Appendix C

Team #4: RoboSub

Fall 2014 – Spring 2015

Dennis Boyd

Bjorn Campbell

Samantha Cherbonneau

Kevin Matungwa

Elliot Mudrick

Introduction

As this is a multi-year project, the total code has become immense. So only the noteworthy code that was written or edited this year is included in this appendix. A full software listing has been uploaded to GitHub. The code here is organized by file and then function (if applicable).

Table of Contents

1. RoboSubControl_v2.cpp

This is the primary code for the sub. In contains all the controllers alongside the task management system, decision making control system, and main routine.

1.1 Includes, Globals, Definitions, Declarations, and Function Prototypes

```
/*
 * RoboSub_Control_v2.cpp
 *
 *  Created on: Mar 8, 2014
 *      Author: cobbju, Dennis Boyd, Kevin Matungwa, Elliot Mudrick
 *
 *    Version 2: Latest update 4-8-15
 */

#include <iostream>
#include <string>
#include <SerialStream.h>
#include <sstream>
#include <pthread.h>
#include <errno.h>
#include <cmath>
#include <iomanip>
#include <unistd.h>
#include "Logger.h"
#include "MissionTask.h"
#include "Types.h"
#include "RazorAHRS.h"
#include "../RoboSubObjectTracking/Database.h"

using namespace LibSerial;

// Serial port declarations
#define COM_PORT "/dev/ttyACM0"              // name of port the Mega is connected to
#define COM_PORT3 "/dev/ttyACM1"        //  port defined for the UNO
#define COM_PORT2 "/dev/ttyUSB0"     // port IMU is connected to

#define PWM_MAX 165                  // max PWM for safe operation
// Note: currently, no thruster is bridged.
#define PWM_MAX_BRIDGED 255          // max PWM for safe operation (bridged)
#define MAINTAIN_DEPTH 65            // not final

// global variables
SerialStream myArduino;                      // link to Arduino Mega
SerialStream myArduinoUno;              // link to the Arduino Uno
RazorAHRS *myIMU;                    // link to IMU
float myDepth;                               // variable holds the current depth
Logger myLogger;                     // log file link
bool myIsStopped = false;            // flag for software STOP
// Thusters 0 through 5
short dataBuf[6] = {0, 0, 0, 0, 0, 0};
std::stack<MissionTask*> missionTasks;
MissionTask myGate (Types::MISSION_TASKS::GATE);
Database db;

// Mutex and CV declarations
pthread_mutex_t imuMutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t queryMutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t heaveMutex = PTHREAD_MUTEX_INITIALIZER; // for vertical thrusters
pthread_mutex_t surgeMutex = PTHREAD_MUTEX_INITIALIZER; // for side thrusters
pthread_mutex_t taskIdentifyWait = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t nextCommandWait = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t taskIdentified = PTHREAD_COND_INITIALIZER;
pthread_cond_t cmdComplete = PTHREAD_COND_INITIALIZER;
pthread_mutex_t depthMutex = PTHREAD_MUTEX_INITIALIZER;

// thread declarations
void* stabilizationMain(void *);
void* taskMgrMain(void*);
void* dmcsMain(void*);
void* distrThrustBkgrdMain(void*);
void* zController(void*);
void* zControllerDepth(void*); //added
void* xController(void*);
```

```
//void* yController(void*); // Not currently in use
void* yawController(void*);
void* yawControllerVision(void*);
void* pitchController(void*);
void* depthSensorMain(void*);

// other functions
int initSerialLink();
void on_error(const std::string &msg);       // RazorAHRS function
void on_data(const float data[]);    // RazorAHRS function
void distributeThrust(char);
void mySleep();
void queryDatabase(std::string, int&);
void queryDatabase(std::string, float&);
void zController();
//void yController();
short convertZCtrlOutput(float);
short convertZCtrlOutputDepth(float); //added
short convertXCtrlOutput(float);
//short convertYCtrlOutput(float);
short convertYawCtrlOutput(float, bool);
short convertPitchCtrlOutput(float, bool);

// mission specific functions
void stopThrusters();
void brakeThrusters();
void submergeToOperatingDepth();
void goThroughGate();
```

1.2 main

The main routine creates the most important threads.

```
int main()
{
      std::string thread = "RoboSub_Control_v2";
      std::string method = "main()";
      std::string msg = "";

      if(initSerialLink())
        return 0;

      // push Mission Tasks onto the stack in reverse order
      missionTasks.push(&myGate);

      // spawn main child threads
      pthread_t tskMgr;
      pthread_t dmcs;
      pthread_t stblz; //added 2-15-15
      pthread_t dpthSnsr; //added
      pthread_create(&stblz, NULL, &stabilizationMain, NULL); //added 2-15-15
      pthread_create(&tskMgr, NULL, &taskMgrMain, NULL);
      pthread_create(&dmcs, NULL, &dmcsMain, NULL);
      pthread_create(&dpthSnsr, NULL, &depthSensorMain, NULL); //added
      pthread_join(stblz, NULL); //added 2-15-15
      pthread_join(tskMgr, NULL);
      pthread_join(dmcs, NULL);

      msg = "Scheduled Mission Tasks are complete !";
      myLogger.LOG(1, thread, method, msg);
      sleep(1);

      return 0;
}
```

1.3 taskMgrMain

The task manager is tightly coupled with the DMCS. When the DMCS is ready for a new task, the task manager pops the stack. It also contains a loop to wait for the camera to identify the gate, but this always breaks on the first loop, as the camera isn't being used for gate traversal right now.

```
/*------------------------- Task Manager ----------------------------*/

void* taskMgrMain(void*)
{
      std::string thread = "Task_Manager";
      std::string method = "taskMgrMain()";
      std::string msg = "";
      int taskId = 1;

      while(true)
      {
        if(missionTasks.empty())
                break;

        /* --------------- Gate Task ----------------------*/

        else if(missionTasks.top()->getTaskName() == "GATE")
        {
            sleep(5);

                while(taskId != 1)
                {
                        pthread_mutex_lock(&queryMutex);
                        queryDatabase("SELECT task_detected FROM Tasks_List WHERE "
                                        "task_id = 1", taskId);
                        pthread_mutex_unlock(&queryMutex);
                        sleep(1);
                }

                msg = "Gate identified!\n";
                myLogger.LOG(0, thread, method, msg);
                //std::cout << msg;

                missionTasks.top()->gateCmds.pop();
                pthread_cond_signal(&taskIdentified);

                while(!missionTasks.top()->gateCmds.empty())
                {
                        /* Task Manager should sleep while DMCS is working on current
                         * command. Once the DMCS is finished with the command, the
                         * DMCS will signal the condition variable to allow the TM
                         * to update the command/mission task stack as needed.
                         */
                        pthread_mutex_lock(&nextCommandWait);
                        pthread_cond_wait(&cmdComplete, &nextCommandWait);
                        missionTasks.top()->gateCmds.pop();
                        pthread_mutex_unlock(&nextCommandWait);
                }

                // mission task is either completed or timed out
                taskId = 0;
                missionTasks.pop();

                        db.open();
                        db.updateQuery("UPDATE Tasks_List SET task_completed = 1 WHERE task_id = 1");
                        db.close();
        }

        else
                ;       // do nothing

      } // all Mission Tasks are done

      myIsStopped = true;

      return NULL;
}
```

