixy.blocks[maxIndex].x;
    buoyY = pixy.blocks[maxIndex].y;
    if(millis() - approach_start_time >= min_approach_time){
      // Could potentially be close to a buoy, compare area
      missionBuoyIsClose = (maxArea > CLOSE_BUOY_AREA) ? true : false;
    }
    digitalWrite(13, HIGH);
    if(missionBuoyIsClose){
      XBee.println("Mission buoy is close");
    }
  }
  else {
    digitalWrite(13, LOW);
    canSeeMissionBuoy = false;
    buoyX = -1;
    buoyY = -1;
  }
  return canSeeMissionBuoy;
}
```

Updating the State

Because the sense/think Arduino's job has five distinct parts, the code is broken up into header files that are dedicated each to one of the parts. The main ino file includes all of these files, calls the setup functions for all components in the main setup(), and depending on if the system sensors are okay, sets the state of the robot to Standby.

```
/**
 * SenseThinkDolphin.ino
 * Purpose: Code for the sense/think Arduino of team Clicks and Whistles robotic dolphin.
 * Notes: The full code is broken up into header files.
 *        This file is reserved for updating the state of the robot based on sensor and
 *        camera input, as well as updating the current step in the mission.
 * Author: Katya Soltan
 */

#include "DolphinVariables.h" // Handles dolphin state and mission variables that are
                              // used between modules (pixy/mission com/act params)
#include "SensorLogic.h" // Handles sensors and system check functions
#include "PixyLogic.h" // Handles pixycam and image processing
#include "ActCommunicationLogic.h" // Handles I2C communication with ACT Arduino
#include "MissionCommunicationLogic.h" // Handles download of mission from XBee

// check that systems are okay; assign initial state accordingly
```

```
void setup() {
  setupPins();
  setupPixy();
  setupI2C();
  setupMissionCom();

  if(!areSystemsOK()){
    dolphinState = HELPME;
    printDolphinState();
  }
  else{
      dolphinState = STANDBY;
      printDolphinState();
      XBee.print("Input Mission (r-red, y-yellow, w-white): ");
  }
  freeze_motors(); // Make sure servos are off at the beginning of mission.
}
```

In the main loop of execution, the sense/think Arduino attempts to receive a mission from the XBee, if it is in Standby mode. Once it has a mission, it will check if the user has sent a command over the XBee to override its current state. Then, it reads the Pixycam and depending on the output, updates the state of the robot. Lastly, it sends an updated set of act parameters to the act Arduino. It repeats this loop indefinitely.

```
void loop() {
  if(dolphinState == STANDBY){
    hasMission = downloadMission();//attempt to download mission here with XBee
    if(hasMission){ // when get one, start searching
      dolphinState = SEARCH;
      release_motors(); // Allow power to the motors.
      current_mission_step = 0;
      XBee.print("\nMission recieved: ");
      XBee.print(mission);
      XBee.println();
      printDolphinState();
    }
    else return; // Otherwise keep waiting for mission, remain in STANDBY mode
  }

  communicateWithXBee(); // Check if there is a message from the user that overrides the
      state of the robot

  if(millis() - lastTimePixySampled >= pixySampleTime){ // Sample PixyCam at slow rate to
      allow for processing time and to smooth the signal
      readPixyCam();
      lastTimePixySampled = millis(); // Update time pixy sampled.
      updateDolphinState();

  }
  sendActParams(); // Ping the ACT Arduino with state and servo positions
}
```

The dolphin has five separate states. In Standby mode, it is not moving and is waiting for a mission to be transmitted over the XBee. Once it receives the mission, the robot enters Search mode. It turns in a circle until it detects the buoy of the correct color for the first step in the mission. Once it detects the buoy, it switches into Approach mode; it swims directly at the buoy. If it loses the buoy, it returns into Search mode. Otherwise, once it determines the buoy is close enough, it switches into Victory mode. If there is another step in the mission, it returns into Search mode, updating the color of the buoy it is searching for. If it has no more buoys to search for, it enters Standby mode and waits for another mission.

