RBE 2002 (B-24)
Lab 5
Lifting and Weighing

Last modification: November 20, 2024

Be sure to follow along with the Worksheet at the end of this document as you do the activities.

You will work in pairs to complete the Activities and Worksheet. Do not take a 'divide-and-conquer' approach – it is important that each student masters the material.

1 Introduction

For this week's activities, you will incorporate an RC servo motor and a load cell into your Romi and program the Romi to both lift and weigh a garbage can. With alignment proven last week, this week's main deliverable is to show that your Romi can find a garbage can, align the lifter, pick up the garbage, and weigh it.

Be gentle with the lifters. The load cells are rated for 1kg max, which means that if you subject them to significantly more stress (typically twice the rating), you will damage the internal strain gauges.

There is a base set of activities to earn 90 points for the lab. To receive a higher score, you will need to go beyond expectations and complete one or more extra activities. Suggested extra activities can be found near the end of this document.

2 Background

Background material can be found in the lectures and associated slides. Additional details, along with links to datasheets and such, can be found in the specific activity sections below.

Lab Activities

Note: One of your deliverables will be a state transition diagram for the entire process from searching to a completed weight measurement. We do not ask for intermediate diagrams, but you should build one up as you go – doing so will make the programming phases much easier.

3 Lifting

You should have attached a lifter mechanism to your Romi last week, but without the load cell. Here, you will attach the load cell and hook and set up the servo motor to actuate the lifter.

3.1 RC Servo Motor

We don't cover the RC Servo Motor in great depth here, nor the mechanics of the lifter. If you have taken RBE 2001, you can certainly analyze the mechanism and torque requirements, but since this is a course in sensing, we don't cover the details here. However, we do cover the interface, as there are important differences from previous implementations you may have used.

RC servo motors[1] are controlled by sending pulses of different lengths to the motors. A typical RC servo will operate on a 20ms cycle. The control pin will be held HIGH for a short period of time (typically around $800 - 2200\mu\text{s}$, though it varies by manufacturer) and then held LOW for the remainder of the 20ms. Short pulses will correspond to motion in one direction and long pulses in the other. A $1500\mu\text{s}$ pulse is a common mid-point.

You may have used the Arduino Servo library in the past, but we don't use it here for two reasons:

- Although the library is flexible – you can use it with most any pin – the way it is constructed leads to occasional jitter of the servo. In short, the library uses an interrupt-based control module, which can lead to delays when toggling pins if the processor is servicing another interrupt. It's typically not a big problem, but you can sometimes hear the jitter.
- More importantly, the Arduino library uses `Timer1`, which is dedicated to motor control on the Romi. It is fairly straightforward to change to `Timer3` – the two have nearly the same specs on the ATmega32U4 – but there is no need to, and the library we provide is cleaner.

Instead, we provide a Servo library that more properly controls a subset of pins directly with hardware timers. The result is glitch-free (timers run in the background), but suffers from the drawback that you can only use certain pins. Nevertheless, you only need to control one servo, so you have plenty of options for implementation.

The library allows you to declare a servo on a specific pin, where each pin is referenced through a distinct class. For example, if you want to put the servo on pin `6`, then you'd use the `Servo32U4Pin6` class.

You have the choice of the following pins:

pin `5` uses `Timer3` and has the fewest restrictions, as well as the highest precision. Because `Timer3` is a 16-bit timer, it can be set up for a resolution of $0.5\mu\text{s}$. This is a good choice if you have no other plans for `Timer3`.

pin `13` uses `Timer4`, which is already set up for a 20ms period for the main Romi control loop. Because `Timer4` is only 10-bits, however, the precision is lower (resolution of $64\mu\text{s}$). **Warning**: pin `13` is active in the bootloading phase – the LED on pin `13` flashes when you upload – which will require you to unplug the servo when you upload code, lest your servo go nuts during uploads.

pin `6` **and pin** `12` also use `Timer4`, so they have the same resolution as pin `13`. The two pins share a the same output compare, except one is inverted. Practically, that means you can use one or

[1] "Servo motor" can refer to many kinds of motors where feedback is built into the motor itself. In our case, the "RC" stands for "radio controll," which indicates that these servo motors are a specific kind of motor intended to work with pulse commands sent from an RC controller.

the other for a servo, but not both at the same time. pin `6` is also connected to the buzzer, so you'll need to cut the buzzer trace to shut it up.