1.4 dmcsMain

The DMCS is where the stuff the sub does (such as moving forward) is programmed.

```
/*--------------------- Decision Making Control System ---------------------*/

void* dmcsMain(void*)
{
      std::string thread = "DMCS";
      std::string method = "main()";
      std::string msg = "";

      pthread_t zCtrlr;
      //pthread_t yCtrlr;
      pthread_t xCtrlr;
      pthread_t yawCtrlr;
      pthread_t yawCtrlrV; // for vision
      pthread_t distrThrustBkgrd;

      float arg; // for depth
      float *argPtr = &arg;

      float argGyroX;
      float *argGyroXPtr = &argGyroX;

      float argGyroY;
      float *argGyroYPtr = &argGyroY;

      float argGyroZ;
      float *argGyroZPtr = &argGyroZ;

      while(!missionTasks.empty())
      {
          stopThrusters();
        if(missionTasks.top()->getTaskName() == "GATE")
        {
                // spawn background distribute thrust thread
                pthread_create(&distrThrustBkgrd, NULL, &distrThrustBkgrdMain, NULL);

                pthread_mutex_lock(&taskIdentifyWait);
                pthread_cond_wait(&taskIdentified, &taskIdentifyWait);
                pthread_mutex_unlock(&taskIdentifyWait);

                myGate.setState(Types::TASK_STATE::IN_PROGRESS);

                while(myGate.getState() != Types::TASK_STATE::COMPLETED
                                && myGate.getState() != Types::TASK_STATE::ABORTED)
                {
                        /* DMCS should act depending on the current command
                         * at the top of the command stack
                         */
                        switch(missionTasks.top()->gateCmds.top())
                        {
                                case Types::GATE_COMMANDS::ALIGN_WITH_GATE:
                                {
                                        /* Emergency bailout loop

                                         dataBuf[0] = 0;
                                         dataBuf[1] = -60;
                                         dataBuf[2] = -60;
                                         dataBuf[3] = 0;
                                         dataBuf[4] = -60;
                                         dataBuf[5] = -60;

                                        while (true) {} // */

                                        // this section is for testing the gyroscope
                                        /*
                                        for (;;)
                                        {

                                                pthread_mutex_lock(&imuMutex);
                                                argGyroX = myIMU->getGyroX();
                                                argGyroY = myIMU->getGyroY();
                                                argGyroZ = myIMU->getGyroZ();
                                                pthread_mutex_unlock(&imuMutex);
```

```
                                        std::cout << "GyroX = " << std::setw(10) << argGyroX << "; GyroY =
" << std::setw(10)
                                                << argGyroY << "; GyroZ = " << std::setw(10) <<
argGyroZ*0.02 << "\n";

                                        usleep(100000);
                                }
                                // */

                                // When testing without laptop
                                //sleep(60);

                                argGyroZ = myIMU->getGyroZ();
                                // create yaw control at current heading
                                pthread_create(&yawCtrlr, NULL, &yawController, (void*) argGyroZPtr);

                                arg = 5.0f;
                                pthread_create(&zCtrlr, NULL, &zControllerDepth, (void*) argPtr);

                                dataBuf[0] = 100; // best 60
                                dataBuf[3] = 105; // best 62

                                // Wait to go through gate
                                sleep(60);

                                stopThrusters();

                                pthread_cancel(zCtrlr);
                                pthread_cancel(yawCtrlr);
                                pthread_cancel(distrThrustBkgrd);

                                std::cout << "Spinning...\n";
                                while(true) {}
                                // not using stuff below here right now

                                std::cout << "aligned !\n";
                                pthread_cond_signal(&cmdComplete);
                                mySleep();
                                break;
                        }

                        case Types::GATE_COMMANDS::APPROACH_GATE:
                        {
                                msg = "Approachinxg the gate...";
                                myLogger.LOG(0, thread, method, msg);

                                // spawn Z controller thread to maintain Z position
                                //pthread_create(&zCtrlr, NULL, &zController, NULL);

                                // spawn X controller thread to approach target
                                pthread_create(&xCtrlr, NULL, &xController, NULL);
                                // xController();

                                pthread_join(xCtrlr, NULL);
                                pthread_cancel(zCtrlr);

                                msg = "Gate distance threshold reached !";
                                myLogger.LOG(0, thread, method, msg);
                                std::cout << msg << "\n";

                                pthread_cond_signal(&cmdComplete);
                                mySleep();
                                break;
                        }

                        case Types::GATE_COMMANDS::GO_THROUGH_GATE:
                        {

                                std::cout << "Going through gate...";
                                goThroughGate();
                                pthread_cancel(yawCtrlr);
                                pthread_cancel(distrThrustBkgrd);
                                std::cout << "Task completed !\n";

                                pthread_cond_signal(&cmdComplete);
```

```
                                myGate.setState(Types::TASK_STATE::COMPLETED);
                                mySleep();
                                break;
                        }

                        default:
                                break;

                } // end gate command case statements

            } // all gate commands are completed

        } // end DMCS gate section

    } // all Mission Tasks are done

    stopThrusters();

    return NULL;
}
```

1.5 stabilizationMain

The stabilization initializes the IMU and creates the pitch controller.

```
/*------------------------- Stabilization Stuff ----------------------------*/

void *stabilizationMain(void *)
{
      std::string thread = "Stabilization";
      std::string method = "stabilizationMain()";
      std::string msg = "";
      std::string yawString = "";

      // initialize and connect to the Sparkfun Razor IMU
      try
      {
        /* Create Razor AHRS object. Serial I/O will run in background thread
         * and report errors and data updates using the callbacks on_data()
         * and on_error()
         */
        myIMU = new RazorAHRS(COM_PORT2, on_data, on_error, RazorAHRS::ROBOSUB); // changed from ROBOSUB
      }

      catch(std::runtime_error &e)
      {
        msg = "Could not create tracker: " + std::string(e.what());
        myLogger.LOG(0, thread, method, msg);
        msg = "Exiting Stabilization thread...";
        myLogger.LOG(0, thread, method, msg);

        pthread_exit(NULL);
      }

      pthread_t pitchCtrl;
      pthread_create(&pitchCtrl, NULL, &pitchController, NULL);
      pthread_join(pitchCtrl, NULL);

      msg = "Exiting Stabilization thread";
      myLogger.LOG(0, thread, method, msg);
      return NULL;
}
```

1.6 depthSensorMain

This function simply polls the depth sensor.

```
/*-------------------------- dpthSnsrMain ------------------------------*/

// added back in
void* depthSensorMain(void*)
{
      float tmp; //changed to float

      while(true)
      {
        // Read from Arduino Uno
        myArduinoUno >> tmp;
        pthread_mutex_lock(&depthMutex);
        myDepth = (float)tmp;
        pthread_mutex_unlock(&depthMutex);
      }
      return NULL;
}
```

1.7 distrThrustBkgrdMain

This runs in the background as a thread and distributes the thrust by translating the dataBuf values to things the MEGA can use and sending them to it.

```
/*------------------------- distrThrustBkgrd ----------------------------*/

void* distrThrustBkgrdMain(void*)
{
      std::string thread = "dTB";
      std::string method = "dTBMain()";
      std::string msg = "";
      std::string data;

      std::stringstream leftFront, rightFront, leftSide, rightSide, leftBack, rightBack;

      while(true)
      {
        pthread_testcancel();
        mySleep();

        leftFront << (int)(dataBuf[0] * 0.01 * PWM_MAX);
        rightFront << (int)(dataBuf[1] * 0.01 * PWM_MAX);
        leftSide << (int)(dataBuf[2] * 0.01 * PWM_MAX);
        rightSide << (int)(dataBuf[3] * 0.01 * PWM_MAX);
        leftBack << (int)(dataBuf[4] * 0.01 * PWM_MAX);
        rightBack << (int)(dataBuf[5] * 0.01 * PWM_MAX);

        data = leftFront.str() + "," + rightFront.str() + "," +
        leftSide.str() + "," + rightSide.str() + "," +
        leftBack.str() + "," + rightBack.str();

        myArduino << data;

        msg = "** Outgoing Message ** : " + data;
        myLogger.LOG(0, thread, method, msg);

        leftFront.str("");
        rightFront.str("");
        leftSide.str("");
        rightSide.str("");
        leftBack.str("");
        rightBack.str("");
      }

      return NULL;
}
```

1.8 zController

This function is intended to be called a single time to submerge the sub to the z-center of the gate. It is not currently implemented as it needs more testing (and zControllerDepth just works much better).

```
/*-------------------- Controller Related Stuff ------------------------*/

void zController()
{
      float prevError = 0.0f;
      float derivative = 0.0f;
      float error = 0.0f;
      float setpoint = 300.0f;
      float delayTime = 500000.0f;
      float output = 0.0f;
      float kP = 1.851f;
      float kD = 101.6f;
      float measuredValue;
      short tmp;

      do
      {
        pthread_mutex_lock(&heaveMutex);
        pthread_mutex_lock(&queryMutex);
        queryDatabase("SELECT z_center FROM Tasks_List WHERE task_id = 1", measuredValue);
        pthread_mutex_unlock(&queryMutex);

        error = (setpoint - measuredValue);

        // figure the output
        derivative = (error - prevError) / delayTime;
        prevError = error;
        output = kP * error + kD * derivative;

        // map the output
        tmp = convertZCtrlOutput(output);
        std::cout << "z " << error << "  " << tmp << "\n";

        //*2 added because thrust was too weak
        dataBuf[1] = tmp * 2.0f;
        dataBuf[2] = tmp * 2.0f;
        dataBuf[4] = tmp * 2.0f;
        dataBuf[5] = tmp * 2.0f;

        pthread_mutex_unlock(&heaveMutex);
        usleep(500000);

      }while(error > 10.0f || error < -10.0f);

      dataBuf[1] = MAINTAIN_DEPTH;
      dataBuf[2] = MAINTAIN_DEPTH;
      dataBuf[4] = MAINTAIN_DEPTH;
      dataBuf[5] = MAINTAIN_DEPTH;
}
```

1.9 zControllerDepth

This is the maintain depth controller. It is run as a thread. Unlike most of the controllers, it takes a parameter: the desired depth. This is passed in by reference and so the desired depth can be changed dynamically without having to recreate the thread.

```
void* zControllerDepth(void* arg)
{
      float prevError = 0.0f;
      float derivative = 0.0f;
      float error = 0.0f;
      float delayTime = 0.5f;         // in seconds
      float output = 0.0f;
      float kP = 220.0f;  //changed from 1.851f then increased from 75; 220 was best
      float kD = 1000.0f;  //was 101.6f; 1000 was best
      float measuredValue;
      bool decreaseThrustToGoUpSlowly = false;

      float * setpoint = (float*) arg;

      short tmp;

      do
      {
        pthread_testcancel();

        // check current depth reading
        pthread_mutex_lock(&depthMutex);
        measuredValue = myDepth;
        pthread_mutex_unlock(&depthMutex);
        //std::cout << "Depth measured: " << myDepth << "\n";

        error = (*setpoint - measuredValue);  // in feet

        while(error > 0.1f || error < -0.1f)  // in feet
        {
                // added abs to to remove erronious calculations
                derivative = abs((error - prevError) / delayTime);
                prevError = error;
                output = kP * error + kD * derivative;

                // map the output
                tmp = abs(convertZCtrlOutput(output));       // Thrusters should never change direction because
sub floats

                // don't let the controller wait longer than its iteration time
                if(pthread_mutex_trylock(&heaveMutex) == 0)
                {
                        if (tmp > 85) //ensure thrust isn't too high
                                tmp = 85;
                        //std::cout << "z -> sP:" << *setpoint << " mV:" << measuredValue << " E:" << error << "
o:" << output << " tmp:" << tmp << "\n";
                        dataBuf[1] = -1.0f * tmp; //change polarity to accomodate new thruster arragnement
                        dataBuf[2] = -1.0f * tmp;
                        dataBuf[4] = -1.0f * tmp;
                        dataBuf[5] = -1.1f * tmp; //compensate for uneven sitting/weaker thruster

                        pthread_mutex_unlock(&heaveMutex);
                }

                usleep(50000); // was 500,000

                // check the current depth reading
                pthread_mutex_lock(&depthMutex);
                measuredValue = myDepth;
                pthread_mutex_unlock(&depthMutex);

                error = (*setpoint - measuredValue);
        }

      }while(true);

      return NULL;
}
```

