Team 10:
Matthew Haahr (MH)
Brian Shin (BS)
Nick Hom (NH)

RBE 2002: Unified Robotics II—Lab 3: Collision Detection Post- Lab

Contribution statement: For this lab, work was divided well. Matt took on the bulk of the programming and testing of the robot. Brian briefly helped debug some code. Brian and Nick worked on the documentation of this lab and answering the questions.

1. Encoder counts for 90 degree swing turn on different surfaces.
    a. Surface 1: Foisie Table
        i. Encoder counts:
            1. Matt: 1450
            2. Brian: 1450
            3. Nick: 1455
    b. Surface 2: Foisie Concrete Floor
        i. Encoder Counts:
            1. Matt: 1500
            2. Brian: 1520
            3. Nick: 1530
    c. Explanation for different values:
        i. Theoretically, the encoder counts should all be 1440 (the amount for a 90-degree swing turn), however there are physical environment factors that affect the necessary counts to reach 90 degrees robot orientation across different surfaces. The most significant factor is friction. Different surfaces interact differently with the friction between the rubber on the wheels and the actual surface. In our testing, the lab tables responded well with only a small amount of error in the counts. When testing on the floor of the lab, the error was significantly greater due to the floor having less friction and the wheels not gripping as well.

2. Acceleration Data
    a. Plots

Figure 2.1:

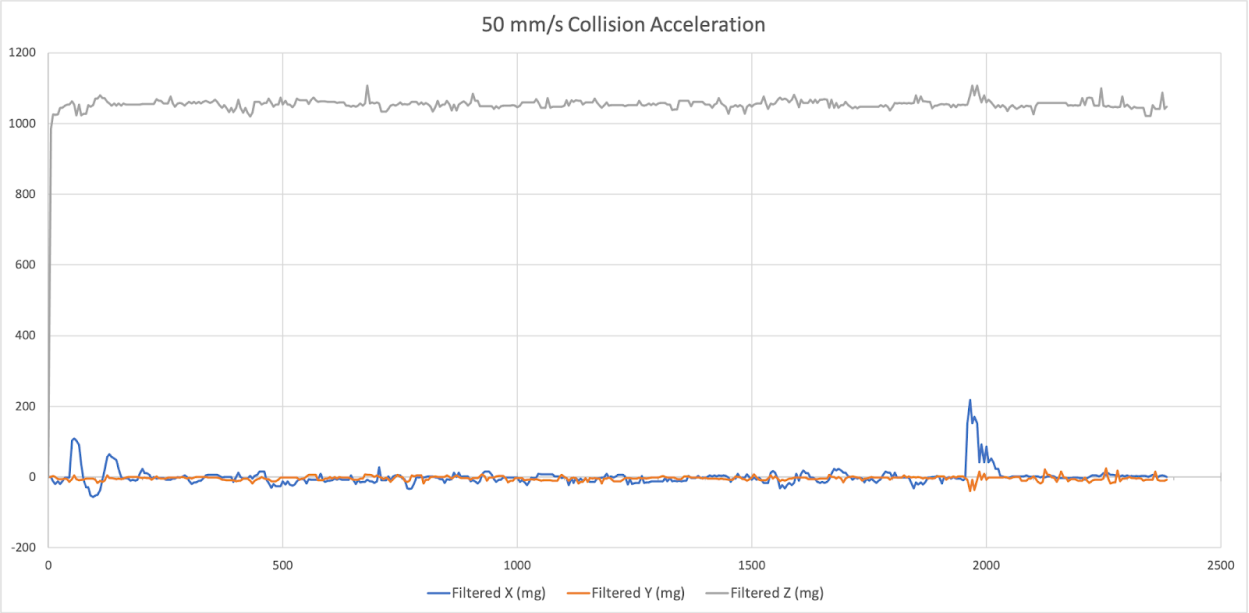

Figure 2.1: x, y, and z acceleration readings for the romi running into a wall at 50 mm/s.

Figure 2.2:

