Final Report

Team 5

RBE 2002

Hushmand Esmaeili, Sam Alden, Freud Oulon, John Robinson

Table of Contents

Introduction and Overview

Our team aims to use our knowledge of sensors to put on a performance of *Romi and Juliet*, a twist on the famous play by William Shakespeare of a similar name.

This endeavor involves us casting our Romi bots as different characters and then interacting with each other as well as their surroundings to know what to do in each scene. Our robots utilized the Pololu 32U4 control board as our microprocessor and ran on the Pololu Romi chassis. As well as using the AtMega 32U4 as the microprocessor, the control board also has many other integrated devices. Specifically of note in this project was the LSM6 which was used as the IMU. Along with the control board we made use of the OpenMV Cam H7 to detect and follow Aprilags. Communication between devices was done over I2C using the Arduino Wire library.

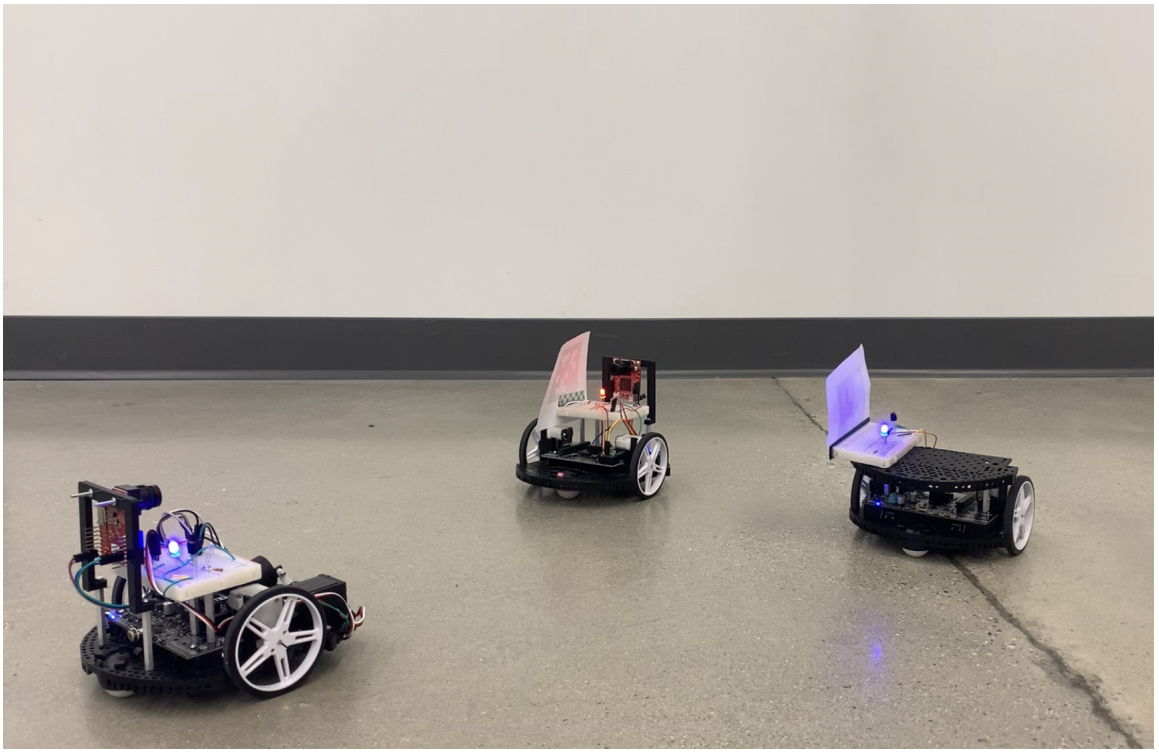


Figure 1: The Fight Scene

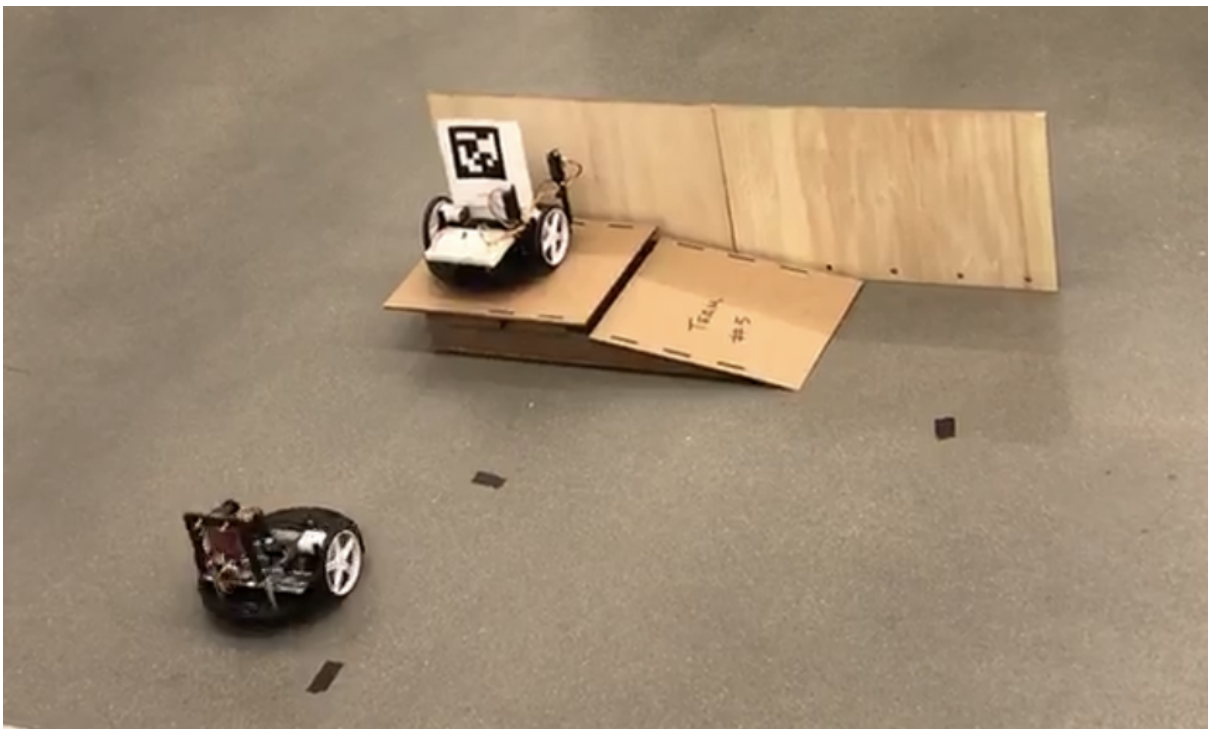


Figure 2: The Balcony Scene

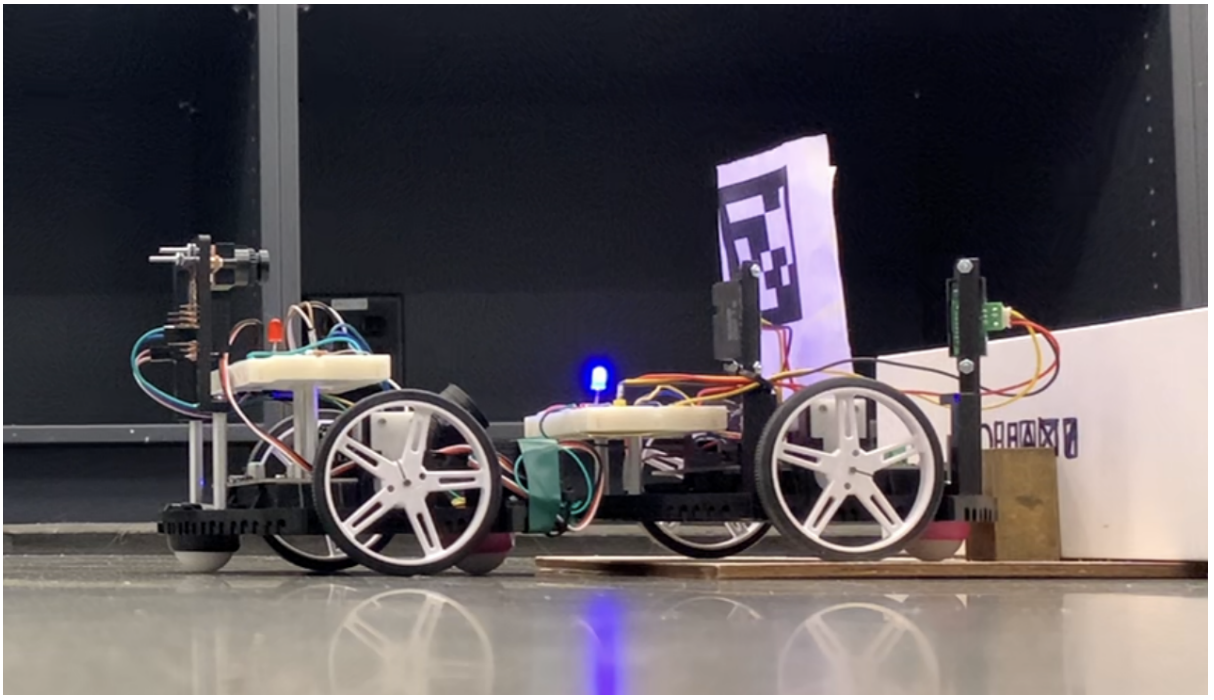


Figure 3: The Final Scene

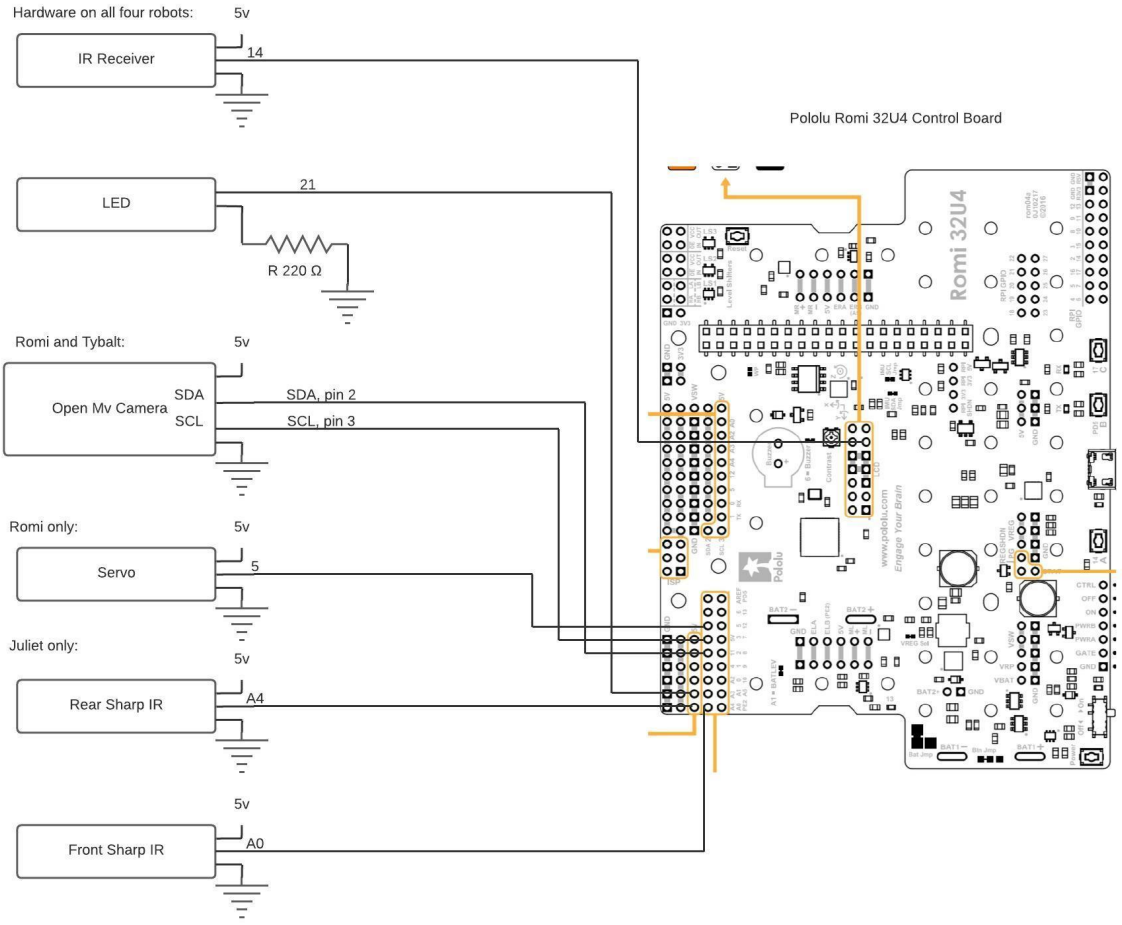


Figure 4: Wiring Diagram of robots

Scene 1: Fight scene

Solutions and Justifications

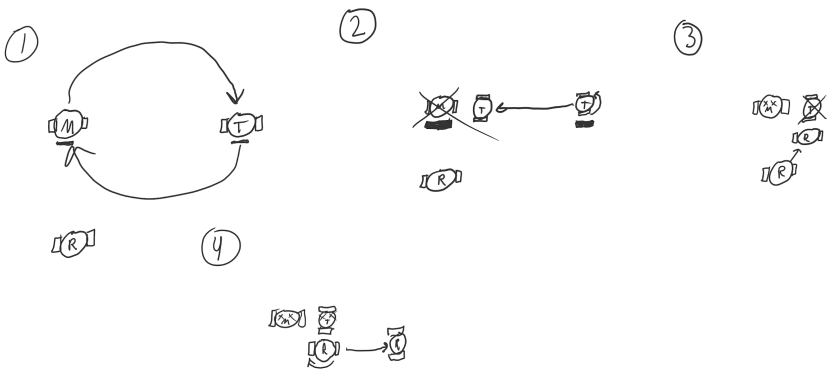


Figure 5: Fight Scene Diagram

In the Fight Scene, Mercoolio and Tybalt circle each other three times. Tybalt turns towards Mercoolio and charges him, and once they collide, Mercoolio's LED turns off indicating that he has died. Romi turns towards Tybalt and charges him. Tybalt's LED turns off to indicate his death. Romi then backs away, turns, and dashes off stage.

One of the core functionalities of this scene was performing a circle maneuver. Initially, we thought of doing this by discretizing a circle into 4-6 points and using inverse kinematics to move to the different points. However, after some initial tests, we determined this was not the best solution. Since our **MoveToPoint()** function used proportional control to get to a point based on the current pose, the robot did not drive smoothly in a circle. Hence, we decided to implement the function **driveCircle()**, which set left and right motor speeds to a constant speed based on the radius of the circle we wanted the robots to drive and at a speed. Then, to keep track of how many circles the robot had travelled, we initially thought of using line detection, where we would have stage cues on the floor, and the robot would keep count of how many times it had driven past a line. However, we ultimately decided to keep track of this by using inverse kinematics and our **updatePose()** function, which kept track of the current theta/heading (in radians) of our robot. Therefore, since we wanted the robot to drive in a circle three times, the drove the robot until theta was greater than 6π. This method was a better option to implement than the line detection method. The line detection method would have required that we wired the line sensor and calibrated it; the calibration would have also just worked on one particular surface. However, by choosing to keep track of pose instead, it allowed us to perform the maneuver on any floor without worrying about calibration and also it was one less peripheral that needed wiring, as well as one less function/class implementation we needed to do.