1.10 yawControllerVision

This is intended to use the camera to guide the sub through the gate (though it could easily be reworked to guide it based on another object). However, it has never worked very well and thus is not in current use. Some modifications might render quite useful however.

```
void* yawControllerVision(void*)
{
	float derivative = 0.0f;
	float error = 0.0f;
	float delayTime = 1.0f;
	float output = 0.0f;
	float kP = 1.0f; //
	float kD = 0.5f; //
	float prevError;
	float measuredValue;
	float prevMV;
	float setpoint = 400.0f;
	short tmp;
	short initialThrust;
	short initialThrustTmp;
	// counter to see if sub is too close to see gate -> switch to gyro-based yawController
	// if the gate can be seen, the y_center should always be changing, at least a little
	// if it stays the same for a long time, the database must not be being updated because
	// the gate is off screen.
	int gateGone = 0;

	sleep(3);

	pthread_testcancel();
	pthread_mutex_lock(&queryMutex);
	queryDatabase("SELECT y_center FROM Tasks_List WHERE task_id = 1", measuredValue);
	pthread_mutex_unlock(&queryMutex);

	std::cout << "Yaw Vision mv: " << measuredValue << "\n";

	while(measuredValue > 250.0f && measuredValue < 550.0f && gateGone != 5)
	{
	  //pthread_testcancel(); // needed?

	  prevMV = measuredValue;

	  pthread_mutex_lock(&queryMutex);
	  queryDatabase("SELECT y_center FROM Tasks_List WHERE task_id = 1", measuredValue);
	  pthread_mutex_unlock(&queryMutex);

	  if (prevMV == measuredValue)
		++gateGone;

	  error = setpoint - measuredValue;

	  std::cout << "Error above if: " << error << "\n";

	  if(error < -70.0f)
	  {
		pthread_mutex_lock(&surgeMutex);

		/*
		 * If the thruster is off when the Yaw controller kicks in,
		 * we need to boost the initial value so the controller output
		 * has an effect
		 */
		if(dataBuf[3] < 30 && dataBuf[3] >= 0)
		{
			initialThrustTmp = 30;
			initialThrust = dataBuf[3];
		}
		else
			initialThrust = initialThrustTmp = dataBuf[3];

		prevError = error; // + 0.2f;

		do
		{
			derivative = (std::abs(error) - prevError) / delayTime;
```

```
                        prevError = std::abs(error);
                        output = kP * std::abs(error) + kD * derivative;
                        // A side needs more thrust to account for drift
                        tmp = 1.2f * convertYawCtrlOutput(output, false);
                        dataBuf[3] = initialThrustTmp + tmp;
                        std::cout << " mv: " << measuredValue << " A yaw error: " << error << " tmp: " << tmp <<
"\n";

                        usleep(500000); //was 500000

                        prevMV = measuredValue;

                        pthread_mutex_lock(&queryMutex);
                        queryDatabase("SELECT y_center FROM Tasks_List WHERE task_id = 1", measuredValue);
                        pthread_mutex_unlock(&queryMutex);

                        if (prevMV == measuredValue)
                                ++gateGone;

                        error = setpoint - measuredValue;

                }while(error < -50 && measuredValue > 250.0f && measuredValue < 550.0f);

                // return to initial thrust once yaw is corrected
                dataBuf[3] = initialThrust;
                pthread_mutex_unlock(&surgeMutex);
        }
        else if(error > 70.0f)
        {
                pthread_mutex_lock(&surgeMutex);
                if(dataBuf[3] > -30 && dataBuf[3] <= 0)
                {
                        initialThrustTmp = -30;
                        initialThrust = dataBuf[3];
                }
                else
                        initialThrust = initialThrustTmp = dataBuf[3];

                prevError = std::abs(error);

                do
                {
                        derivative = (std::abs(error) - prevError) / delayTime;
                        prevError = std::abs(error);
                        output = kP * std::abs(error) + kD * derivative;
                        tmp = convertYawCtrlOutput(output, true);
                        dataBuf[3] = initialThrustTmp + tmp;
                        std::cout << " mv: " << measuredValue << " B yaw error: " << error << " tmp: " << tmp <<
"\n";

                                        usleep(500000); //was 500000

                        prevMV = measuredValue;

                        pthread_mutex_lock(&queryMutex);
                        queryDatabase("SELECT y_center FROM Tasks_List WHERE task_id = 1", measuredValue);
                        pthread_mutex_unlock(&queryMutex);

                        if (prevMV == measuredValue)
                                ++gateGone;

                        error = setpoint - measuredValue;

                }while(error > 50.0f && measuredValue > 250.0f && measuredValue < 550.0f);

                // return to initial thrust once yaw is corrected
                dataBuf[3] = initialThrust;
                pthread_mutex_unlock(&surgeMutex);
        }
        //pthread_testcancel(); // from 2013-14 team, dunno what it does
        sleep(2);
      }

      return NULL;

}
```

1.11 yawController

This is the main controller for maintaining the sub's heading. It uses the angular velocity from the IMU to correct for drift.

```
void* yawController(void* arg)
{
      float derivative = 0.0f;
      float error = 0.0f;
      float delayTime = 1.0f;
      float output = 0.0f;
      float kP = 120.0f; //was 5.0, best: 120
      float kD = 50.0f; //was 20.0, best: 20
      float prevError;
      float measuredValue;
      float sumOfErrors = 0;
      float * setpoint;
      short tmp;
      short initialThrust;
      short initialThrustTmp;

      sleep(3);
      std::cout << "Ready!\n";

      while(true)
      {
        pthread_mutex_lock(&imuMutex);
        measuredValue = (myIMU->getGyroZ()) * 0.02;  //multiply by 0.02 to convert from deg/s to deg
        pthread_mutex_unlock(&imuMutex);

        error = measuredValue;

        if(error < -0.5f) //was 1
        {
                pthread_mutex_lock(&surgeMutex);

                /*
                 * If the thruster is off when the Yaw controller kicks in,
                 * we need to boost the initial value so the controller output
                 * has an effect
                 */
                if(dataBuf[3] < 30 && dataBuf[3] >= 0)
                {
                        initialThrustTmp = 30;
                        initialThrust = dataBuf[3];
                }
                else
                        initialThrust = initialThrustTmp = dataBuf[3];

                prevError = error; // + 0.2f;

                do
                {
                        derivative = (measuredValue - prevError) / delayTime;
                        prevError = measuredValue;
                        output = kP * measuredValue + kD * derivative;
                        // A side needs more thrust to account for drift
                        tmp = 2.0f * convertYawCtrlOutput(output, false);
                        dataBuf[3] = initialThrustTmp + tmp;
                        std::cout << " mv: " << measuredValue << " A yaw error: " << error << " tmp: " << tmp <<
"\n";

                        usleep(500000); //was 500000

                        sumOfErrors += measuredValue;

                        pthread_mutex_lock(&imuMutex);
                        measuredValue = (myIMU->getGyroZ()) * 0.02;
                        pthread_mutex_unlock(&imuMutex);

                }while(-1 * sumOfErrors > error);

                // return to initial thrust once yaw is corrected
                dataBuf[3] = initialThrust;
                pthread_mutex_unlock(&surgeMutex);
        }
```

```
        else if(error > 0.5f) //was 1;
        {
                pthread_mutex_lock(&surgeMutex);
                if(dataBuf[3] > -30 && dataBuf[3] <= 0)
                {
                        initialThrustTmp = -30;
                        initialThrust = dataBuf[3];
                }
                else
                        initialThrust = initialThrustTmp = dataBuf[3];

                prevError = std::abs(error);

                do
                {
                        derivative = (std::abs(measuredValue) - prevError) / delayTime;
                        prevError = std::abs(measuredValue);
                        output = kP * std::abs(measuredValue) + kD * derivative;
                        tmp = convertYawCtrlOutput(output, true);
                        dataBuf[3] = initialThrustTmp + tmp;
                        std::cout << " mv: " << measuredValue << " B yaw error: " << error << " tmp: " << tmp <<
"\n";

                                        usleep(500000); //was 500000

                        sumOfErrors += measuredValue;

                        pthread_mutex_lock(&imuMutex);
                        measuredValue = (myIMU->getGyroZ()) * 0.02;
                        pthread_mutex_unlock(&imuMutex);

                }while(-1 * sumOfErrors < error);

                // return to initial thrust once yaw is corrected
                dataBuf[3] = initialThrust;
                pthread_mutex_unlock(&surgeMutex);
        }
        sumOfErrors = 0;
        pthread_testcancel();
        sleep(2);
      }

      return NULL;
}
```

