Carnegie Mellon University

16-681

MRSD Project I

Task 4 Sensors and Motor Control Individual Lab Report 1 Team C - COBORG

Jonathan Lord-Fonda
Teammates: Husam Wadi, Feng Xiang, Yuqing Qin, Gerry D’ascoli

February 25, 2020

Table of Contents

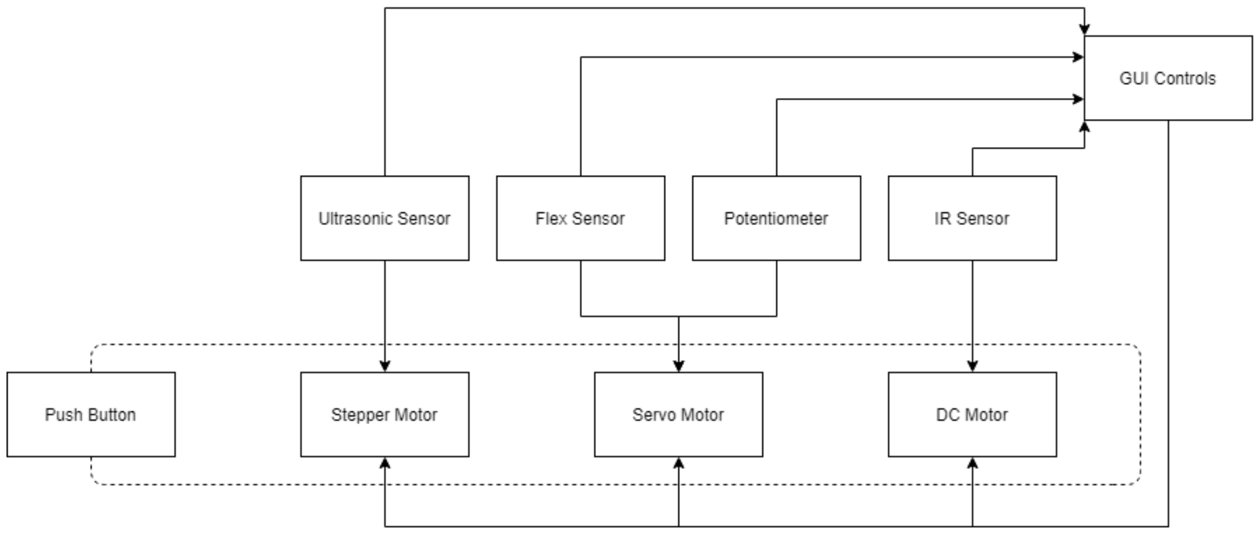
1. Individual Progress

1.1. Sensors & Motors Lab

My responsibilities for this project included soldering the DC motor controller board, developing code for the stepper motor, the ultrasonic sensor, and the push button with debouncer, and also creating the software framework for the project as a whole and integrating most of the code.

Figure 1 demonstrates the structure of our Arduino code. As our team began to understand how to pull information from the sensors and operate the motors, it was important to lay out the framework for how they would all mesh together. Establishing a consistent understanding of which sensors would relate to which motors and how the push button would control everything allowed individual teammates to complete their tasks separately and then combine their sections with few issues. In addition to this I created the first integrated code structure, labelling sections for each sensor and motor and establishing the overarching if statement that would control the output to the motors.

Figure 1 - Arduino Code Structure



The above figure demonstrates the movement of information through the code. The sensors generate information and send them both to their respective motors and the GUI controls. The GUI displays the information from the sensors and also generates motor commands for each of the motors based on user input. The push button controls which information the motors use to operate.

The push button is a simple sensor that sends a high signal when pressed and a low signal the rest of the time. I attached the push button pin to an interrupt that checked for its rising. Within the interrupt the program checks to see if enough time has passed since the previous state change and, if so, changes the state and sets the time of the previous press to the current time. If sufficient time has not passed since the last change nothing happens and the interrupt is exited. Including this check in the interrupt prevents switch bouncing from changing the state multiple times. Code for this section can be found in Appendix 1.