Another core functionality of this scene was to simulate a robot stabbing another, which can be divided into two functionalities: to drive towards, and crash against, another

robot, and to detect the collision. Our first proposed solution for driving from one robot to another was to use inverse kinematics to move to a point, the point being where the other robot was. However, this solution was too basic, and unreliable, in that it required that the second robot (the one to be collided against) was always in the same position relative to the driving robot. Therefore, we decided to test the use of the OpenMV cameras and AprilTags to perform this functionality. We implemented functions that allowed a robot to detect and follow an AprilTag at a certain distance by using ther camera, details of which are found later in this report. After some initial tests, we determined that this solution would work very well, since, in contrast to the inverse kinematics solution, it allowed the robot to more easily adjust and adapt to changes in stage set up and positionings. Additionally, we determined that implementing AprilTag detection and following could be used to solve other challenges from other scenes.

The second functionality related to simulating the robot stabbing was the ability to detect collisions. Initially, we planned to implement our own collision detection functions, by using the IMU and acceleration values. However, in the end, the instructor provided some code for this that used the built-in tap-detection of the IMU. After reading the documentation on it and doing some initial tests, details of which are explained later in the report, we determined that the tap-detection was sufficient to fulfill our requirements and we opted for that solution.

Lastly, one other functionality we added to this scene, and throughout the scenes, was a robot life indicator, which indicated whether the robot was alive or dead. This was done through the use of an LED on each robot. Blue LEDs were used for the Montague family (Romi and Mercoolio) and Red LEDs were used for the Capulet family (Juliet and Tybalt). Although initially we attempted to make the LEDs fade in and out through PWM — to indicate dying or coming back to life (as for Juliet in the final scene) —, we ended up just implementing on/off functionality for the LEDs.

Scene 2: Balcony scene

Solutions and Justifications

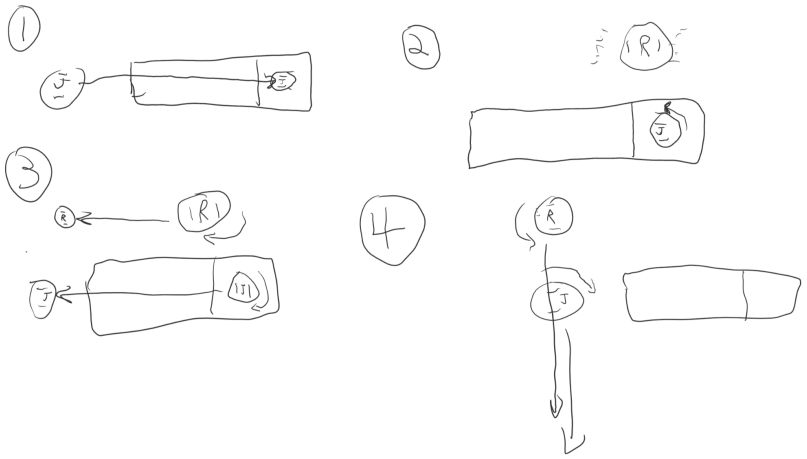


Figure 6: Balcony Scene Diagram

In this scene, Juliet goes up a ramp in order to signal Romi from above. Once she is on top of the “balcony,” she turns as Romi approaches to show her AprilTag to him, signaling her descent. Romi approaches and “wriggles” in excitement after seeing his lovely Juliet. As Juliet makes her descent, Romi follows along on the ground. Once she has reached the bottom, Romi will follow her AprilTag offstage.

To detect the ramp, the most intuitive option we had was to use the IMU with updatePitch and checkPitch functions that we wrote.

To drive up and down the ramp, we could have used solely our speed controller, but given that the robot is likely to fall off the ramp, we decided to implement wall following which enables the robot to stay away from the edges of the ramp. (refer to Wall Follow). Since the robot (Juliet) could have been equipped with a camera, we could have made it drive to an AprilTag that would have been attached to the top of the ramp. One problem with this option is that the camera could have lost the AprilTag as the robot drove up or down. In addition, by using the wall following in this scene, we are making better use of the range of sensors that are available to us.