```
void updateDolphinState(){
  // THINK: Figures out which state the robot should be in.
  if(!areSystemsOK()){
    dolphinState = HELPME;
    freeze_motors();
  }

  if(dolphinState == SEARCH){
    if(canSeeMissionBuoy){
      dolphinState = APPROACH;
      if(!approaching_timer_set){
        approach_start_time = millis(); // update time of last victory
        approaching_timer_set = true; // Have set the timer from the time it's approaching.
        XBee.println("Set timer.");
      }
    }
    if(missionBuoyIsClose){
      dolphinState = VICTORY;
      approaching_timer_set = false;
      XBee.println("Reset timer.");
    }
  }

  else if(dolphinState == APPROACH){
    if(missionBuoyIsClose){
      dolphinState = VICTORY; // Use this state to update mission.
      approaching_timer_set = false;
      XBee.println("Reset timer.");
    } else if(!canSeeMissionBuoy){
      dolphinState = SEARCH;
    }
  }

  else if(dolphinState == VICTORY){
    incrementMissionTarget();
    if(current_mission_step == -1){
      dolphinState = STANDBY;
      resetMission();
      freeze_motors();
      XBee.print("Input Mission (r-red, y-yellow, w-white): ");
    }
    else{
      dolphinState = SEARCH;
      XBee.print("Searching for ");
      XBee.print(mission[current_mission_step]);
      XBee.print(" buoy.\n");
    }
  }
  printDolphinState();
}

void printDolphinState(){
  switch(dolphinState){
    case STANDBY:
      XBee.println("STANDBY");
      break;
    case SEARCH:
      XBee.println("SEARCH");
      break;
    case APPROACH:
      XBee.println("APPROACH");
      break;
    case VICTORY:
      XBee.println("VICTORY");
      break;
    case HELPME:
    default:
      XBee.println("HELPME");
      break;
  }
}
```

To update the mission to the next step, the Arduino checks whether there are any buoys left for it to find in its current mission. If so, it will trigger a return to Standby mode. Otherwise, it increments the step in the mission and causes a switch to Search mode.

```
// Update the target buoy when one is found.
void incrementMissionTarget(){
  current_mission_step++;
  if(current_mission_step >= lengthMission){
    current_mission_step = -1; // Indicate mission is over.
  }
}
```

Calculating Act Parameters

For each given state, there is a set movement for the servos. Each servo was calibrated to determine its maximum up/down or right/left positions. Every ten milliseconds, the Arduino recalculates all servo position parameters to send to the act Arduino. Within these ten seconds, if the yaw or pitch servo's positions need to change, they will update and the new value will be included in the transmission. This setup requires three timers, one for what frequency to check for updates in positions, and one each for the yaw and pitch servo position calculators.

```
/**
 * ActParams.h
 * Purpose: Defines the yaw and pitch servo position patterns for each state at a given time
    .
 * Warning: The timing for updating act parameters does not play well with the timing of
    each of the act parameter calculators. Consider moving this code to the Act Arduino.
 */
// Servo Movement Timing Variables
long unsigned lastTailMoveTime = 0; // Last time we calculated tail servo positions.
long unsigned lastUpdateTime = 0;   // Keeps track of the last time at which we checked to
                                    //update act parameters (yaw and tail positions).
long unsigned lastYawMoveTime = 0;

int updateDelayTime = 10; // Every couple of ms recalculate tail/yaw positioning. This value
                          // has to be low if you want the tail to flap at a very high
                          // frequency, because otherwise, it doesn't update fast enough.