The ultrasonic sensor measures distance by sending a high signal to the trigger pin for 10 microseconds and then reads the response as a digital pulse from the echo pin. The distance measured by the ultrasonic sensor is represented by the length of time that the echo pin maintains a high signal. The distance in centimeters is given by multiplying the measured time by 0.034 (due to the speed of sound) and dividing it by 2 (due to travel time both to and from the object). After receiving a measurement from the ultrasonic sensor, the datum is sent through an averaging filter that stores the most recent ten recorded measurements and returns their average. Using the average result acts as a low-pass filter, helping to remove noise from the sensor. Upon receiving the filtered result, upper and lower bounds are applied to ensure that we stay within the usable range. Code for this section can be found in Appendix 2.

After receiving the ultrasonic sensor reading, and assuming that the Arduino is in state “0” (reading signals from the sensors and not the GUI), commands are given to the stepper motor to rotate to the measured position. The range of the ultrasonic sensor (from 0 to 100 cm) are mapped to the positions of the stepper motor and stored as the desired position. If the desired position is larger than the current position of the stepper motor (which is initialized at 0), then the direction pin is set to high, otherwise it is set to low. After the direction of the stepper motor is determined the motor takes steps until the desired position is reached. Steps are taken by using a for loop to alternate between high and low signals sent to the step pin, with 1000 microsecond delays between each. Once the desired position is reached, the current position is updated to it. Code for this section can be found in Appendix 3.

1.2. MRSD Project

Since the CODR report, my responsibilities for the COBORG project have included creating a Bill of Materials (BoM) for the system as is, creating an initial ROS node map for the system, sketching out the main state node for the system, and testing the arm’s max lift force at full extension.

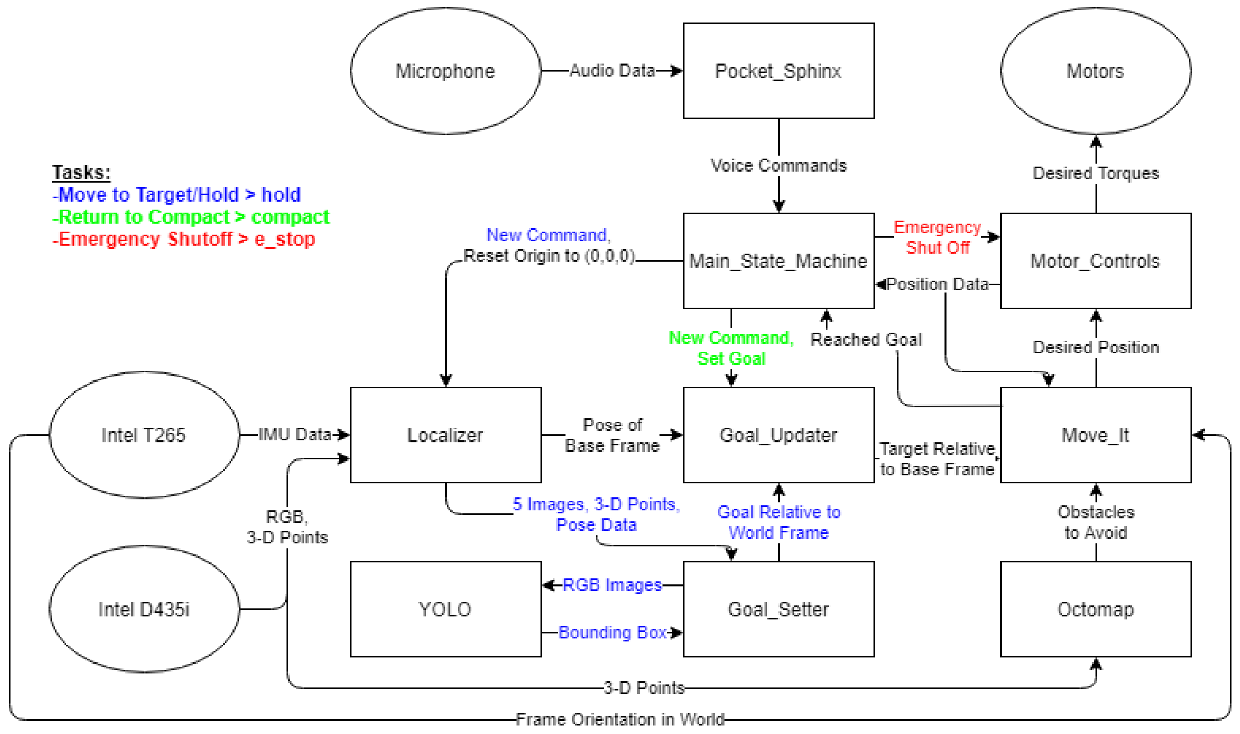
For the BoM, I listed each individual component currently a part of the COBORG system and grouped them together in sensible clusters of larger constructs. For now I did not include the internal electrical components due to a lack of detail on our team’s current circuit diagram.