For the section where Romi drives to face Juliet, we make Romi drive for a certain amount of time then turn and do nothing until he detects an AprilTag (refer to April Tags Detection and Following). When he detects the AprilTag, he drives to the beginning of the ramp. We considered using an IR beam signal instead of an AprilTag. But this option would have required that we use a PWM signal which we discovered is rare on the Romi. So, instead we decided to use AprilTag detection with the camera which we also used for closing/exiting the scene/stage.

For the part where Romi follows Juliet, Romi, waiting at the bottom of the ramp, sees Juliet and follows her off stage. To accomplish this, we use the AprilTag that is attached to Juliet. We could have made both of them drive off stage at the same speed, one Romi starting after Juliet. But, we would have had to use another system to detect when Juliet is off the ramp to know when to set Romi's speed. The advantage of using an AprilTag here is that the same cue is used to know when to start driving, and to direct Romi.

Scene 3: Final Scene

Solutions and Justifications

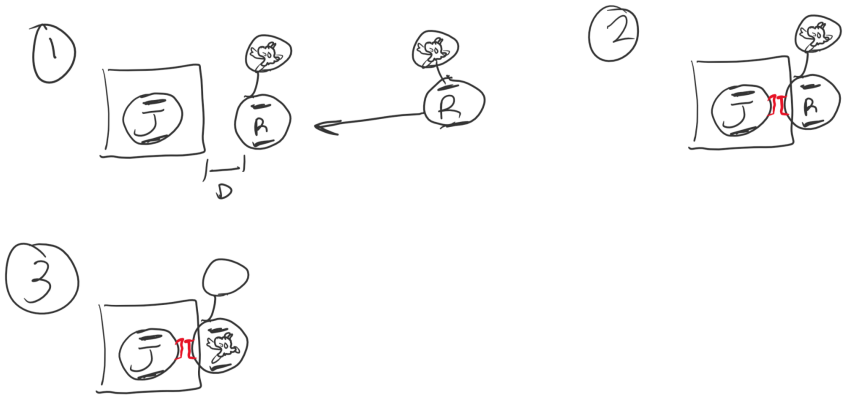


Figure 7: Final Scene Diagram

Romi drives over to a pedestal to see a dead Juliet. Seeing her dead is enough for him to commit suicide, so he drinks (or dumps) poison. Once he drinks the poison and is

dying, he gives Juliet one last kiss, which wakes her up to see his dead body, ending the scene and the tragic play.

The main functionalities in this scene are AprilTag following and detection, collision detection and our servo mechanism. Initially, we were going to use button pressing code on Romi and Juliet for the kissing portion of the scene, as we thought it would be simple and effective, but after consideration we decided that the difficulty of getting two buttons and/or limit switches to line up would be impractical. After being given the collision detection code, however, we decided to use that. There were a few options in terms of Romi dumping poison on himself, but since we had multiple servo motors and code for them from previous RBE classes, we decided to use one in this scene. We 3D printed a mount for the servo, along with a bucket to hold the “poison”, then defined two positions for the servo arm using servo.Write(); up and down. Romi starts the scene by driving straight towards Juliet. He stops once he detects her AprilTag using his OpenMV camera, meaning that he sees her lying dead on the pedestal. The AprilTag detection also triggers the servo mechanism, which will dump the poison onto him, killing him. After the poison is dumped, Romi will drive towards Juliet until they collide. This collision will result in Juliet’s LED light coming back on, signifying the tragic ending of the scene.

General Implementation

Overall Code Structure

In designing the structure for our final project code we knew that we wanted code for every scene that was easily organized and used the same code for core functionalities. To do this we created a main class which created an object for every robot/scene. In main we would select which robot and which scene we were going to run, and in so doing it would run the setup() and loop() functions for the respective class. To ensure that each of these

classes had access to the same core functionalities, they each created a common object, which held our code base.