// Time delay must be smaller than time step for moving the servo at a constant period
int fastFlapPeriod = 1.8 * 1000; // 1 full cycle in x milliseconds
int slowFlapPeriod = 2 * 1000; // 1 full cycle in x milliseconds

// Servo Positions: Look into making sure that the servoLeft/Right are
// not negative, since transmitting unsigned bytes.
int yawServoLeft = 30; // degrees  Calibrated with test_Servo_Pos in Arduino folder.
int yawServoRight = 130; // degrees

int tailServoLeft = 65; // degrees (Actually down)
int tailServoRight = 140; // degrees (Actually up)
int servoAngleChange = 2; // Smallest value by which to increment servo position
int yawServoPos = (yawServoRight + yawServoLeft) / 2; // Initialize to the midpoint.
int tailServoPos = (tailServoRight + tailServoLeft) / 2; // Initialize to the midpoint.
int tailDir = 1; // If 1: yaw servo is moving to the right (incrementing). If -1: yaw servo
                 // is moving to the left (decrementing)

int numStepsInTailCycle = 2 * abs(tailServoRight - tailServoLeft) / servoAngleChange;
  // Number of steps to complete full period (up, down, up) motion for
  // the tail with servoAngleChange update

int numStepsInYawApproachAdjustment =
  2 * abs(yawServoRight - yawServoLeft) / servoAngleChange;
  // Number of steps to complete full sweep (right, left) motion for
  // yaw with servoAngleChange update

int yawDir = 1;
```

To achieve the biomimetic dolphin tail motion, the pitch servo moves smoothly up and down at a set frequency. Once it gets to the maximum right or left position, it switches direction and continues its motion. Each update step moves the servo 2 degrees for a smooth motion. The function updateTailPosition() takes a period that the tail should achieve, calculates the time step required to move completely right and left with the angle change to achieve this period, and if it time to move, updates the pitch (tail in the code) servo position.

```
// Moving the tail position at a constant frequency up and down
void updateTailPosition(int period){
  int timeToMove = period / numStepsInTailCycle; // Need to move servo every x milliseconds
                                                 // to achieve this frequency
  if(millis() - lastTailMoveTime >= timeToMove){ // Update servo direction only if correct
                                                 // amount of time has passed

    // Check to see if the servo will move out of bounds with current direction, either
    // left (too much decrement) or right (too much increment)
    if(tailServoPos + tailDir * servoAngleChange > tailServoRight ||
        tailServoPos + tailDir * servoAngleChange < tailServoLeft) {
      // Switch direction
      tailDir *= -1;
    }

    tailServoPos += tailDir * servoAngleChange; // Update tail servo pos
    lastTailMoveTime = millis();
  }
}
```

In Approach mode, the robot attempts to swim in a straight line towards the target. Because drift is inevitable, the yaw servo must compensate for th motion by moving right or left. The function updateYawPosition() will move the servo to the right or left at a constant period until the target is in the middle of the PixyCam frame.

```
void updateYawPosition(int period){ // Moves the yaw servo slowly to the right or to the
    left in Approach mode
  int timeToMove = 70; // Update by servoAngleChange every 70ms.
  if(millis() - lastYawMoveTime >= timeToMove){ // Update servo direction only if correct
                                                // amount of time has passed

    // Check to see if the servo will move out of bounds with current direction, either
    // left (too much decrement) or right (too much increment)
    if(yawServoPos + yawDir * servoAngleChange > yawServoRight ||
        yawServoPos + yawDir * servoAngleChange < yawServoLeft) {
      //Don't move.
  } else{
    yawServoPos += yawDir * servoAngleChange;
  }
    lastYawMoveTime = millis();
  }
}
```

In Standby mode, both servos are in the neutral position (halfway between their extremes). In actuality, the servos do not get power when the robot is waiting for a mission because the relay is turned off in the change of state code.

```
void getStandbyActParams(){
  yawServoPos = (yawServoRight + yawServoLeft) / 2; // Initialize to the midpoint.
  tailServoPos = (tailServoRight + tailServoLeft) / 2; // Initialize to the midpoint.
}
```

In Search mode, the robot moves the tail up and down for propulsion while moving the yaw servo completely to the right to achieve a circling motion which allows it to turn around and locate the buoy.