In order to test the arm’s max lift force at full extension I went to the lab with my teammates and we booted up the system and extended the arm straight out. Once the arm was fully extended I pushed on the end of the arm until it began to move and noted the highest back-torque displayed by the motors in the HEBI interface. The highest torque listed was 4.36 Nm which, given an arm length of 28 inches, leads to ~3 lbs. of max lift force generated by the motors. This was an important statistic to test early because it aligns directly with our requirement that the arm will be able to lift at least 2 lbs. at full extension. A failure to meet this requirement would require an immediate response from the team.

Similar to my work creating a code structure for the Sensors and Motors Lab, I realized that we would need to create a ROS node map so that the team understood what functionality was required, what information was available, and how each of these would interact with one another. I spent some time thinking through the steps the COBORG would have to take to accomplish its functions and wrote out a flow chart describing the different nodes and how they would interact with one another. Figure 2 contains the semantic description of how the system will work. Color coding highlights the paths followed by unique commands. Creating the diagram also helped highlight the difficult parts of the system that will require the most work from the team, namely the Goal_Setter, Localizer, and Move_It (Note that the Octomap node and collision avoidance is currently scheduled to be implemented in the fall, but a full plan was created so as to accommodate future integration). A full description of how the code will function is included in Figure 2's caption. Figure 3 contains the topic description of how the system will work. The names of the topics listed may change or be reorganized, but it is imperative to have a standard that the team can reference when writing code for their individual nodes. Having a centralized plan allows individuals to publish and subscribe to topics in their nodes and bring them together seamlessly into the final system. Buried in these two figures are some decisions made through team discussion, such as to treat the relationship with YOLO as a publisher/subscriber relationship instead of a service. Treating YOLO as a service would make more sense with the code logically, since we only need it to analyze a few images, and then continue on with the program, but YOLO as written is a complex node that runs on a publisher/subscriber relationship. Since the YOLO node was already working we decided not to try to change it so that it could be easily utilized by our code. The downside is that we will have to ensure that the correct number of messages is always sent to YOLO and if another function needs to use YOLO we will have to create a second node or at least separate topics to accommodate the function without polluting the initial function.