Warning: The servo can draw a fair bit of current. Along with your camera, the two can pull enough power that if your batteries are low, the voltage on the `5V` bus will droop and you'll get a brown out, leading to unexpected resets of your Romi. Fresh batteries will help, of course, but the library is also set up so that the servo motion is slower and smoother to avoid spikes in current draw.

Procedure

1. Wire the servo to `5V`, `GND`, and the signal pin of your choice. There is no good way to plug the three prong servo plug directly into the Romi, so you'll have to decide among:

   (a) Plugging it into a breadboard and then wiring the breadboard to the Romi,

   (b) Pulling the signal line out from the three wire prong and wiring each wire directly to the Romi, or

   (c) Creating a harness to manage the conversion.

   Do not just splice in three jumper wires – they'll come apart too easily.

2. To help keep things organized, you may want to add a new `.cpp` file to manage the servo and the load cell interfaces, for example, `robot-garbage.cpp`.

3. Add an object of the relevant servo class to `Robot`. Be sure to add `<servo>::attach()` to `Robot::InitializeRobot()`.

4. Add a method to class `Robot` to manage the servo. You could pass an angle and then convert that to microseconds – I chose simply to pass the pulse length directly:

   ```
   void Robot::SetLifter(uint16_t pulseLengthUS);
   ```

   (Alternatively, you can skip this function and make calls to the servo object directly.)

5. Add a line in `Robot::RobotLoop()` that will call `<servo>::update()` on the 20ms schedule that is used to manage the line sensors and motors. Dig into `<servo>::update()` to see why we manage the servo in this manner.

6. Add commands to the `TELEOP` mode that allow you to raise and lower the lifter.

7. Experiment with commanding the servo. Note that the default pulse range is $1000 - 2000\mu\text{s}$, which may not drive the lifter through its full range. Explore the library to determine how to increase the range. Attach the load cell and hook and prove that you can lift a garbage can and hold it in the air.

4 Weighing

Once you have lifted a garbage can, you will need to weigh it. To do that, you'll need to connect the load cell to an amplifier and implement a sensor interface.

4.1 Load Cells

We covered the load cells in lecture – review your notes or the slides for more information. The load cells are rated for $1\mathrm{kg}$. **Do not subject them to forces greater than expected from a single garbage can.** It does not take much to break the strain gauges! Also, be careful with the wires. They are fragile. We will have zip ties that can be used for strain relief for the wires.

From the (nearly useless) datasheet, the sensitivity of the load cell is given as $2\mathrm{mV/V}$, which we will take to mean that when loaded to the rated value ($1\mathrm{kg}$), the output voltage will be given by,

$$V_{out} = V_{EX} \times 2(\mathrm{mV/V})$$

Though they don't carry the exact model we're using, SparkFun has a good reference on load cells. Consult your notes or the slides from lecture for details on the load cell that you'll use here.

Procedure

1. Measure the resistance across the red and black wires. Assuming each strain gauge is the same, what is the nominal resistance of *each* strain gauge? What do you expect the resistance to be between the white and green wires? What do you expect the resistance to be between the green and red? Do **not** press on the load cell to try and measure changes in resistance – they are too small and you may damage the load cell while trying.

4.2 Amplification

We also covered amplification in lecture. In short, you will have two options: using the provided `HX711` breakout *or* homebrewing your own instrumentation amplifier circuit.

4.3 The HX711 Amplifier Breakout

The HX711 Amplifier has a built-in 24-bit ADC. Data is retrieved by toggling a clock to get the 24 bits of data and then an additional 1 - 3 toggles to command the next reading. To check for a reading, you will need to call `HX711::GetReading()`, which takes the reading parameter *by reference* and returns `true` if there is new data. You will need to ignore the result when the function returns `false`. To get a more precise reading, you will be expected to average a number of readings, where the exact number is determined below. As such, a good procedure for reading the sensor is:

1. When the lifter reaches the correct position (event), wake up the chip (action), so that it starts to take readings.

2. In the main robot loop, check for and handle readings from the chip. If there is a result, add it to a sum (not a running sum, like with the bias, but just a plain sum). You may *not* use blocking code for this step – i.e., you may not simply write a `for` loop, but instead you must write a proper handler that checks that the Romi is in the `WEIGHING` state.

3. When the last reading is taken (event + guard condition), divide the sum by the number of readings and store the average (action). Put the sensor to sleep (action) and move to the next state.