1.12 pitchController

This is run as a thread to maintain the sub's pitch. It uses angular velocity from the IMU.

```
void* pitchController(void* arg)
{
      float derivative = 0.0f;
      float error = 0.0f;
      float delayTime = 1.0f;
      float output = 0.0f;
      float kP = 100.0f;
      float kD = 47.0f;
      float prevError;
      float measuredValue;
      short tmp;
      short initialThrust;
      float sumOfErrors = 0;

      sleep(3);

      while(!myIsStopped)
      {

        pthread_mutex_lock(&imuMutex);
        measuredValue = (myIMU->getGyroY()) * 0.02;
        pthread_mutex_unlock(&imuMutex);
        //std::cout << "gyroY " << measuredValue << "\n";

        //std::cout << "Measured value: " << measuredValue << "\n";
        error = measuredValue;

        if(error < -0.4f) // was 5
        {
                pthread_mutex_lock(&heaveMutex);
                initialThrust = dataBuf[4];
                prevError = error;

                do
                {
                        derivative = (measuredValue - prevError) / delayTime;
                        prevError = measuredValue;
                        output = kP * measuredValue + kD * derivative;
                        tmp = convertPitchCtrlOutput(output, true);
                        std::cout << "A pitch entrance error: " << error << " Correction: " << measuredValue << "
out: " << output << " tmp: " << tmp << "\n";
                        dataBuf[4] = initialThrust + tmp;
                        dataBuf[5] = initialThrust + tmp;

                        usleep(500000);

                        if (measuredValue > 0)
                                sumOfErrors += measuredValue;

                        //std::cout << "sum err: " << sumOfErrors << "\n";
                        pthread_mutex_lock(&imuMutex);
                        measuredValue = (myIMU->getGyroY()) * 0.02;
                        pthread_mutex_unlock(&imuMutex);

                }while(-1 * sumOfErrors > error); // changed from 2

                // return to initial thrust once pitch is corrected
                dataBuf[4] = initialThrust;
                dataBuf[5] = initialThrust;
                pthread_mutex_unlock(&heaveMutex);
        }

        else if(error > 0.4f) // was 5
        {
                pthread_mutex_lock(&heaveMutex);
                initialThrust = dataBuf[4];
                prevError = std::abs(error);

                do
                {
                        derivative = (std::abs(measuredValue) - prevError) / delayTime;
                        prevError = std::abs(measuredValue);
```

```
                        output = kP * std::abs(measuredValue) + kD * derivative;
                        tmp = convertPitchCtrlOutput(output, false);
                        std::cout << "B pitch  entrance error: " << error << " Correction: " << measuredValue <<
" out: " << output << " tmp: " << tmp << "\n";

                        dataBuf[4] = initialThrust + tmp;
                        dataBuf[5] = initialThrust + tmp;

                        usleep(500000);

                        if (measuredValue < 0)
                                sumOfErrors += measuredValue;
                        //std::cout << "sum err: " << sumOfErrors << "\n";

                        pthread_mutex_lock(&imuMutex);
                        measuredValue = (myIMU->getGyroY()) * 0.02;
                        pthread_mutex_unlock(&imuMutex);

                }while(-1 * sumOfErrors < error); // changed from 2

                // return to initial thrust once pitch is corrected
                dataBuf[4] = initialThrust;
                dataBuf[5] = initialThrust;
                pthread_mutex_unlock(&heaveMutex);
        }
        sumOfErrors = 0;
        pthread_testcancel();
        sleep(1);
      }

      return NULL;
}
```

1.13 on_data

Razor IMU function. It was modified to include all nine values from the IMU.

```
/*------------------------- Other Functions ----------------------------*/

void on_data(const float data[])
{
      std::cout << "  " << std::fixed << std::setprecision(1)
          << "ACC = " << std::setw(6) << data[0] << ", " << std::setw(6) << data[1] << ", " << std::setw(6) <<
data[2]
          << "        MAG = " << std::setw(7) << data[3] << ", " << std::setw(7) << data[4] << ", " <<
std::setw(7) << data[5]
          << "        GYR = " << std::setw(7) << data[6] << ", " << std::setw(7) << data[7] << ", " <<
std::setw(7) << data[8] << std::endl;
}
```

2. multipleObjectTracking.cpp

This is the vision code for the sub.

2.1 Includes and Globals

```
// make file in folder.... type make or make clean
//
//
//To Run type ./ColorDetection
//      To Calibrate type ./ColorDetection -calibrate
//

/*NOTES
  I notice the second camera comes on after making th changes then
  when an obkect of color orange is placed infont of the camera,
  the drawing tends to appear on the window designated to the bottom
  camera this may have something to do with the drawing function
  not being updated ... Kevin
  Latest Modification: 3-30-15
*/

#include <sstream>
#include <string>
#include <iostream>
#include <vector>

#include "opencv/highgui.h"
#include "opencv/cv.h"
#include "RoboSubColors.h"
#include "Database.h"

using namespace cv;

Database db;

//initial min and max HSV filter values.
//these will be changed using trackbars
int H_MIN = 0;
int H_MAX = 256;
int S_MIN = 0;
int S_MAX = 256;
int V_MIN = 0;
int V_MAX = 256;
//default capture width and height
const int FRAME_WIDTH = 800;
const int FRAME_HEIGHT = 600;
//max number of objects to be detected in frame
const int MAX_NUM_OBJECTS=50;
RNG rng(12345);
//minimum and maximum object area
const int MIN_OBJECT_AREA = 200;
const int MAX_OBJECT_AREA = FRAME_HEIGHT*FRAME_WIDTH;
//names that will appear at the top of each window
const string windowName = "Targeting Image";
const string windowNameDwn = "Targeting Image Down";
const string windowName1 = "HSV Image";
const string windowName2 = "Thresholded Image";
const string windowName3 = "After Morphological Operations";
const string trackbarWindowName1 = "TrackbarsFront"; // changed from Trackbars
const string trackbarWindowName2 = "TrackbarsBottom"; // added for unique window
```

2.2 Unchanged Functions

```
void on_trackbar( int, void* )
{//This function gets called whenever a
      // trackbar position is changed

}
```

```
string intToString(int number){


      std::stringstream ss;
      ss << number;
      return ss.str();
}

void createTrackbars(string name){
      //create window for trackbars

      namedWindow(name,0); // replaced trackBarWindowName with parameter
      //create memory to store trackbar name on window
      char TrackbarName[50];
      sprintf( TrackbarName, "H_MIN", H_MIN);
      sprintf( TrackbarName, "H_MAX", H_MAX);
      sprintf( TrackbarName, "S_MIN", S_MIN);
      sprintf( TrackbarName, "S_MAX", S_MAX);
      sprintf( TrackbarName, "V_MIN", V_MIN);
      sprintf( TrackbarName, "V_MAX", V_MAX);
      //create trackbars and insert them into window
      //3 parameters are: the address of the variable that is changing when the trackbar is moved(eg.H_LOW),
      //the max value the trackbar can move (eg. H_HIGH),
      //and the function that is called whenever the trackbar is moved(eg. on_trackbar)
      //                                  ---->    ---->     ---->
      createTrackbar( "H_MIN", name, &H_MIN, H_MAX, on_trackbar );
      createTrackbar( "H_MAX", name, &H_MAX, H_MAX, on_trackbar );
      createTrackbar( "S_MIN", name, &S_MIN, S_MAX, on_trackbar );
      createTrackbar( "S_MAX", name, &S_MAX, S_MAX, on_trackbar );
      createTrackbar( "V_MIN", name, &V_MIN, V_MAX, on_trackbar );
      createTrackbar( "V_MAX", name, &V_MAX, V_MAX, on_trackbar );

}

void drawObject(vector<RoboSubColors> theColors,Mat &frame){ //,Mat &frame2){

      for(int i =0; i<theColors.size(); i++){

      cv::circle(frame,cv::Point(theColors.at(i).getXPos(),theColors.at(i).getYPos()),10,cv::Scalar(0,0,255));
      cv::putText(frame,intToString(theColors.at(i).getXPos())+ " , " +
intToString(theColors.at(i).getYPos()),cv::Point(theColors.at(i).getXPos(),theColors.at(i).getYPos()+20),1,1,Sca
lar(0,255,0));
      cv::putText(frame,theColors.at(i).getType(),cv::Point(theColors.at(i).getXPos(),theColors.at(i).getYPos()-
30),1,2,theColors.at(i).getColor());

      }

}

void morphOps(Mat &thresh){

      //create structuring element that will be used to "dilate" and "erode" image.
      //the element chosen here is a 3px by 3px rectangle

      Mat erodeElement = getStructuringElement( MORPH_RECT,Size(3,3));
      //dilate with larger element so make sure object is nicely visible
      Mat dilateElement = getStructuringElement( MORPH_RECT,Size(8,8));

      erode(thresh,thresh,erodeElement);
      erode(thresh,thresh,erodeElement);


      dilate(thresh,thresh,dilateElement);
      dilate(thresh,thresh,dilateElement);

}
```

```
void trackFilteredObject(Mat threshold,Mat HSV, Mat &feedClone){

      vector <RoboSubColors> colors;

      Mat temp;
      threshold.copyTo(temp);
      //these two vectors needed for output of findContours
      vector< vector<Point> > contours;
      vector<Vec4i> hierarchy;
      //find contours of filtered image using openCV findContours function
      findContours(temp,contours,hierarchy,CV_RETR_CCOMP,CV_CHAIN_APPROX_SIMPLE );
      //use moments method to find our filtered object
      double refArea = 0;
      bool objectFound = false;
      if (hierarchy.size() > 0) {
        int numObjects = hierarchy.size();
        //if number of objects greater than MAX_NUM_OBJECTS we have a noisy filter
        if(numObjects<MAX_NUM_OBJECTS){
                for (int index = 0; index >= 0; index = hierarchy[index][0]) {

                        Moments moment = moments((cv::Mat)contours[index]);
                        double area = moment.m00;

                        //if the area is less than MIN_OBJECT_AREA then it is probably just noise
                        //if the area is the same as the image size, probably just a bad filter
                        //we only want the object with the largest area so we safe a reference area each
                        //iteration and compare it to the area in the next iteration.
                        if(area>MIN_OBJECT_AREA){

                                RoboSubColors color;

                                color.setXPos(moment.m10/area);
                                color.setYPos(moment.m01/area);


                                colors.push_back(color);

                                objectFound = true;

                                // Find the convex hull object for each contour
                         vector<vector<Point> >hull( contours.size() );
                         for( int i = 0; i < contours.size(); i++ )
                            {  convexHull( Mat(contours[i]), hull[i], false ); }

                         /// Draw contours + hull results
                         Mat drawing = Mat::zeros( threshold.size(), CV_8UC3 );
                         for( int i = 0; i< contours.size(); i++ )
                            {
                              drawContours( feedClone, contours, i, cv::Scalar(0,0,255), 1, 8, vector<Vec4i>(), 0,
Point() );
                              drawContours( feedClone, hull, i, cv::Scalar(0,0,255), 1, 8, vector<Vec4i>(), 0,
Point() );
                            }

                        }else objectFound = false;


                }
                //let user know you found an object
                if(objectFound ==true){
                        //draw object location on screen
                        drawObject(colors,feedClone);
                             }

        }else putText(feedClone,"TOO MUCH NOISE! ADJUST FILTER",Point(0,50),1,2,Scalar(0,0,255),2);
      }
}
```