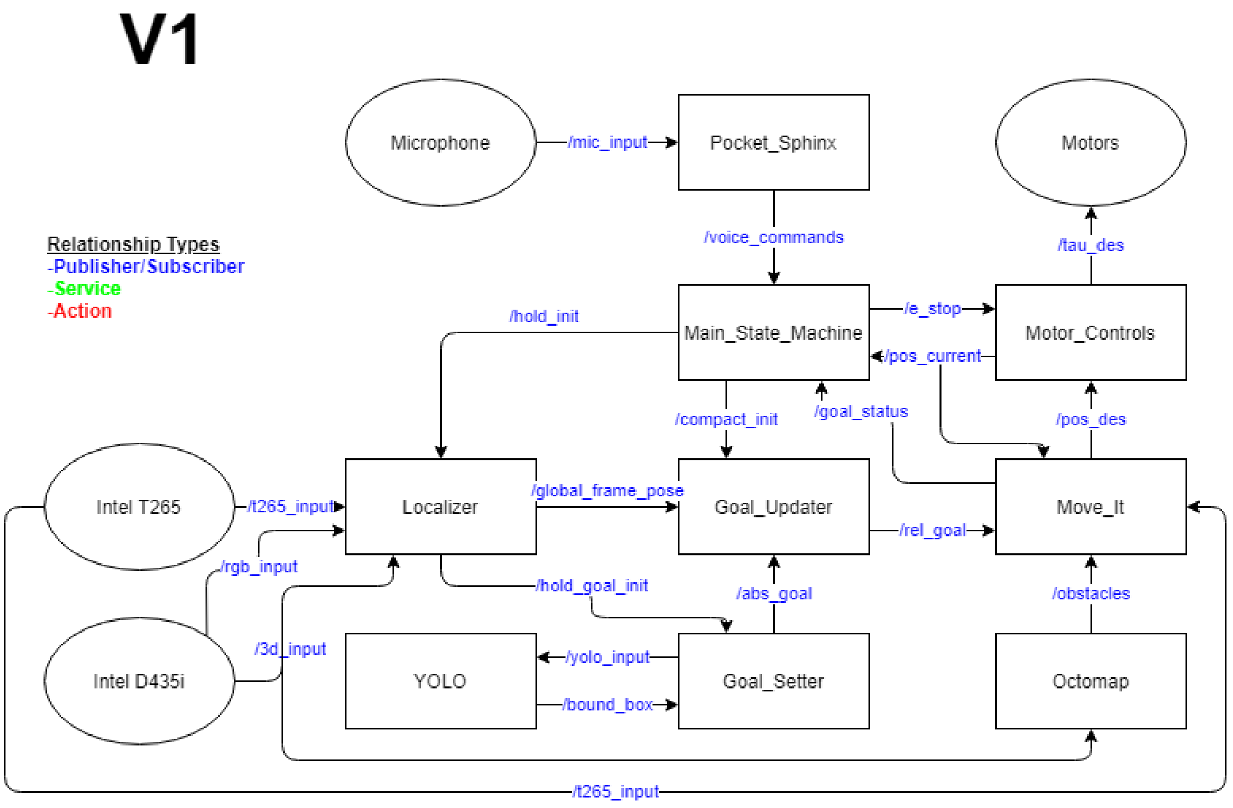
Figure 2 - ROS Node Map, Semantic

V1



The above figure represents the flow of information through our ROS system. Audio information first comes in through the microphone and is analyzed by the Pocket_Sphinx node, which sends voice commands to the Main_State_Machine node. The Main_State_Machine node determines whether to enact or ignore the commands based on whether the previous command was finished. An emergency shut off command will go directly to the motors, turning them off. A compact command will tell the Goal_Updater to set a pre-defined, relative goal for the arm. The most complex command, hold, is sent to the Localizer. The Localizer takes input in the form of RGB images and point-cloud data from the Intel D435i camera and local position data from the Intel T265's integrated visual SLAM. Combining this information together the Localizer node constantly publishes the current pose of the COBORG backpack in the world frame to the Goal_Updater and, when initially commanded by the Main_State_Machine, sends a fresh set of images, 3-d points, and local pose data to the Goal_Setter node. The Goal_Setter sends the images to YOLO, which analyzes them for human hands and returns the bounding boxes. The Goal_Setter integrates the 3-d point cloud, pose data, and bounding box from YOLO each image to determine where the hands are in the world frame and where the robot's goal should be in the world frame. The various goals are analyzed, checking for union and intersection, to throw out outliers and determine the true goal in the world frame. Goal_Setter sends the new goal for the command to Goal_Updater which, using the relative pose data from Localizer, creates a homography to transform the global goal to a local goal for the robot. Upon receiving the goal from Goal_Setter it continues publishing the updated local goal constantly until sent a new command from Main_State_Machine or receiving a new goal from Goal_Setter. The Move_It node receives the local goal from Goal_Setter and plans a path for the COBORG arm to that goal using the current motor position data from Motor_Controls and obstacle information from Octomap. As soon as it finishes and publishes a path, it begins generating a new one with the most recent data. A fast Move_It node will be required to ensure that we can account for movements of the robot. Upon securely reaching the intended goal Move_It will let Main_State_Machine know that the COBORG is ready for its next task. The Motor_Controls node will most likely use impedance control to constantly publish directions to the motors to follow the most recent path it has received from Move_It. It will also publish the current motor state to Move_It, for the reasons described above, and to Main_State_Machine, so that if the motors violate the safety bounding box of the human "trunk" the Main_State_Machine can immediately shut down the Coborg, preventing user injury.

Figure 3 - ROS Node Map, Topics



The above figure lists the topic names to be published and subscribed to by each of the nodes. It also establishes the type of relationship that each of the nodes will have, currently entirely publisher/subscriber relationships.

In addition to the activities listed above, I also sketched out a first-draft, semantic description of how I expect the Main_State_Machine node will function. The draft can be found in Appendix 4.