OK, let's implement the interface.

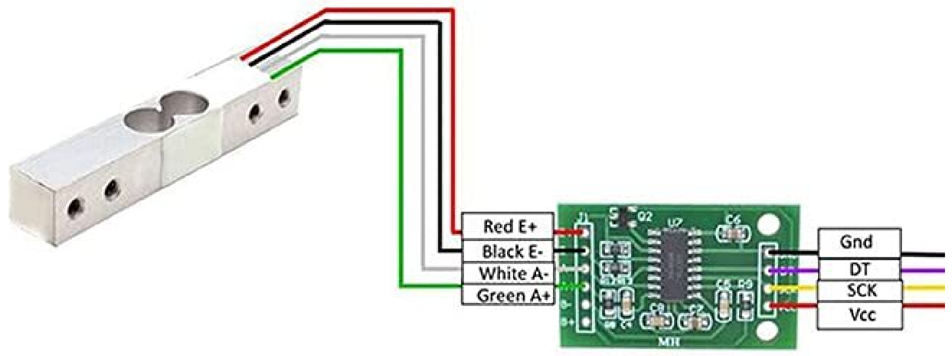

Figure 1: Wiring diagram for the HX711 breakout board.

Procedure

1. Following Figure 1, carefully connect the thin wires coming out of the load cell to the screw terminals on the breakout board. Add a loop in the wires and a zip tie to act as strain relief.
2. Consult the Load Cell repository for how to setup and initialize the HX711 sensor.[2] You can clone the repository if you want, but it's easiest to simply include it through `lib_deps`.
3. Add code to allow you to command the sensor to start taking readings. For example, add an IR remote command that puts the Romi into the `ROBOT_WEIGHING` state.
4. Command the Romi to start taking readings to verify that you have connected everything correctly and that it takes readings.
5. **Gently** press on the hook to verify that the values change as expected.

Once your implementation is nominally working, skip to Section 4.5 to start the calibration process.

[2]The repository also has a nominal library for a ZSC amplifier, but it has not been tested. If you want to try one out, you may for extra points.

4.4 The AD620 Instrumentation Amplifier

The AD620 Instrumentation Amplifier was presented and demonstrated in class. Consult your notes or the slides for details. Unlike the HX711, which has a built in ADC, if you choose to build the circuit, you will read the output of the AD620 through one of the ADC pins on the Romi.

To get a more precise reading, you will be expected to average a number of readings. You may not, however, simply use a `for` loop – although the duration may not be all that long, it's not good practice to use blocking code. As such, a good procedure for reading the sensor is:

1. When the lifter reaches the correct position (event), initiate a timer (action) and move to a `WEIGHING` state. Let's arbitrarily set the timer for $50\mathrm{ms}$.
2. When the timer expires (event), read the ADC (action) and restart the timer (action). Add the result to a sum (not a running sum).
3. Repeat for `N` readings. When the last reading is complete (event with guard condition), calculate and store the average (action), stop the timer (action), and move to the next state.

OK, let's build it.

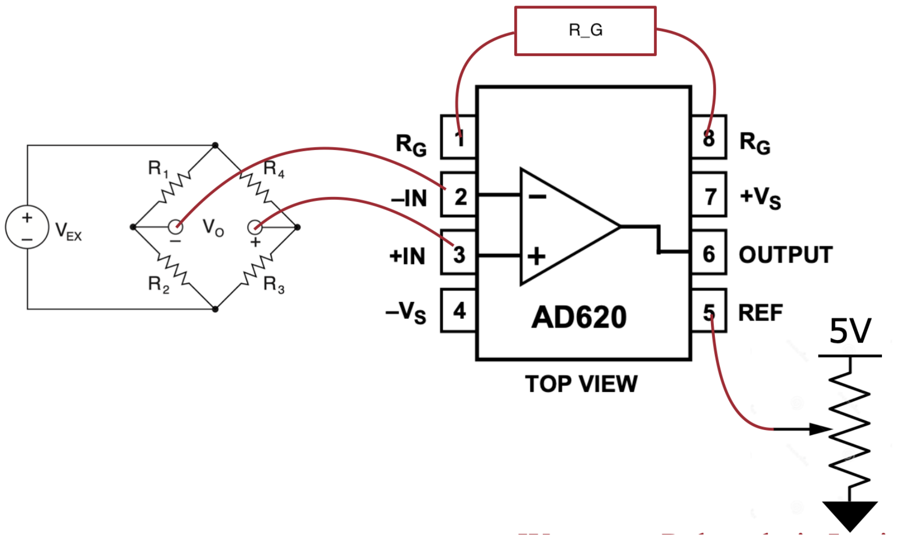

Figure 2: Schematic for implementing the AD620 Instrumentation Amplifier. $+V_S$ should be $5\mathrm{V}$ and $-V_S$ should be connected to `GND`.