2.3 getOrientation

This function was obtained from robospace.wordpress.com and is used by the bottom camera to find the angle of paths.

```
// function to get angle for line following orientation. (From robospace.wordpress.com)
double getOrientation(vector<Point> &pts, Mat &img)
{
    if (pts.size() == 0) return false;

    //Construct a buffer used by the pca analysis
    Mat data_pts = Mat(pts.size(), 2, CV_64FC1);
    for (int i = 0; i < data_pts.rows; ++i)
    {
        data_pts.at<double>(i, 0) = pts[i].x;
        data_pts.at<double>(i, 1) = pts[i].y;
    }


    //Perform PCA analysis
    PCA pca_analysis(data_pts, Mat(), CV_PCA_DATA_AS_ROW);

    //Store the position of the object
    Point pos = Point(pca_analysis.mean.at<double>(0, 0),
                      pca_analysis.mean.at<double>(0, 1));

    //Store the eigenvalues and eigenvectors
    vector<Point2d> eigen_vecs(2);
    vector<double> eigen_val(2);
    for (int i = 0; i < 2; ++i)
    {
        eigen_vecs[i] = Point2d(pca_analysis.eigenvectors.at<double>(i, 0),
                                pca_analysis.eigenvectors.at<double>(i, 1));

        eigen_val[i] = pca_analysis.eigenvalues.at<double>(0, i);
    }

    // Draw the principal components
    circle(img, pos, 3, CV_RGB(255, 0, 255), 2);
    line(img, pos, pos + 0.02 * Point(eigen_vecs[0].x * eigen_val[0], eigen_vecs[0].y * eigen_val[0]) ,
CV_RGB(255, 255, 0));
    line(img, pos, pos + 0.02 * Point(eigen_vecs[1].x * eigen_val[1], eigen_vecs[1].y * eigen_val[1]) ,
CV_RGB(0, 255, 255));

    return atan2(eigen_vecs[0].y, eigen_vecs[0].x);
}
```

2.4 trackFilteredObject

One of the primary vision functions. It calculates centers and distances for the front camera. It was modified take into consideration the task at hand when updating the database.

```
void trackFilteredObject(RoboSubColors theRoboSubColors,Mat threshold,Mat HSV, Mat &feedClone, string task){


      vector <RoboSubColors> colors;
      Point2f topTL, topBR;

      Mat temp;
      threshold.copyTo(temp);
      //these two vectors needed for output of findContours
      vector< vector<Point> > contours;
      vector<Vec4i> hierarchy;
      //find contours of filtered image using openCV findContours function
      findContours(temp,contours,hierarchy,CV_RETR_CCOMP,CV_CHAIN_APPROX_SIMPLE );
      //use moments method to find our filtered object
      double refArea = 0;
      bool objectFound = false;
      if (hierarchy.size() > 0) {
        int numObjects = hierarchy.size();
        //if number of objects greater than MAX_NUM_OBJECTS we have a noisy filter
        if(numObjects<MAX_NUM_OBJECTS){
                for (int index = 0; index >= 0; index = hierarchy[index][0]) {

                        Moments moment = moments((cv::Mat)contours[index]);
                        double area = moment.m00;

                        //if the area is less than MIN_OBJECT_AREA then it is probably just noise
                        //if the area is the same as the image size, probably just a bad filter
                        //we only want the object with the largest area so we safe a reference area each
                        //iteration and compare it to the area in the next iteration.
                        if(area>MIN_OBJECT_AREA){

                                RoboSubColors color;
                                vector <Rect> boundRect;

                                color.setXPos(moment.m10/area);
                                color.setYPos(moment.m01/area);
                                color.setType(theRoboSubColors.getType());
                                color.setColor(theRoboSubColors.getColor());

                                colors.push_back(color);

                                objectFound = true;
                                double perm;
                                // Find the convex hull object for each contour
                         vector<vector<Point> >hull( contours.size() );
                         for( int i = 0; i < contours.size(); i++ )
                            {
                              perm = arcLength(Mat(contours[i]), true);
                              if(perm > 350)
                              {
                                  convexHull( Mat(contours[i]), hull[i], false );
                                  boundRect.push_back(boundingRect(Mat(contours[i])));
                              }

                             }

                          /// Draw hull results
                          for( int i = 0; i< contours.size(); i++ )
                            {
                              drawContours(feedClone, hull, i, cv::Scalar(0,0,255), 1, 8, vector<Vec4i>(), 0,
Point() );
                            }

                         for( int i = 0; i< boundRect.size(); i++ )
                            {
                            if(i == 0)
                            {
                              topTL = boundRect[i].tl();
                              topBR = boundRect[i].br();
                            }
                            if(topTL.x > boundRect[i].tl().x)
```

```
                        {
                              topTL = boundRect[i].tl();
                        }
                        if(topTL.y > boundRect[i].tl().y)
                        {
                              topTL.y = boundRect[i].tl().y;
                        }
                        if(topBR.x < boundRect[i].br().x)
                        {
                              topBR.x = boundRect[i].br().x;
                        }
                        if(topBR.y < boundRect[i].br().y)
                        {
                              topBR.y = boundRect[i].br().y;
                        }

                        }
                        rectangle(feedClone, topTL, topBR, theRoboSubColors.getColor(), 1, 8, 0);

                        float distance = 0.00;
                        Point2f centerP;
                        centerP.x = (topBR.x + topTL.x)/2.0f;
                        centerP.y =(topBR.y + topTL.y)/2.0f;
                        const double focalLength = 1524.00, gateHeight = 5.00;
                        circle(feedClone, centerP, 22, cv::Scalar(0,0,255), 1, 8, 0);
                        line(feedClone, Point(centerP.x,centerP.y+20), Point(centerP.x,centerP.y-20),
cv::Scalar(0,0,255), 1, 8, 0);
                       line(feedClone, Point(centerP.x-20, centerP.y),Point(centerP.x+20,centerP.y),
cv::Scalar(0,0,255), 1, 8, 0);

                        distance = (focalLength/(topBR.y - topTL.y)) * gateHeight;


                          string dis = static_cast<ostringstream*>( &(ostringstream() << distance) )->str();
                          //changes for Julius' 3d axises
                          string ycenter = static_cast<ostringstream*>( &(ostringstream() << centerP.x) )-
>str();
                          string zcenter = static_cast<ostringstream*>( &(ostringstream() << centerP.y) )-
>str();

                          db.open();
                          //changed Where task_id from 1 to see if bottom camera will work
                      db.updateQuery("UPDATE Tasks_List SET task_detected = 1, distance = "+dis+", y_center =
"+ycenter+", z_center = "+zcenter+" WHERE task_id = "+task);

                          db.close();



                      }else objectFound = false;


               }
               //let user know you found an object
               if(objectFound ==true){
                      //draw object location on screen
                      drawObject(colors,feedClone);

               }

       }else putText(feedClone,"TOO MUCH NOISE! ADJUST FILTER",Point(0,50),1,2,Scalar(0,0,255),2);
      }
}
```