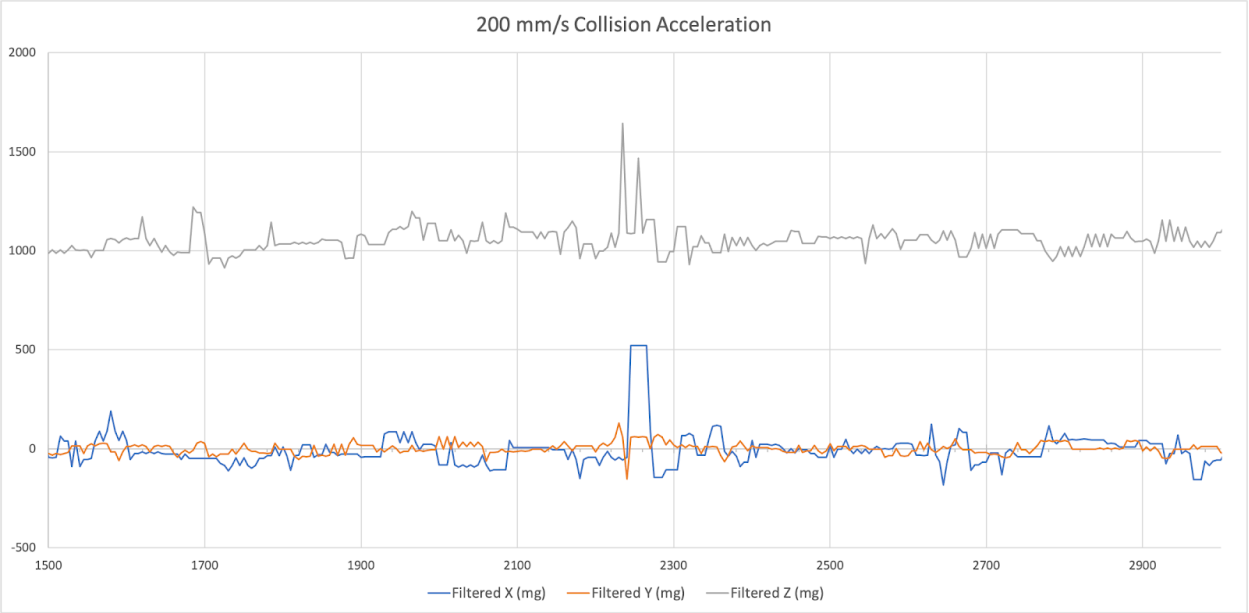

Figure 2.2: x, y, and z acceleration readings for the romi running into a wall at 200 mm/s.

b. Analysis
    i. From the first plot of the acceleration measurements of the Romi driving at 50 mm/s, we can see that the robot experiences a collision on the x axis around 2000 ms, due to the sudden increase in magnitude from near-zero to 200 milli-gravity. There is not a significant change in the y axis. For the z-axis, this collision triggers a small increase in acceleration which can be caused by the angle of the collision imparting a small amount of acceleration upwards and the robot lifting up momentarily. From the second plot, with the Romi driving at 200 mm/s, the results are much more significant. Firstly, the acceleration in all axes display a higher "noise" and greater fluctuation about the respective average values, due to the robot driving faster and producing a greater amount of mechanical vibration. This is not as apparent in the 50 mm/s plot. The collision in this scenario occurs at around 2250 ms, and as seen in the x-axis plot of the acceleration, there is a sharp increase from near-zero to about 500 milli-gravity. Similar to the 50 mm/s plot, the z-axis also experiences a peak at this collision as the robot recoils upwards from the collision. But this collision produced a z-axis acceleration of greater magnitude when compared to the 50 mm/s collision, which makes sense when considering the difference in collision momentum.

3. Collision Threshold
    a. Table

Velocity $\frac{mm}{s}$	Collision Threshold Value
50	300
100	300
150	400
200	400

    b. Testing Method:
        i. Start with 50 mm/s collisions and look at the data from part 2 to find the peak value at the collision. Then start by setting the threshold slightly higher and lower until it works consistently and repeatedly, while not over triggering. Then proceed to the next speed carrying over the threshold for the last velocity for the new speed, then raising it as necessary if over triggering occurs. Sometimes the robot would not detect the collision timing with the polling rate. As the microcontroller is requesting the acceleration from the IMU everytime through the loop, a collision could occur during the period where other operations are occurring, this means that specific peaks can slip through and as part of the median filter, a single high value will be thrown out. A way of fixing this is to have another microcontroller polling the IMU at a very high rate then triggering an

interrupt on the ATMega32u4. This also has issues when changing surfaces as the friction and grip characteristics change making robot dynamics change as well.

4. Median Filters
    a. Median_filter.h
        i. This short header file defines the necessary classes and dependencies needed for the full implementation of the median filter. A line by line analysis is below:

```
#ifndef MEDIAN_FILTER // This line serves as a header guard. This makes
                         sure that the class is defined only once. By using
                         preprocessor directives, it checks if it has
                         already been defined, and if not, it will proceed.
#define MEDIAN_FILTER // This defines a flag for the preprocessor to know
                         whether the header file has already been executed.

#include <Romi32U4.h> // Includes the Romi32U4 library which allows us to
                         use pre-existing methods and other data that
                         pertain to the Romi system.

class MedianFilter { // Defines the class MedianFilter
    private: // Indicates a section for private (class-level access) class
                attributes and methods
        int array[5] = {0}; // Initializes an integer array with 5 elements

    public: // Indicates a section for public (package/project access)
               class attributes and methods
        void Sort(int, int); // Declaration of Sort method, takes in 2
                                indexes to the array initialized in Private
                                and if the associated values are in the
                                wrong order, swaps the values within the
                                array
        void Init(void); // Declaration of Init method, with a void
                            return and void parameters. Nothing is actually
                            done here, it is just good practice.
        int Filter(int); // Declaration of the main Filter method, which
                            returns an integer and also takes in an integer
                            parameter for measurement. Sorts the data
                            stored in the array in ascending order, and
                            takes the median value.
};
```

```
#endif // The last part of the header guard, which encloses all definitions
          and declarations.
```