Procedure

1. For the given load cell, estimate the gain needed to produce a good output range that can be read by the ADC. Keep in mind that you will need to add an offset to the AD620 to raise the zero point above ground. See the INPUT section of Table 2 - SPECIFICATIONS in the datasheet for the minimum offset.

2. Build the circuit in Figure 2. Choose the resistor, `R_G`, needed to get close to your nominal gain calculated above. You will need to measure the actual resistance with a DMM, since we only have $\pm 10\%$ tolerance resistors. If you need help wiring the load cell (those wires are thin!), see the course or lab staff. You can use a potentiometer for the offset or just hard-wire a voltage divider. In the end, you'll calibrate the sensor so the exact value of the offset is not critical.

3. Attach one channel of an oscilloscope to the differential input pins and the other to the output of the of the amplifier, measured from the *reference voltage*.[3] **Gently** press on the load cell to verify that the gain is as expected.

4. Add a timer (e.g., with the `EventTimer` class) and code to check and handle the timer expiration. You'll need a counter to keep track of the number of readings.

5. Add code to allow you to command the sensor to take a reading. For example, add an IR remote command that puts the Romi into the `ROBOT_WEIGHING` state.

6. Command the Romi to take a set of readings to verify that you have connected everything correctly.

7. **Gently** press on the hook and command another set of readings to verify that the values change as expected.

Once the circuit is nominally working, skip to Section 4.5 to start the calibration process.

4.5 Calibration

In principle, you can calculate the relationship between force on the load cell and the output of your interface circuit to your microcontroller. Regardless of the accuracy of your calculations, however, the readings that you take will have significant error in them. Even if you knew the response of the load cell, all the uncertainty in the gains, imperfections in the chips and resistors, noise in the system, non-linearities, and so forth will lead to errors in the readings. As such, you'll improve performance in two ways:

1. Reduce the effect of noise by averaging multiple readings, and

2. Calibrate the sensor system to account for systematic errors.

Be sure to record your data in the table in the Worksheet as you complete the following activities.

Procedure

1. For each weight in the set of weights provided, hang the weight from the lifter. Command the Romi to take 50 readings for each weight and capture the results. Though inelegant, copy/paste into a spreadsheet should work fine.

2. Create a *single plot* of weight vs. reading. Your plot should have several sets of 50 points, one for each weight (including zero).

[3] I measured from ground in my demo – measuring from the reference is a better demonstration of gain.

3. Find the transfer function as the best *linear* fit for *all* of the data. Do not constrain the intercept to 0.
4. “Invert” the transfer function and program your Romi to output the calculated weight, along with the raw sensor values.
5. Calculate the standard deviation of the readings. Keep in mind you cannot simply find the standard deviation for all the points at once, but you have to compare each reading to the average for that set. You can calculate the standard deviation at each weight or first subtract the mean from each set and then find the standard deviation for the whole set. What is the standard deviation in terms of the weight (not just the raw readings).
6. Find the non-linearity error (again, in terms of weight) as the maximum deviation between the *averages* at each weight and the best-fit line.
7. Find the absolute accuracy of your sensor system in terms of weight. What contributes the most to the overall error? Non-linearity? Noise?
8. Using your estimate of noise and the Central Limit Theorem, discuss the trade-offs between accuracy and processing time. Determine the number of readings your Romi will average and the expected increase in precision. Just so you get some practice, we will specify that you must average at least 4 readings, even if your analysis does not conclude that that is needed.

5 Integration

Integrate the work from last week with the lifter. That is, instead of going to IDLE when properly positioned, add a `LIFTING` state, where the servo is lifts the garbage, followed by the `WEIGHING` state. Demonstrate that your robot can search for, approach, lift, and weigh a garbage can.

Extra Activities

To earn extra points, you have several options, some of which are listed here, but we'll entertain other suggestions. To receive credit, you will need to include appropriate evidence in your Worksheet submission.