2. Challenges

2.1. Sensors & Motors Lab

One of the difficulties that I encountered was in soldering the DC motor controller board. I have some moderate experience with soldering, but I certainly got better over the course of soldering the board. When it came time to solder the L298 chip, with all of its closely-seated pins, I ended up connecting a number of them with solder. After a fruitless half hour of trying to remove the solder I turned to Husam for help and with his experience, he was able to remove the solder that I couldn't. Sometimes an expedient solution is important to remove barriers to progress.

Another difficulty that I faced in setting up the code structure was how to allow all of the sensors and motors to operate simultaneously without delaying the other sensors and motors. To this end we tried to remove loops from most of the code to prevent delays and hang-ups. Additionally, I changed the encoder interrupts to be detached while the board was receiving instructions from the ROS GUI to prevent any interruption of function on the ROS side of things.

2.2. MRSD Project

The biggest challenge that I faced for the COBORG was deciding how to integrate the YOLO, Localizer, and Goal_Setter functionality. The major concern was that we needed to keep the 3-d point cloud and pose data alongside the RGB images, but YOLO only accepts RGB images. Additionally, we needed Localizer to tie all of the information together, but it also needed to run quickly, fast enough to keep up with the T265's publishing of local position. I went through a few iterations before landing on a solution that I believe will retain the data that we need, while also allowing a continuous flow of local states in the world frame. I still have some doubts about how to integrate Move_It into the map as a whole and whether the Main_State_Machine should be doing more or have increased functionality.

3. Teamwork

3.1. Sensors & Motors Lab

When I ran into problems with the DC motor control board Husam and Gerry helped fix my soldering job and gave me tips about how to solder more effectively. After it was assembled, Gerry, Husam, and I tested the board to make sure that it worked correctly.

Gerry and I discussed how to implement code for the DC motor encoder to track the motor's position, even while the loop was continuously running. We also discussed how to implement the PID control.

Early on I established the structure of the code, as well as a map displaying its functionality. This helped all team members understand where and how to integrate their sections. It was particularly helpful for Husam so that he knew how he should connect ROS to the rest of the Arduino code.

3.2. MRSD Project

In order to test the arm's max lift force, Jason and Husam helped me boot up the robot, move it into position, and use HEBI's GUI to read out the back-driven torques on the motors.

Yuqing helped me install Linux and ROS on my personal flash drive, despite running into many, many problems and having to reformat my flash drive ~6 times.

I discussed my plans for the node map, particularly the vision system, with Yuqing and Husam. My initial idea was to treat YOLO as a client, but concerns from Yuqing over re-writing the node convinced me to change it to the publisher/subscriber relationship that it currently has. Our conversation also led to a new understanding and restructuring of the Localizer, Goal_Setter, and YOLO node structure. I discussed my plans for the node map, particularly the path planning and control system with Jason. He explained the functionality of Move_It and Octomap and we came to the conclusion that we agreed on how the path planning nodes should be set up. I discussed my plans for the node map, particularly the voice system, with Gerry. We agreed on the implementation and he explained that only a small tweak would be required for it to function as envisioned. Discussing with the team was key for the node map because it is the plan that we all need to follow if we intend to bring the system together as a whole. Additionally, leaning on the expertise of my teammates ensured that I didn't push forward with an idea that ended up proving faulty.

Jason created a 3-d model of our current COBORG system and asked me to check it over. I reviewed it, said it looked great, but also brought up the concern of the assembly being static. Since I knew that we intended to use the CAD model to export to the URDF, I wasn't sure if this was a concern or not. Bringing up issues now, before they're a problem, is key.

4. Plans

4.1. Sensors & Motors Lab

The Sensors & Motors Lab shares little relevance to our project, so we currently have no plans to further build on its development. This is largely due to the fact that we're using HEBI motors, which provide a simple, robust interface and that our main sensor is a camera with 3-d point cloud data, which was not a part of the Sensors & Motors Lab project.

4.2. MRSD Project

One of my future tasks for the COBORG project is updating the BoM with more details, as well as the electrical components excluded on the first pass. Additionally I intend to fully write out a functioning Main_State_Machine node and integrate it with Gerry's Pocket_Sphinx node. I would also like to learn more about Move It and Octomap with Jason and sketch out some rough details for those nodes.