```
void getSearchActParams(){
  yawServoPos = 0; // stays constant.
  updateTailPosition(fastFlapPeriod);
}
```

During Approach mode, the tail continuous its undulatory motion while the pitch servo gets updated if the center of the buoy is not in the center of the frame of the PixyCam. If it is in the center, the yaw servo stays in its position.

```
// When the robot is approaching, want to keep buoy centered in vision
// Can achieve by beating the tail at a reasonable frequency and adjusting yaw continuously
void getApproachActParams(){
if(X_CENTER - buoyX > 10){ // Buoy is to the left of center
    yawDir = -1; // Compensate left
    updateYawPosition(slowFlapPeriod);
  }
  if(buoyX - X_CENTER > 10){ // Buoy is to the right of center
    yawDir = 1; // Compensate right
    updateYawPosition(slowFlapPeriod);
  }
// otherwise, the buoy is straight on, do not change yaw. We can also change the statements
// above to givemore leeway. Say, if the center of the blob is within 10 of the center,
// keep going straight.

  updateTailPosition(fastFlapPeriod);
}
```

Because the Victory mode is currently just a transitional mode, the yaw and pitch servo positions stay constant during this state. The same applies to the Helpme state because the servos should not be getting power.

```
// For now, the victory transition state can stay in its exact position
// Can do a funky thing when the mission has been ended later.
void getVictoryActParams(){
  yawServoPos = yawServoPos;
  tailServoPos = tailServoPos;
}

// Helpme mode has to press the estop, so the servo positions will not change.
void getHelpmeActParams(){
  yawServoPos = yawServoPos;
  tailServoPos = tailServoPos;
}
```

The act parameters are updated at a constant time interval. Instead of flooding the act Arduino with servo positions which may not be changing, it would be a better design to only update the parameters when there has been a change in them.

```
boolean getActParams(){ // Return true if it is time to send act parameters
  if(millis() - lastUpdateTime >= updateDelayTime){
    switch(dolphinState){
    case STANDBY:
      getStandbyActParams();
      break;
    case SEARCH:
      getSearchActParams();
      break;
    case APPROACH:
      getApproachActParams();
      break;
    case VICTORY:
      getVictoryActParams();
      break;
    case HELPME: // If our state is not one of the above, we have a problem
    default:
      getHelpmeActParams();
      break;
    }
    lastUpdateTime = millis(); // Update timer
    return true;}
    return false; // Not time yet to check for param updates
}
```

Transmitting Act Parameters

The dolphin state, yaw servo position, and tail position are transmitted to the Act Arduino using the Wire library for I2C communication.

```
/**
 * ActCommunicationLogic.h
 * Purpose: Transmits state and yaw and pitch servo positions via I2C to the Act Arduino.
 * Notes: If sendActParams() is called, the receiving Arduino must be powered and able to
    receive messages. Otherwise, the program freezes here.
 */
#include <Wire.h>
#include "ActParams.h"
#define ACT_ADDRESS 8 // Address at which ACT Arduino is expecting Serial communication

void setupI2C(){
  Wire.begin(); // The address is optional for the master
}

// Transmits STATE, YAW POSITION, TAIL POSITION
void sendActParams(){
  if(getActParams()){
    Wire.beginTransmission(ACT_ADDRESS);
    // Transmit state
    Wire.write(dolphinState);
    Wire.write(",");
    // Transmit yaw position
    Wire.write(yawServoPos);
    Wire.write(",");
    Wire.write(tailServoPos);
    Wire.write(";");
    Wire.endTransmission();
  }
}
```

Sensor Checks

For this iteration of the robot, only the programmatic emergency stop has been implemented. If the Arduino writes a high signal to the relay pin, the relay will close, allowing current to pass through. This connection is furthermore regulated by the magnetic reed switch which must also be closed for servo operation. A low signal will open the relay and no current will pass through.