- After weighing the garbage, send a bill to the customer. For a small bonus, print the tag ID and the weight to the Serial Monitor. For a larger bonus, send a bill wirelessly through MQTT.
- For a large bonus, build a second amplification circuit, which could be either the AD620 or HX711 – whichever you didn't do first – or the ZSC chip alluded to in the Load Cell library (we have several breakout boards in the back room).
- For a medium bonus, use a different gain for the amplifier circuit and compare your results. You must list out pros and cons and draw conclusions from your findings.
- For a medium bonus, incorporate this week's activities with the overall project goals. For example, first line follow and then find the garbage. Or take the garbage to the dump. You'll need to do that anyway – why not start now?
- For a small bonus, fit a higher-order function to your data to find and implement a non-linear transfer function.

Worksheet

Question

What is the resistance between the black and red wires on the load cell? What is the nominal resistance of each strain gauge? Explain your calculations.

The resistance between the red and black wires on the load cell was ~1 kOhm. Nominally, each strain gauge is 1 kOhm

Question

Calculate the resistance between the red and *green* wires. Measure the actual resistance. How did they compare?

Question

If you built the HX711 circuit, what is the clock rate on the HX711? What is the output data rate? How often will it produce a new reading? How did you determine these values? From the datasheet? Experiments?

Question

If you built the AD620 circuit, what was your target gain for the AD620? How did you calculate the target gain? What was the measured resistance of the actual resistor and what gain does that produce?

Question

Attach the requested graph of the data. Add the equation for the linear best fit to the data or report it here.

Question

Record the summary of your statistical analysis here. Write your answers in terms of "raw" response, either the value returned by the HX711 or your ADC. Note that the average is not the value obtained from your best fit line, but the average of that set of data. To find the non-linearity error, take the difference between the average and the expected value from the best-fit line.

Weight	Avg. Response	Std. Dev.	Expectation (from Best Fit)	Non-linearity Error
0g				
50g				
100g				
150g				
200g				
250g				
300g				

Question

What contributes the most to the overall error? Non-linearity? Noise? Explain.

Question

For the standard deviation of the output (from the previous question), what is the equivalent uncertainty in the input?

Question

Write an expression to calculate the weight from a sensor reading (i.e., invert the relationship above).

Question

How many readings will your Romi average for each weigh cycle? What is the expected improvement in precision? How long will it take to complete a weigh cycle?

Question

Draw out state transition diagrams for searching, aligning, lifting, and weighing a garbage can.

Question

Create a release of your repository as `lastname-firstname-week05`.

Question

Create a video of your robot completing the challenge and submit a clickable link here.

6 Wrapping up

Question

Team Member Contributions

1. Independently, describe how you contributed to the design tasks for this week's activities.

   Student A ______________________

   Student B ______________________

2. For the core activities, assign a weight (as a percentage) to each team member's contribution. The core activities will be worth a total of 90 points. To receive higher marks, you will need to complete one or more extra activities. (Squeeze in a third column if you're a triplet this week.)

Activity	Student A Pct.	Student B Pct.
Mechanical assembly of lifter/servo		
Integration (code) and testing of servo		
Physical integration of amplifier		
Logical (coding) integration of amplifier		
Testing and data collection		
Data analysis		
System integration and testing		
Overall contribution		

Question

If either of you went beyond expectations for this lab, describe any extra work that you did. There is no requirement that both students work on the same activities, so be sure to identify who did what. Write a short paragraph explaining your procedure and verification. Include evidence, either as a video or with code snippets or both.

Appendix

7 Notes on the Servo

Because the servo moves slowly, you'll need to have a `LIFTING` state – you don't want to start weighing the garbage until you're holding it still. You'll also need a `WEIGHING` state, since you'll need to take multiple readings.

The event that causes the transition from `LIFTING` to `WEIGHING` is the servo reaching the target position. Instead of writing a `Servo::checkComplete()` function and calling it over and over, notice that the servo position only gets updated in `Servo::update()`. So a good idea would be to return a `bool` from `update()`, with `true` meaning that the servo has *reached* the target. You will still need to check for the *event* of reaching the target so that you don't call the handler over and over. That means keeping track of if the servo is moving or not. You might add two states to your `Servo` class: `SERVO_MOVING` and `SERVO_AT_TARGET`. Then your logic is "if the servo is in the `MOVING` state and has now reached the target, set the state to `AT_TARGET` and return `true`.