5. Appendices

5.1. Appendix 1 - Button Debounce Code

```
// Function to debounce switch and change state
void State_Change() {
  if ((millis() - B0Press) > debounceDelay) { // Check to see when button was last pressed
    state += 1;   // If Button 0 is pushed, increment the state
    if (state == 2){
      state = 0;    // If currently state 1, cycle back to state 0
      B0Press = millis(); // Grab the time at which the button was pressed
    }

  }
}
```

5.2. Appendix 2 - Ultrasonic Sensor Code

```
  // Ultrasonic Sensor
    digitalWrite(triggerPin, LOW);
    delayMicroseconds(2);
    digitalWrite(triggerPin, HIGH);
     delayMicroseconds(10); // Hold trigger for 10 microseconds, which is signal for sensor to measure distance.
    digitalWrite(triggerPin, LOW);
    unsigned long durationMicroSec = pulseIn(echoPin, HIGH);
    int avgDurationMicroSec = avgFilterUltra((int) durationMicroSec);
       double distanceCm = avgDurationMicroSec / 2.0 * 0.0340;   // 340m/s 0.0340cm/microsec
    if(distanceCm > 100.0){//bounds check
        distanceCm = 100.0;
    }
    if(distanceCm < 0){
        distanceCm = 0;
    }
```

5.3. Appendix 3 - Stepper Motor Code

```
// Update Stepper Motor
  desiredStep = round(map(distanceCm, 0, 100, 0, stepsPerRevolution));
  if(desiredStep > currentStep) digitalWrite(dirPin,HIGH);
  else digitalWrite(dirPin,LOW);
  for(int x = 0; x < abs(desiredStep-currentStep); x++){
     digitalWrite(stepPin, HIGH);
     delayMicroseconds(1000);
     digitalWrite(stepPin, LOW);
     delayMicroseconds(1000);
  }
  currentStep = desiredStep;
```

5.4. Appendix 4 - Semantic First Draft of Main_State_Machine Node

```
Main_State_Machine Intuitive Draft

// States:
// hold > Move to and hold plate
// compact > Return to home position
// e_stop > Shut off power to motors

On launch{
    state = compact
    Begin subscribing to /voice_commands
    Begin subscribing to /pos_current
    Begin subscribing to /goal_status
    Start publish to /hold_init
    Start publish to /compact_init
    Start publish to /e_stop
    goal_status = 0
    Send command to /compact_init
}

On publish in /voice_commands{
    if voice_command == e_stop{
          state = e_stop
          send stop command to /e_stop
    }
    if voice_command == hold{
          if goal_status == 1{
                state = hold
                send initialization command to /hold_init
          }
    }
    if voice_command == compact{
          if goal_status == 1{
                state = compact
                send initialization command to /compact_init
          }
    }
}

if pos_current interferes with "human core" restriction{
    state = e_stop
    send stop command to /e_stop
}

Loop
```

6. Sensors & Motor Control Quiz

1) Reading a datasheet.
   a) What is the sensor's range?
      +/-3.6 g
   b) What is the sensor's dynamic range?
      7.2 g
   c) What is the purpose of the capacitor C_{DC} on the LHS of the functional block diagram on p.1? How does it achieve this?
      The capacitor maintains the voltage source at 3V, removing dips and noise. If the voltage source dips below 3V, the capacitor will act like a small battery, outputting power to the circuit to make up the difference. This will only function at high frequencies.
   d) Write an equation for the sensor's transfer function.
      V is the output voltage in Volts and x, y, and z acc are the measured accelerations in g's.
      $x_{acc} = (V - 1.5)/0.3$
      $y_{acc} = (V - 1.5)/0.3$
      $z_{acc} = (V - 1.5)/0.3$
      $Total_{acc} = (x_{acc}^2 + y_{acc}^2 + z_{acc}^2)^{0.5}$
   e) What is the largest expected nonlinearity error in g?
      $3.6V/0.3mV/g * 0.003 = 0.036g$
   f) How much noise do you expect in the X- and Y-axis sensor signals when the sensor is excited at 25 Hz?
      $150/(25)^{0.5} = 30ug$
   g) How about at 0 Hz? If you can't get this from the datasheet, how would you determine it experimentally?
      This is not possible to determine from the data sheet, because it suggests an answer of infinite noise. I would test this by setting the accelerometer on a motion-cancelling platform and reading the sensor's output, recording it as the error.
2) Signal conditioning
   a) Filtering

i) Name at least two problems you might have in using a moving average filter.
A moving average filter with a large window size could result in significant lag. A moving average filter is also prone to be thrown off balance by individual errors (such as the occasional 0 or huge number) thrown by the sensor.