2.5 trackFilteredObjectDwn

Primary vision for bottom camera. Instead of calculating distance it calculates the angle of the object.

```
// second tracking function: for bottom camera
void trackFilteredObjectDwn(RoboSubColors theRoboSubColors,Mat threshold,Mat HSV, Mat &feedClone, string task){

      double angleY;

      vector <RoboSubColors> colors;
      Point2f topTL, topBR;

      Mat temp;
      threshold.copyTo(temp);
      //these two vectors needed for output of findContours
      vector< vector<Point> > contours;
      vector<Vec4i> hierarchy;
      //find contours of filtered image using openCV findContours function
      findContours(temp,contours,hierarchy,CV_RETR_CCOMP,CV_CHAIN_APPROX_SIMPLE );
      //use moments method to find our filtered object
      double refArea = 0;
      bool objectFound = false;
      if (hierarchy.size() > 0) {
        int numObjects = hierarchy.size();
        //if number of objects greater than MAX_NUM_OBJECTS we have a noisy filter
        if(numObjects<MAX_NUM_OBJECTS){
                for (int index = 0; index >= 0; index = hierarchy[index][0]) {

                        Moments moment = moments((cv::Mat)contours[index]);
                        double area = moment.m00;

                        //if the area is less than MIN_OBJECT_AREA then it is probably just noise
                        //if the area is the same as the image size, probably just a bad filter
                        //we only want the object with the largest area so we safe a reference area each
                        //iteration and compare it to the area in the next iteration.
                        if(area>MIN_OBJECT_AREA){

                                RoboSubColors color;
                                vector <Rect> boundRect;

                                color.setXPos(moment.m10/area);
                                color.setYPos(moment.m01/area);
                                color.setType(theRoboSubColors.getType());
                                color.setColor(theRoboSubColors.getColor());

                                colors.push_back(color);

                                objectFound = true;
                                double perm;
                                // Find the convex hull object for each contour
                         vector<vector<Point> >hull( contours.size() );
                         for( int i = 0; i < contours.size(); i++ )
                            {
                              perm = arcLength(Mat(contours[i]), true);
                              if(perm > 350)
                              {
                                  convexHull( Mat(contours[i]), hull[i], false );
                                  boundRect.push_back(boundingRect(Mat(contours[i])));
                              }

                             }

                          /// Draw hull results
                          for( int i = 0; i< contours.size(); i++ )
                            {
                              drawContours(feedClone, hull, i, cv::Scalar(0,0,255), 1, 8, vector<Vec4i>(), 0,
Point() );

                        angleY = getOrientation(hull[i], feedClone);
                        //std::stringstream angleT;
                        //angleT << angleY;
                        //std::string angleYS = angleT.str();

                        std::cout << "Angle = " << angleY << "\n";

                        string angleYS = static_cast<ostringstream*>( &(ostringstream() << angleY) )->str();
```

```
                    db.open();
                             //changed Where task_id from 1 to see if bottom camera will work
                    if (angleYS != "0")
                        db.updateQuery("UPDATE Tasks_List SET task_detected = 1, angle_y = "+angleYS+" WHERE
task_id = "+task);
                             db.close();
                            }


                        }else objectFound = false;


                }
                //let user know you found an object
                if(objectFound ==true){
                        //draw object location on screen
                        drawObject(colors,feedClone);
                        }

        }else putText(feedClone,"TOO MUCH NOISE! ADJUST FILTER",Point(0,50),1,2,Scalar(0,0,255),2);
      }
}
```

2.6 main

The main routine for the camera. Creates and runs both cameras.

```
int main(int argc, char* argv[])
{
        //if we would like to calibrate our filter values, set to true.
      bool calibrationMode = false;
      if(argc > 1){
        std::string cali = argv[1];
        if(cali == "-calibrate")
        {
            calibrationMode = true;
        }
        }


      //Matrix to store each frame of the webcam feed
      Mat cameraFeed;
        Mat camFeedDwn;
      Mat threshold;
      Mat threshDwn;
      Mat HSV;
      Mat HSVDwn;
      Mat feedClone;
      Mat feedClnDwn;


      if(calibrationMode){
        //create slider bars for HSV filtering
        createTrackbars(trackbarWindowName1);
        createTrackbars(trackbarWindowName2);

      }
      //video capture object to acquire webcam feed
      VideoCapture capture;
        VideoCapture camDwn;   // added for bottom camera

      //open capture object at location zero (default location for webcam)
      capture.open(0);
      camDwn.open(1);        //added for bottom camera


      //set height and width of capture frame
      capture.set(CV_CAP_PROP_FRAME_WIDTH,FRAME_WIDTH);
      capture.set(CV_CAP_PROP_FRAME_HEIGHT,FRAME_HEIGHT);

      //added for bottom camera
        camDwn.set(CV_CAP_PROP_FRAME_WIDTH,FRAME_WIDTH);
      camDwn.set(CV_CAP_PROP_FRAME_HEIGHT,FRAME_HEIGHT);

      //start an infinite loop where webcam feed is copied to cameraFeed matrix
      //all of our operations will be performed within this loop
      while(1){
        //store image to matrix
        capture.read(cameraFeed);
        camDwn.read(camFeedDwn); //added for bottom camera

        feedClone = cameraFeed.clone();
                 feedClnDwn = camFeedDwn.clone();

                vector<Mat> channels;
        vector<Mat> channelsDwn;
                vector<Mat> hls_planes;
        vector<Mat> hls_planes_dwn;
                Mat HLS;
        Mat HLSDwn;

      // split(cameraFeed,channels);
      // split(camFeedDwn,channelsDwn);
        //equalizeHist(channels[0], channels[0]); orignally commented
        // equalizeHist(channels[1], channels[1]);
        // equalizeHist(channelsDwn[2], channelsDwn[2]);
        //equalizeHist(channels[2], channels[2]); originally commented
        // merge(channels,cameraFeed);
        // merge(channelsDwn,camFeedDwn);
```

```
        //imshow("Contrast",cameraFeed);  // commented out by kevin both lines just gave an extra screen
        //imshow("Contrast",camFeedDwn);

        cvtColor(cameraFeed,HLS,COLOR_BGR2HLS);
        cvtColor(camFeedDwn,HLSDwn,COLOR_BGR2HLS); // changed was not including dwn
        hls_planes.clear();
        hls_planes_dwn.clear();
        split(HLS,hls_planes);
        split(HLSDwn,hls_planes_dwn);
      //equalizeHist(hls_planes[1],hls_planes[1]);originally commented
        //equalizeHist(hls_planes[2],hls_planes[2]);originally commented
        //merge(hls_planes,HLS);
        //merge(hls_planes_dwn,HLSDwn);
       // cvtColor(HLS,cameraFeed,COLOR_HLS2BGR);
       // cvtColor(HLSDwn,camFeedDwn,COLOR_HLS2BGR);
        // imshow("HLS",cameraFeed);   // this bring up the fron camera < HHHMMMMM >
        // imshow("HLSDwn",camFeedDwn); //changed by kevin HLS to HLSDwn then commented later after realizing
this just added an extra screen

        GaussianBlur(cameraFeed, cameraFeed, Size(3,3), 0, 0, BORDER_DEFAULT );
        GaussianBlur(camFeedDwn, camFeedDwn, Size(3,3), 0, 0, BORDER_DEFAULT );

        if(calibrationMode==true){
        //if in calibration mode, we track objects based on the HSV slider values.
        cvtColor(cameraFeed,HSV,COLOR_BGR2HSV);
        cvtColor(camFeedDwn,HSV,COLOR_BGR2HSV);
        inRange(HSV,Scalar(H_MIN,S_MIN,V_MIN),Scalar(H_MAX,S_MAX,V_MAX),threshold);
        inRange(HSVDwn,Scalar(H_MIN,S_MIN,V_MIN),Scalar(H_MAX,S_MAX,V_MAX),threshDwn);
        morphOps(threshold);
        morphOps(threshDwn);
        //imshow(windowName1,threshold); // changed window this to see if it was window one
        //imshow(windowName2,threshDwn);
        trackFilteredObject(threshold,HSV,feedClone);
        trackFilteredObject(threshDwn,HSVDwn,feedClnDwn);
        }else{
        //create some temp RoboSubColors objects so that
        //we can use their member functions/information
        RoboSubColors green("green"), yellow("yellow"), red("red"), blue("blue"), orange("orange"),
orangeUW("orangeUW");

                //orange (in lab) -> now orange underwater (in practice)
                cvtColor(cameraFeed,HSV,COLOR_BGR2HSV);
                inRange(HSV,orange.getHSVmin(),orange.getHSVmax(),threshold);
        morphOps(threshold);
        trackFilteredObject(orange,threshold,HSV,feedClone,"1");

        cvtColor(camFeedDwn,HSVDwn,COLOR_BGR2HSV);
                inRange(HSVDwn,orange.getHSVmin(),orange.getHSVmax(),threshDwn);
        morphOps(threshDwn);
        trackFilteredObjectDwn(orange,threshDwn,HSVDwn,feedClnDwn,"2");


        }


        imshow(windowName,feedClone);
        imshow(windowNameDwn,feedClnDwn);

        //delay 30ms so that screen can refresh.
        //image will not appear without this waitKey() command

        if(waitKey(30) == 27)
        {
            break;
        }
      }

      return 0;
}
```