```
/**
 * SensorLogic.h
 * Purpose: Defines all system sensors and functions to check for normal performance.
 * Notes: All sensors (water, temperature, battery) were disabled for demo.
 */
#define estop_pin 8

void setupPins(){
    pinMode(estop_pin, OUTPUT);
}

// Writing 0 to the relay will open it, triggering the e-stop.
void freeze_motors(){
  digitalWrite(estop_pin, LOW);
}

// Release the E-stop in code. Allow servos to get power by closing the relay.
void release_motors(){
  digitalWrite(estop_pin, HIGH);
}

bool areSystemsOK(){
  return true; // No sensors are read, system assumed to be OK.
}
```

Act

The Act Arduino receives the dolphin state, yaw servo position, and the tail position from the sense/think Arduino using the Wire library again.

```
/**
  * ActDolphin.ino
  * Purpose: Code for act Arduino of team Clicks and Whistles robotic dolphin. Receives
    robot state and servo positions from sense/think Arduino over I2C. Writes positions to
     servos and blinks lights.
  * Warnings: Make sure battery powering the servos is FULLY charged; otherwise, the code
    may be running and receiving correctly, but there will be no movement from the servos.
  * Author: Katya Soltan
**/

#include <Wire.h>
#include <Servo.h>
#define ACT_ADDRESS 8
#define yellowLedPin 8
#define blueLedPin 9

int tailServoPos = 0;
int yawServoPos = 0;

#define yawServoPin 5
#define tailServoPin 6
Servo yawServo;
Servo tailServo;

enum robotState {
  STANDBY, // No movement
  SEARCH, // Searching movement, rotating about itself
  APPROACH, // Moving towards target
  VICTORY, // Victory dance on finding target (could also use to
           // go into center of pool to search for next target)
  HELPME // E-Stop
};

enum robotState dolphinState;

void setup() {
  setupServos();
  Serial.begin(115200);
  Wire.begin(ACT_ADDRESS);
  Wire.onReceive(receiveEvent); // Receive state, yaw position, and tail position
                                // position from Sense/Think Arduino
}

void setupServos(){
  yawServo.attach(yawServoPin);
  tailServo.attach(tailServoPin);
}

void receiveEvent(int bytes) {  // Receive state, yaw, pitch from SENSE/THINK Arduino
  if (Wire.available() == 6){ // Comma-separated state, yaw position, tail position,
                              // Total of 6 bytes (there's a trailing comma)
    dolphinState = Wire.read(); // Read the state
    Wire.read(); //Drop a comma
    yawServoPos = Wire.read();
    Wire.read();
    tailServoPos = Wire.read(); // Read the tail position as a byte

    // Check that the last character is semicolon
    if(Wire.read() != ';'){
        Serial.println("INCORRECTLY RECEIVED");
    }
  }
}
```

The yaw and tail servo positions are constantly being written to the servos. Because they should be updated when they change, this approach can be improved upon by writing to the servos only when the onReceive() function is called. To reflect the state of the robot, the LEDs need to be updated to either ON or OFF depending on the state.

```
void loop() {
  yawServo.write(yawServoPos);
  tailServo.write(tailServoPos);
  switch(dolphinState){
    case STANDBY:
      blinkStandbySignal();
      break;

    case SEARCH:
      blinkSearchSignal();
      break;

    case APPROACH:
      blinkApproachSignal();
      break;

    case VICTORY:
      blinkVictorySignal();
      break;