ii) Name at least two problems you might have in using a median filter.
A median filter with a large window size could result in significant lag, preventing the device in question from reacting quickly or even moving at high rates of change at all. A median filter also entails a significant computing cost.

b) Opamps

i) In the following questions, you want to calibrate a linear sensor using the circuit in Fig. 1 so that its output range is 0 to 5V. Identify in each case: 1) which of V1 and V2 will be the input voltage and which the reference voltage; 2) the values of Rf/Ri and the reference voltage. If the calibration can't be done with this circuit, explain why.

(1) Your uncalibrated sensor has a range of -1.5 to 1.0V (-1.5V should give a 0V output and 1.0V should give a 5V output).

$V_{out} = (V_2 - V_1)R_f/R_i + V_2$

$0 = (-1.5 - V_1)R_f/R_i - 1.5$

$-1.5R_i/R_f - 1.5 = V_1$

$5 = (1 - V_1)R_f/R_i + 1$

$5 = (1 - (-1.5R_i/R_f - 1.5))R_f/R_i + 1$

$4 = (1 + 1.5))R_f/R_i + 1.5$

$2.5 = (2.5))R_f/R_i$

$R_f/R_i = 1$

$5 = (1 - V_1)1 + 1$

$V_1 =- 3$

V2 will be the input voltage, V1 will be the reference voltage at 3V, and Rf/Ri will be 1.

(2) Your uncalibrated sensor has a range of -2.5 to 2.5V (-2.5V should give a 0V output and 2.5V should give a 5V output).

$V_{out} = (V_2 - V_1)R_f/R_i + V_2$

$0 = (V_2 + 2.5)R_f/R_i - V_2$

$V_2 = (V_2 + 2.5)R_f/R_i$

$V_2/(V_2 + 2.5) = R_f/R_i$

$5 = (V_2 - 2.5)R_f/R_i + V_2$

$5 = (V_2 - 2.5)V_2/(V_2 + 2.5) + V_2$

$5 = (V_2^2 - 2.5V_2)/(V_2 + 2.5) + V_2$

$V_2 \approx 4.04V$

$0 = (4.04 + 2.5)R_f/R_i - 4.04$

$4.04 = (6.54)R_f/R_i$

$R_f/R_i \approx 0.6177$

V1 will be the input voltage, V2 will be the reference voltage at roughly 4.04V, and Rf/Ri will be roughly 0.6177.

3) Control

a) If you want to control a DC motor to go to a desired position, describe how to form a digital input for each of the PID (proportional, integral, Derivative) terms.
A motor encoder should be used to keep track of the position. Once the motor has been set to “home” the position can be set to 0. As the motor turns, the encoder outputs will change. Comparing the new output to the old output will show which direction the motor has turned. Every time the motor encoder output changes, increment or decrement the motor’s position by 1. Dividing this tracked position by the number of encoder states in a given revolution will tell you how many revolutions the motor has undergone and which direction (based on sign). In order to get velocity, record the amount of time between each change in the encoder’s signal. One divided by the number of encoder states per revolution divided by the number of seconds since the last encoder state change, will give the angular velocity of the motor in revolutions per second. In order to get an integral term describing the motor’s error, record the error between the motor’s current position and its desired position and multiply this by a short timestep. Add this number to a record of the integrated error and repeat for every timestep.

b) If the system you want to control is sluggish, which PID term(s) will you use and why?
I would use proportional control because it would generate outputs to close the gap between the motor’s current and desired position, speeding it to its goal.

c) After applying the control in the previous question, if the system still has significant steady-state error, which PID term(s) will you use and why?

I would use integral control because it would generate outputs to correct accumulated errors, nudging the motor to its goal if it has an error built up over a long period of time.

d) After applying the control in the previous question, if the system still has overshoot, which PID term(s) will you apply and why?
I would use derivative control because it would generate outputs to counteract the speed of the motor (if the desired speed is 0), slowing it down so that as it approaches the goal, and proportional control outputs reduce to 0, the derivative controller can apply force to slow the motor down and keep it from rushing past the goal.