3. RoboSubColors.cpp

This is where colors are defined. The definition for orange underwater (“orangeUW”) was added here.

```
#include "RoboSubColors.h"

RoboSubColors::RoboSubColors()
{
      //set values for default constructor
      setType("null");
      setColor(Scalar(0,0,0));

}

RoboSubColors::RoboSubColors(string name){

      setType(name);

      if(name=="green"){

        //TODO: use "calibration mode" to find HSV min
        //and HSV max values

        setHSVmin(Scalar(38,77,56));
        setHSVmax(Scalar(91,256,256));

        //BGR value for Green:
        setColor(Scalar(0,255,0));

      }
      if(name=="yellow"){

        //TODO: use "calibration mode" to find HSV min
        //and HSV max values

        setHSVmin(Scalar(22,129,91));
        setHSVmax(Scalar(38,256,219));

        //BGR value for Yellow:
        setColor(Scalar(0,255,255));

      }
      if(name=="red"){

        //TODO: use "calibration mode" to find HSV min
        //and HSV max values

        setHSVmin(Scalar(160,0,0));
        setHSVmax(Scalar(256,256,256));

        //BGR value for Red:
        setColor(Scalar(0,0,255));

      }
      if(name=="blue"){

        //TODO: use "calibration mode" to find HSV min
        //and HSV max values

        setHSVmin(Scalar(91,54,33));
        setHSVmax(Scalar(130,256,256));

        //BGR value for blue:
        setColor(Scalar(255,0,0));

      }
      if(name=="orange"){

        //TODO: use "calibration mode" to find HSV min
        //and HSV max values

        setHSVmin(Scalar(0,191,200));
        setHSVmax(Scalar(22,256,256));

        //BGR value for orange:
        setColor(Scalar(0,165,255));
```

```
      }
      if(name=="orangeUW"){

        // use google to find HSV values from RGB values on the camera
        setHSVmin(Scalar(20,20,80));
        setHSVmax(Scalar(180,160,200));

        //BGR value for orange:
        setColor(Scalar(0,165,255));
      }

}

RoboSubColors::~RoboSubColors(void)
{
}

int RoboSubColors::getXPos(){

      return RoboSubColors::xPos;

}

void RoboSubColors::setXPos(int x){

      RoboSubColors::xPos = x;

}

int RoboSubColors::getYPos(){

      return RoboSubColors::yPos;

}

void RoboSubColors::setYPos(int y){

      RoboSubColors::yPos = y;

}

Scalar RoboSubColors::getHSVmin(){

      return RoboSubColors::HSVmin;

}
Scalar RoboSubColors::getHSVmax(){

      return RoboSubColors::HSVmax;
}

void RoboSubColors::setHSVmin(Scalar min){

      RoboSubColors::HSVmin = min;
}


void RoboSubColors::setHSVmax(Scalar max){

      RoboSubColors::HSVmax = max;
}
```