    case HELPME:
    default:
      blinkHelpmeSignal();
      break;
  }
}
```

The LEDs show the state of the robot. When both the yellow and blue lights are steadily on, *Flipper* is in Standby mode. When only the yellow light is flashing, the robot is searching for the buoy. Once it finds the target, the yellow becomes steady and the blue light begins to flash. Once it finds the target, both LEDs start to blink slowly. If the robot goes into Helpme mode, both LEDs flash very quickly.

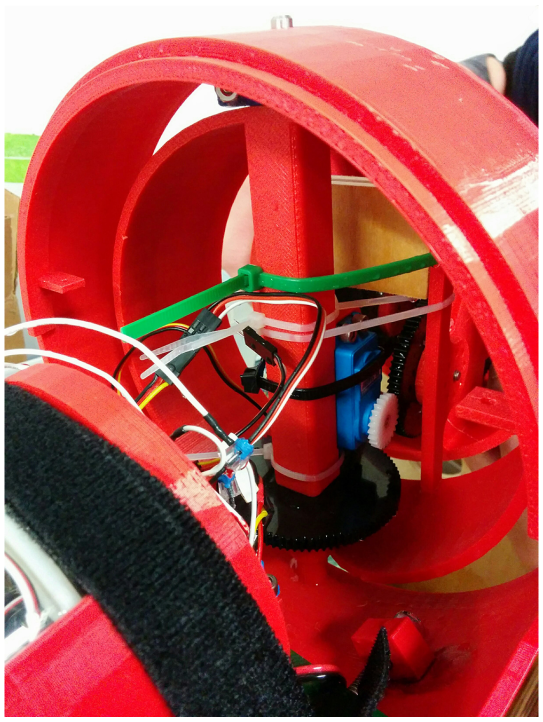
```
// Standby mode: Yellow and Blue LEDs are ON.
void blinkStandbySignal(){
  digitalWrite(yellowLedPin, HIGH);
  digitalWrite(blueLedPin, HIGH);
}

// Search mode: Yellow LED blinks every 500 ms. Blue LED is OFF.
void blinkSearchSignal(){
  digitalWrite(yellowLedPin, millis() % 500 > 250 ? HIGH : LOW); //Blink every 500ms
  digitalWrite(blueLedPin, LOW);
}

// Approach mode: Blue LEDs blink every 500ms. Yellow LED is ON.
void blinkApproachSignal(){
  digitalWrite(yellowLedPin, HIGH);
  digitalWrite(blueLedPin, millis() % 500 < 250 ? HIGH : LOW); //Blink every 500ms
}

// Victory mode: Yellow and Blue LEDs blink assymetrically every 650ms (slow)
void blinkVictorySignal(){
  digitalWrite(yellowLedPin, millis() % 650 > 500 ? HIGH : LOW); //Blink 650ms
       asymmetrically
  digitalWrite(blueLedPin, millis() % 650 > 500 ? HIGH : LOW); //Blink 650ms asymmetrically
}