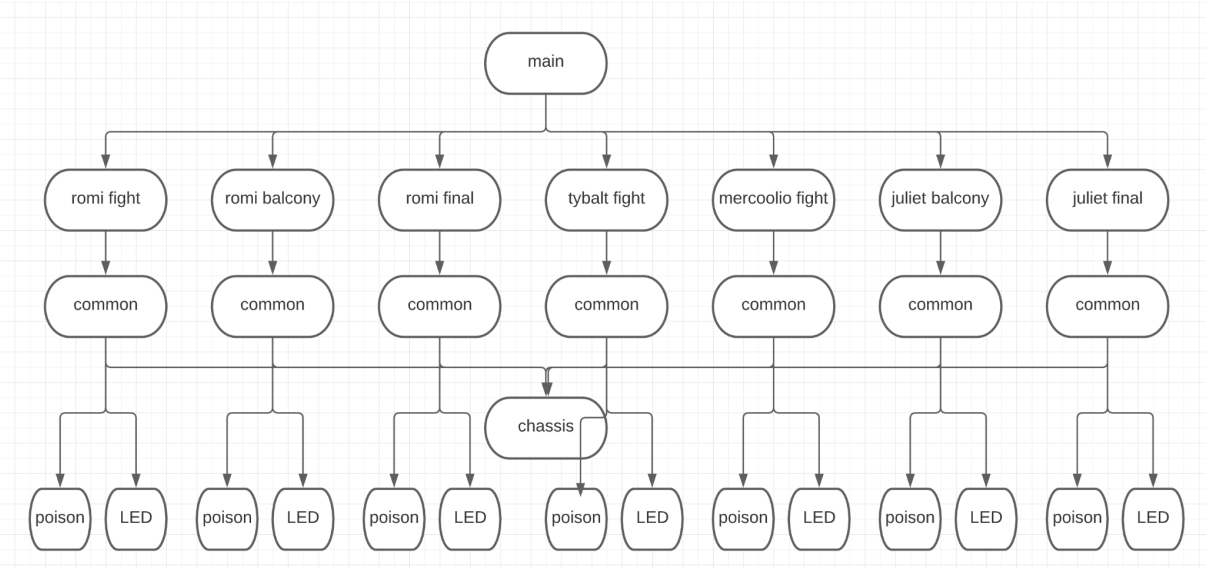


Figure 8: Code Flowchart (not all subclasses pictured)

This model served us very well, it allowed us to easily switch between programs. One issue we ran into was that by creating so many objects and instantiating so many variables, we were using up all of our available ram. To solve this we changed our Chassis class to be a singleton object, meaning that it would only be instantiated once. We recognized that making common a singleton object would likely have been even more effective, but we were worried that it would be too large a change and possibly create issues so we settled on just changing chassis, as it was our largest and most data intensive class. This helped considerably, lowering our RAM usage from nearly one hundred percent to thirty percent.

Every scene had a setup() and loop() function which ran important scene specific code as well as running the setup() and loop() functions in common. The setup and loop in common ran code that was important for all robots. For example common's setup enabled various timers and registers, and common's loop ran our speed controller and updated the pose of the robot. To keep this loop working properly it was essential that all of our code be non-blocking, otherwise we could not continue to update speeds or update pose, as well as a number of other functionalities.

Every scene was fairly similar in many respects. We have already discussed the setup and loop functions, however they also all employ a state machine with a number of similar states. After making a number of these classes we also created reference.h and reference.cpp files to facilitate easier class creation. Reference.h is pictured below.

src > C reference.h > ...

```
1   #pragma once
2   #include "commonCode.h"
3
4   class reference {
5       public:
6           void setup();
7           void loop();
8       private:
9           commonCode c;
10          typedef enum{
11          IDLE,
12          WAIT,
13          STOP,
14          TEST
15          } State;
16          State state;
17          State nextState;
18
19          unsigned long waitTime, timeLast, printTime;
20          bool enteringState = 1;
21  };
```