4. RazorAHRS.cpp

This is the code for the IMU. It was modified to include functions which can be used in RoboSubControl_v2.cpp to get the IMU values for use.

```
/*
* Mac OSX / Unix / Linux C++ Interface for Razor AHRS v1.4.2
* 9 Degree of Measurement Attitude and Heading Reference System
* for Sparkfun "9DOF Razor IMU" and "9DOF Sensor Stick"
*
* Released under GNU GPL (General Public License) v3.0
* Copyright (C) 2013 Peter Bartz [http://ptrbrtz.net]
* Copyright (C) 2011-2012 Quality & Usability Lab, Deutsche Telekom Laboratories, TU Berlin
* Written by Peter Bartz (peter-bartz@gmx.de)
*
* Infos, updates, bug reports,contributions and feedback:
*     https://github.com/ptrbrtz/razor-9dof-ahrs
*
* Edited by Julius Cobb - revised the initial implementation for RoboSub use
* Edited by Dennis Boyd, Elliot Mudrick, and Kevin Matungwa to work - 3-4-15
*/

#include "RazorAHRS.h"
#include <cassert>
#include <iostream>

RazorAHRS::RazorAHRS(const std::string &port, DataCallbackFunc data_func, ErrorCallbackFunc error_func,
    Mode mode, int connect_timeout_ms, speed_t speed)
    : _mode(mode)
    , _input_pos(0)
    , _connect_timeout_ms(connect_timeout_ms)
    , data(data_func)
    , error(error_func)
    , _thread_id(0)
    , _stop_thread(false)
{
  for(int i = 0; i < 9; i++)
              this->myImuData[i] = 0;

  // check data type sizes
  assert(sizeof(char) == 1);
  assert(sizeof(float) == 4);

  // open serial port
  if (port == "")
    throw std::runtime_error("No port specified!");
  if (!_open_serial_port(port.c_str()))
    throw std::runtime_error("Could not open serial port!");

  // get port attributes
  struct termios tio;
  if (int errorID = tcgetattr(_serial_port, &tio))
    throw std::runtime_error("Could not get serial port attributes! Error # " + to_str(errorID));

  /* see http://www.easysw.com/~mike/serial/serial.html */
  /* and also http://linux.die.net/man/3/tcsetattr */
  // basic raw/non-canonical setup
  cfmakeraw(&tio);

  // enable reading and ignore control lines
  tio.c_cflag |= CREAD | CLOCAL;

  // set 8N1
  tio.c_cflag &= ~PARENB; // no parity bit
  tio.c_cflag &= ~CSTOPB; // only one stop bit
  tio.c_cflag &= ~CSIZE;  // clear data bit number
  tio.c_cflag |= CS8;     // set 8 data bits

  // no hardware flow control
  tio.c_cflag &= ~CRTSCTS;
  // no software flow control
  tio.c_iflag &= ~(IXON | IXOFF | IXANY);

  // poll() is broken on OSX, so we set VTIME and use read(), which is ok since
  // we're reading only one port anyway
```

```
  tio.c_cc[VMIN]  = 0;
  tio.c_cc[VTIME] = 10; // 10 * 100ms = 1s

  // set port speed
  if (int errorID = cfsetispeed(&tio, speed))
    throw std::runtime_error(" " + to_str(errorID)
        + ": Could not set new serial port input speed to "
        + to_str(speed) + ".");
  if (int errorID = cfsetospeed(&tio, speed))
    throw std::runtime_error(" " + to_str(errorID)
        + ": Could not set new serial port output speed to "
        + to_str(speed) + ".");

  // set port attributes
  // must be done after setting speed!
  if (int errorID = tcsetattr(_serial_port, TCSANOW, &tio))
  {
    throw std::runtime_error(" " + to_str(errorID)
        + ": Could not set new serial port attributes.");
  }

  // start input/output thread
  _start_io_thread();
}

RazorAHRS::RazorAHRS(const std::string &port,
    Mode mode, int connect_timeout_ms, speed_t speed)
    : _mode(mode)
    , _input_pos(0)
    , _connect_timeout_ms(connect_timeout_ms)
    , _thread_id(0)
    , _stop_thread(false)
{
  for(int i = 0; i < 9; i++)
              this->myImuData[i] = 0;

  // check data type sizes
  assert(sizeof(char) == 1);
  assert(sizeof(float) == 4);

  // open serial port
  if (port == "")
    throw std::runtime_error("No port specified!");
  if (!_open_serial_port(port.c_str()))
    throw std::runtime_error("Could not open serial port!");

  // get port attributes
  struct termios tio;
  if (int errorID = tcgetattr(_serial_port, &tio))
    throw std::runtime_error("Could not get serial port attributes! Error # " + to_str(errorID));

  /* see http://www.easysw.com/~mike/serial/serial.html */
  /* and also http://linux.die.net/man/3/tcsetattr */
  // basic raw/non-canonical setup
  cfmakeraw(&tio);

  // enable reading and ignore control lines
  tio.c_cflag |= CREAD | CLOCAL;

  // set 8N1
  tio.c_cflag &= ~PARENB; // no parity bit
  tio.c_cflag &= ~CSTOPB; // only one stop bit
  tio.c_cflag &= ~CSIZE;  // clear data bit number
  tio.c_cflag |= CS8;     // set 8 data bits

  // no hardware flow control
  tio.c_cflag &= ~CRTSCTS;
  // no software flow control
  tio.c_iflag &= ~(IXON | IXOFF | IXANY);

  // poll() is broken on OSX, so we set VTIME and use read(), which is ok since
  // we're reading only one port anyway
  tio.c_cc[VMIN]  = 0;
  tio.c_cc[VTIME] = 10; // 10 * 100ms = 1s

  // set port speed
  if (int errorID = cfsetispeed(&tio, speed))
```

```
    throw std::runtime_error(" " + to_str(errorID)
        + ": Could not set new serial port input speed to "
        + to_str(speed) + ".");
  if (int errorID = cfsetospeed(&tio, speed))
    throw std::runtime_error(" " + to_str(errorID)
        + ": Could not set new serial port output speed to "
        + to_str(speed) + ".");

  // set port attributes
  // must be done after setting speed!
  if (int errorID = tcsetattr(_serial_port, TCSANOW, &tio))
  {
    throw std::runtime_error(" " + to_str(errorID)
        + ": Could not set new serial port attributes.");
  }

  // start input/output thread
  _start_io_thread();
}

RazorAHRS::~RazorAHRS()
{
  // if thread was started, stop thread
  if (_thread_id) _stop_io_thread();
  close(_serial_port);
}

bool
RazorAHRS::_read_token(const std::string &token, char c)
{
  if (c == token[_input_pos++])
  {
    if (_input_pos == token.length())
    {
      // synch token found
      _input_pos = 0;
      return true;
    }
  }
  else
  {
    _input_pos = 0;
  }

  return false;
}

bool
RazorAHRS::_init_razor()
{
  char in;
  int result;
  struct timeval t0, t1, t2;
  const std::string synch_token = "#SYNCH";
  const std::string new_line = "\r\n";

  // start time
  gettimeofday(&t0, NULL);

  // request synch token to see if Razor is really present
  const std::string contact_synch_id = "00";
  const std::string contact_synch_request = "#s" + contact_synch_id;
  const std::string contact_synch_reply = synch_token + contact_synch_id + new_line;
  write(_serial_port, contact_synch_request.data(), contact_synch_request.length());
  gettimeofday(&t1, NULL);

  // set non-blocking I/O
  if (!_set_nonblocking_io()) return false;

  /* look for tracker */
  while (true)
  {
    // try to read one byte from the port
    result = read(_serial_port, &in, 1);

    // one byte read
    if (result > 0)
```

```
    {
      if (_read_token(contact_synch_reply, in))
        break;
    }
    // no data available
    else if (result == 0)
      usleep(1000); // sleep 1ms
    // error?
    else
    {
      if (errno != EAGAIN && errno != EINTR)
        throw std::runtime_error("Can not read from serial port (1).");
    }

    // check timeout
    gettimeofday(&t2, NULL);
    if (elapsed_ms(t1, t2) > 200)
    {
      // 200ms elapsed since last request and no answer -> request synch again
      // (this happens when DTR is connected and Razor resets on connect)
      write(_serial_port, contact_synch_request.data(), contact_synch_request.length());
      t1 = t2;
    }
    if (elapsed_ms(t0, t2) > _connect_timeout_ms)
      // timeout -> tracker not present
      throw std::runtime_error("Can not init: tracker does not answer.");
  }


  /* configure tracker */
  // set correct binary output mode, enable continuous streaming, disable errors and
  // request synch token. So we're good, no matter what state the tracker
  // currently is in.
  const std::string config_synch_id = "01";
  const std::string config_synch_reply = synch_token + config_synch_id + new_line;

  std::string config = "#o1#oe0#s" + config_synch_id;
  if (_mode == YAW_PITCH_ROLL) config = "#ob" + config;
  else if (_mode == ROBOSUB) config = "#or" + config;
  else if (_mode == ACC_MAG_GYR_RAW) config = "#osrb" + config;
  else if (_mode == ACC_MAG_GYR_CALIBRATED) config = "#oscb" + config;
  else throw std::runtime_error("Can not init: unknown 'mode' parameter.");

  write(_serial_port, config.data(), config.length());

  // set blocking I/O
  // (actually semi-blocking, because VTIME is set)
  if (!_set_blocking_io()) return false;

  while (true)
  {
    // try to read one byte from the port
    result = read(_serial_port, &in, 1);

    // one byte read
    if (result > 0)
    {
      if (_read_token(config_synch_reply, in))
        break;  // alrighty
    }
    // error?
    else
    {
      if (errno != EAGAIN && errno != EINTR)
        throw std::runtime_error("Can not read from serial port (2).");
    }
  }

  // we keep using blocking I/O
  //if (_set_blocking_io() == -1)
  //  return false;

  return true;
}

bool
RazorAHRS::_open_serial_port(const char *port)
```

```
{
  // O_NDELAY allows open even with no carrier detect (e.g. needed for Razor)
  if ((_serial_port = open(port, O_RDWR | O_NOCTTY | O_NDELAY)) != -1)
  {
    // make I/O blocking again
    if (_set_blocking_io()) return true;
  }

  // something didn't work
  close(_serial_port);
  return false;
}

bool
RazorAHRS::_set_blocking_io()
{
  int flags;

  // clear O_NDELAY to make I/O blocking again
  // in fact this is semi-blocking, since we set VTIME on the port
  if (((flags = fcntl(_serial_port, F_GETFL, 0)) != -1) &&
      (fcntl(_serial_port, F_SETFL, flags & ~O_NDELAY)) != -1)
  {
    return true;
  }

  return false;
}

bool
RazorAHRS::_set_nonblocking_io()
{
  int flags;

  // set O_NDELAY to make I/O non-blocking
  if (((flags = fcntl(_serial_port, F_GETFL, 0)) != -1) &&
      (fcntl(_serial_port, F_SETFL, flags | O_NDELAY)) != -1)
  {
    return true;
  }

  return false;
}

bool
RazorAHRS::_is_io_blocking()
{
  return (fcntl(_serial_port, F_GETFL, 0) & O_NDELAY);
}

void*
RazorAHRS::_thread(void *arg)
{
  char c;
  int result;

  try
  {
    if (!_init_razor())
    {
      error("Tracker init failed.");
      return arg;
    }
  }
  catch(std::runtime_error& e)
  {
    error("Tracker init failed: " + std::string(e.what()));
    return arg;
  }

  while (!_stop_thread)
  {
    if ((result = read(_serial_port, &c, 1)) > 0) // blocks only for VTIME before returning
    {
      // read binary stream
      // (type-punning: aliasing with char* is ok)
      (reinterpret_cast<char*> (&_input_buf))[_input_pos++] = c;
```

```
      if (_mode == YAW_PITCH_ROLL) {  // 3 floats
        if (_input_pos == 12) // we received a full frame
        {
          // convert endianess if necessary
          if (_big_endian())
          {
            _swap_endianess(_input_buf.ypr, 3);
          }

          // invoke callback
          data(_input_buf.ypr);

          _input_pos = 0;
        }
      }

      /****************************************/
      // Used to redirect values to main routine
      else if (_mode == ROBOSUB)
      {
        if(_input_pos == 36) // we received a full frame
        {
          if (_big_endian())
          {
            _swap_endianess(_input_buf.amg, 9);
          }

          setImuData(_input_buf.amg); // changed from ypra
          // data(_input_buf.ypra);

          _input_pos = 0;
        }
      }

      /****************************************/

      else { // raw or calibrated sensor data (9 floats)
        if (_input_pos == 36) // we received a full frame
        {
          // convert endianess if necessary
          if (_big_endian())
          {
            _swap_endianess(_input_buf.amg, 9);
          }

          // invoke callback
          data(_input_buf.amg);

          _input_pos = 0;
        }
      }
    }
    // error?
    else if (result < 0)
    {
      if (errno != EAGAIN && errno != EINTR)
      {
        error("Can not read from serial port (3).");
        return arg;
      }
    }
  }

  return arg;
}

// These functions added for RoboSub use
void RazorAHRS::setImuData(const float data[])
{
      // format is {accel-x, accel-y, accel-z; yaw, roll, pitch; gyro-x, gyro-y, gyro-z}
      for(int i = 0; i < 9; i++)
      {
        //std::cout << "In loop at " << i << ": " << data[i] << "\n";
        this->myImuData[i] = data[i];
      }
}
```

```
float RazorAHRS::getXAccel()
{
	return this->myImuData[0];
}

float RazorAHRS::getYAccel()
{
	return this->myImuData[1];
}

float RazorAHRS::getZAccel()
{
	return this->myImuData[2];
}

float RazorAHRS::getYaw()
{
	//std::cout << "Razor.cpp file Yaw: " << this->myImuData[0] << "\n";
	return this->myImuData[3];
}

float RazorAHRS::getRoll()
{
	return this->myImuData[4];
}

float RazorAHRS::getPitch()
{
	return this->myImuData[5];
}

float RazorAHRS::getGyroX()
{
	//std::cout << "Razor.cpp file GyroX: " << this->myImuData[6] << "\n";
	return this->myImuData[6];
}

float RazorAHRS::getGyroY()
{
	//std::cout << "Razor.cpp file GyroY: " << this->myImuData[7] << "\n";
	return this->myImuData[7];
}

float RazorAHRS::getGyroZ()
{
	//std::cout << "Razor.cpp file GyroZ: " << this->myImuData[8] << "\n";
	return this->myImuData[8];
}
```

5. PressureSensorArduino.ino

This sketch is flashed in the UNO to get the voltage from the pressure sensor and turn it into a usually useful depth. It takes advantage of the depth sensor's linear nature.

```
/*
 Author: Julius Cobb
 Modified: Kevin Matungwa, Dennis Boyd

 Date: 2/9/2015

 Description: Code for the UNO to interface with the Levelgage depth
 sensor. It returns a DC voltage of 0-5V. The UNO gets from
 this an int 0-1023. This is converted back to a voltage.
 Then that is converted to feet using the data Bjorn
 gathered.

 Version: 1.0 - initial testing of the sensor using Serial Monitor
 Version: 2.0 - Updated for new Levelgage pressure sensor
 Version: 2.1 - Baud rate set back to 9600
 */

double feetUnderwater = 0.0;
int sensorOutput = 0;
double feetperVolt = 8.78; // calculated reative to the Morcom max depth

void setup()
{
      Serial.begin(9600);
}

void loop()
{
      sensorOutput = analogRead(0); // Analog Pin 0
      feetUnderwater = calcFeetUnderwater(sensorOutput);
      Serial.println(feetUnderwater);

      delay(50); // was 500
}

double calcFeetUnderwater(int sensorOutput)
{
      // sensorOutput will be between 0:1023, so map it to a voltage between 0:5
      double outputVoltage = sensorOutput *(5.0 / 1023.0) ;
      double feet = outputVoltage * feetperVolt;
      //Serial.println(outputVoltage); //for debugging

      return feet;
}
```