// Helpme mode: Yellow and Blue LEDs blink every 150ms (fast).
void blinkHelpmeSignal(){
  digitalWrite(yellowLedPin, millis() % 150 > 75 ? HIGH : LOW); //Blink every 150ms
  digitalWrite(blueLedPin, millis() % 150 > 75 ? HIGH : LOW); //Blink every 150ms
}
```

Performance

The integration of all systems was not entirely successful. The pressure hull with electronics fit and connected to the batteries and servos inside of the front hull. The emergency stop worked flawlessly, with the magnet held by the dorsal fin exerting a strong enough magnetic field that, when the magnet was lowered, interacted with the reed switch and activated it. Separately, software subsystems performed well with the electronics, but when put together onto the robot, failed to move the yaw servo. For demo, *Flipper* only exhibited a straight swimming motion propelled by the undulating action of the tail.

Mechanical

A few flaws in the initial CAD design of the robot lead to some post-fabrication fixes. The most significant oversight was including places for wires. The battery holders did not account for the thick leads as well as the thick casing from their waterproofing; instead of facing the tail and having short leads connecting to the pressure hull screws, the batteries were rotated towards the front of the robot and required very long battery wires. The servo holders were not only too small, but also did not have a way to route the wires out and needed to be dremeled out. Additionally, the servos were not perfectly aligned with the gears and needed to be ziptied into place to prevent the slipping of the gears.



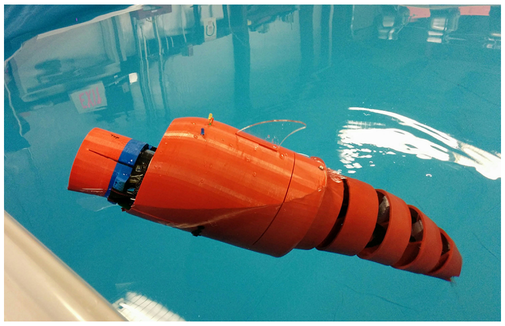
The servos were not aligned due to sizing issues and required zipties to be held in place.

The design of the tail gear train, while elegant, is very hard to tune; the gears clicked and were not fixed well to the links and required tightening, which in the space that was available was hard to impossible.

In the design of the front hull, the dolphin has a nose cone that completes its aesthetic. The cone was not sized properly and in the end needed to be discarded; however, without it, it was significantly easier to take out the pressure hull, which is already a lengthy process.

While the straps and velcro method of securing the pressure hull seems like it would give the most flexibility in debugging and replacing parts, the straps were hard to tighten and needed to be undone every single time the tube required opening. The velcro was very tightly routed through the retaining board, making the tightening process delicate, so as not to crack the board.

Fliipper did not float parallel to the surface of the water and instead has the nose pointed up to the ceiling. Tare rods for extra weight were available for tuning the nose of the robot to counteract the buoyancy of the pressure hull. In the next iteration of the robot, this must be fixed to allow the PixyCam to have a level, underwater image.



The nose of the dolphin points upward, which is not a useful position for the PixyCam.

Electrical

The electrical system, while functional, was very messy, with no good way to organize the wires. One reason for this was the very small size of the pressure hull. In order to create a robot as close to two feet long as possible, we needed to cut the tube very short to account for the length of *Flipper*'s tail.

The greatest setback for the electrical and software systems was the waterproofing of the tube. While the brass screws and Schrader valve were decently water resistant, the plastic valves sheered off, leaving the threaded sections inside of the end cap, and tearing off the silicone glue. It took three more glueing sessions, using both aquarium and gasket sealant to achieve a good seal. O-ring grease also improved the performance of the end cap. A paper towel was put into the tube to soak up any moisture that did end up getting through.

Waterproofing the pressure hull was the biggest setback for testing electronics and software integration.

Software

The XBee communication with the computer worked perfectly with the Sparkfun XBee shield. In the beginning, we had problems connecting to the XBee via the I/O shield, but this problem was corrected.

All sensors in the pressure hull were stripped for demo, including the temperature and water level sensors. Technically, *Flipper* cannot go into Helpme mode with the current code. This decision was made due to time constraints and the inability to calibrate the sensors well enough to distinguish between when the dolphin was actually in an emergency state and when the sensors gave false readings.

We were only able to test forward motion with *Flipper*.

The most obvious improvement in the code is to transfer the act parameter definitions to the act Arduino itself. We suspect that the reason the yaw servo was not getting signals, even though I2C was transmitting and receiving the correct values, is because the double timing loops were skipping update steps while the "updated" values were coming in too fast for the servo to respond.

Another obstacle in testing the code was the PixyCam, which sometimes recalibrated itself and forgot all the colors that it had been trained on. Additionally, the sizes of the blocks of color that it detected varied drastically with the angle at which it was pointed towards the same buoy (at the same distance) and where in the pool it was located. Due to the LEDs' light being dispersed in the water, when *Flipper* was extremely far away from the LED, the area of the buoy was three times that of when it was very close to it on the approach. We tried to implement a time delay between when the Arduino should consider if the buoy is close and the time at which the robot began to approach the target, but were not able to test this due to the fact that only forward propulsion was working.