b. Median_filter.cpp
   i. This source code defines the main functionality and implementation details of the median filter class. A line by line analysis is below.

```
#include "Median_filter.h" // Includes the definitions, dependencies, and
                           declarations from the accompanying header
                           file.

void MedianFilter::Init(void) {
    // no initialization required, but good practice
}

void MedianFilter::Sort(int index_a, int index_b) {
    // The implementation of the Sort method previously declared in the
       header file. Intakes two integers that are used as array indexes for
       the rolling data array and compares them and swaps them if they are
       not in the correct order
    if(array[index_a] < array[index_b]){
        // Compares the values in the
           array at the given indexes and runs the following code if the
           values at the second index is greater
        int temp = array[index_a]; // Creates a temporary placeholder
                                      variable to store the value of the
                                      first index for overwriting.
        array[index_a] = array[index_b]; // Copies the value at the second
                                            index in the array to the first
                                            index in the array and
                                            overwrites the old one
        array[index_b] = temp; // Copies the value from the temporary
                                  placeholder which is the original value
                                  from the first index into the second
                                  index, overwriting the old.
    }
}

int MedianFilter::Filter(int measurement) {
    // The implementation of the Filter method previously declared in the
```

```
    header file. This sorts five data points and finds the median.
  array[0] = measurement; // Set the first element of the array to the
                             inputted new measurement.
  for(int i = 4; i > 0; i--) array[i] = array[i-1];
     // Iterate through the array and set the current element to the
        value of the preceding-index element.

  Sort(0,1); // Call the `Sort` method for the first and second element
                of the array (see documentation for the `Sort` method)
  Sort(3,4); // Call the `Sort` method for the fourth and fifth element
                of the array
  Sort(0,2); // Call the `Sort` method for the first and third element
                of the array
  Sort(1,2); // Call the `Sort` method for the second and third element
                of the array
  Sort(0,3); // Call the `Sort` method for the first and fourth element
                of the array
  Sort(2,3); // Call the `Sort` method for the third and fourth element
                of the array
  Sort(1,4); //  Call the `Sort` method for the second and fifth
                element of the array
  Sort(1,2); // Call the `Sort` method for the second and third element
                of the array

  return array[2]; // Return the third element of the array, which is the
                      median of the data set.
}
```

5. Pick Up Threshold
    a. Threshold
        i. We found the threshold for pickup detection to be $1700\ mg$ which equates to around $16.67\ \frac{m}{s^2}$
    b. Testing Method
        i. The methodology to detect pickup was very similar to that for detecting collisions. First we ran our data acquisition code to find the rough maximum value measured when the robot was picked up. Then we set the threshold slightly higher than that and lowered until the robot consistently detected being picked up while stationary. Then we tested while driving the robot to make sure the robot didn't accidentally detect a collision as wall contact
    c. Why does the robot not stop driving when being picked up during the reverse or turn maneuvers?

        i. The robot doesn't stop driving when picked up during the reverse or turn maneuvers because of the way the state machine is configured. The only time the robot can transition back to the IDLE state is during the DRIVE state, which is where the checking for pickup and button press occur. If the robot is picked up or the button is pressed in the REVERSE or TURN states, nothing will happen. This is for two reasons, the first is that the robot doesn't check either stop condition during those states as well as the actions in both states are blocking, meaning that they operate until the finish and cannot be interrupted.

6. When your robot does not collide with the wall orthogonally, the collision might be not registered using your current collision threshold. Explain an alternative method that could be used to detect collision events
    a. An alternative method to detect collision events would be to calculate the absolute magnitude of the acceleration and compare that to the threshold value. This would enable collision detection from any angle because while the magnitude in a single direction may not exceed the threshold, the magnitude might.
7. Collisions are usually considered catastrophic in robotics. Explain why they are considered catastrophic, making references to momentum, mass, and velocity. Indicate a real-world example of a robot in which a collision would be catastrophic either for people or the robot itself. Why are collisions not catastrophic to our Romi?
    a. When large robots hit people or objects at high speeds, it usually ends in injury or death. Comes with fun lawsuits afterwards too. Jokes aside, these are serious catastrophes. The physics principle of momentum draws a proportional relationship with mass and velocity - thus, a heavy robot moving at a high speed has a great momentum and will impart a devastating impulse to the person or object that it is hitting. If a person were to be hit, that person would likely be seriously injured. Aside from people, the robot would likely be damaged as well since everything inside the robot such as sensitive electronics will experience the same collision. Examples of such a catastrophe in real life are autonomous cars. The implications of a car-human collision are already well known, and if a self-driving car were to fail to detect a person, that would result in serious injury or death.
    b. The Romi is very light and travelling comparatively slowly meaning that these collisions do not have high momentum. This also means that the energy dispersed during the collision is much lower. This means that there is no damage caused to the Romi if used during correct operation as the plastic is more than capable enough of absorbing the energy from the collisions without fracturing or breaking.