Figure 9: reference.h

There were two very helpful shared elements of the state machine, the enteringState boolean and the WAIT state. The entering state boolean was true when leaving a state and was immediately set to false after entering a new state. This allowed for code that was only meant to run once at the beginning of a state such as resetting pose or setting a start time.

```
/*//example of code for a case
case EXAMPLE:
    if(enteringState){
        enteringState = 0;
    }

    if(stopCondition){
        state = the next state;
        enteringState = 0;
    }
break;
*/
```

Figure 10: example case from reference.cpp

The WAIT state was also incredibly helpful, as it simply functioned as a non-blocking delay. It was very versatile in it's usage and made it very easy to keep our code non-blocking. The wait state required two variables to be set going into it, the wait time and the next state to go to. This allowed us to use this state to drive for a certain time, stop and wait for a length of time, and a number of other times it would have been nice to use delay() if it wasn't necessary to keep our code non-blocking.

```
case WAIT: //set waitTime and nextState before entering
    if(enteringState){
        enteringState = 0;
        timeLast = millis();
    }

    if(millis() - timeLast > waitTime){
        state = nextState;
        enteringState = 1;
        waitTime = 0;
        nextState = STOP;
    }
break;
```

Figure 11: wait state from reference.cpp

Another helpful feature of the wait state was that, as it is the only state that uses nextState and waitTime, it resets the values so that it will be easy to realize if we accidentally don't set the necessary values when going to the wait state.

Overall our code allowed us to easily implement our necessary features. This meant that most of our work was in getting our core functionality functioning. After we had

completed our core functionality, it took only three days to complete the project as the code base was easy to use.

PID Controller

We used a simple PID controller which took in error and outputted effort. It uses kp, ki, kd, and errorBound values that get set when the PID controller object gets declared. This allows for the same PID code to be easily used by multiple controllers. This PID controller was essential to our code, being used by the speed controller, the wall follower, and many other functionalities.

Speed Controller

The implementation of a speed controller was essential to our project, since it made the control of the chassis motors much easier. In general, we designed our robots to run on a two-stage control. The first stage used PID (not everything used all control terms) to determine the target velocity of left and right motors, and the second stage used the target velocity of left and right motors to determine the corresponding motor efforts. We decided to have a dedicated speed controller so we had better control of the wheel speeds. Throughout our project, even when we set target speeds directly, we never explicitly set the motor efforts, we always used the speed controller.

To implement the speed controller, we used a PI controller for each motor. The motor speed is read by an encoder and the resulting error fed back into the controller. Hence, we set up a PI(D) loop by setting up Timer 4 to normal mode with a TOP value of 249 (i.e. OCR1C = 249) and a prescaler of 1024; this corresponds to a timestep of 16 ms for each PID loop. Additionally, we enabled the overflow interrupt, which leads to another important element of the speed controller.

In order to perform the PI calculations for the speed controller, we would not only use target speeds for left and right motors, but we also needed to know the current state of the

speed; in our case, the current encoder counts for the left and right motors. Hence, at the beginning of each PI loop, when the PID timer overflow interrupt is triggered, the code captures a "snapshot" of the encoder counts for later processing.

In the updateSpeeds() function in the Chassis class, we calculate the speedLeft and speedRight — in change in encoder counts per change in interval, where the interval is the PID loop interval (i.e. 16 ms) — of the motors through the calculation **speedLeft = countsLeft - prevLeft** and then setting the new previous left speed to the current one for the next loop by doing **prevLeft = countsLeft**; the same is done for the right speed. Next, given the calculated current speedLeft and speedRight, the code computes the **errorLeft** and **errorRight** by finding the difference between **targetLeftSpeed** and **speedLeft**, and **targetSpeedRight** and **speedRight**. We then use the PIDController objects **leftMotorController** and **rightMotorController** to compute the **effortLeft** and **effortRight** and finally set the motor efforts to the latter.

Through a PI closed-loop controller and by keeping track of encoder counts (and the changes in them), we were able to implement the speed controller.

Updating Pose

To update the robot's pose we wrote a function that keeps track of the robot's position based on a frame of reference that has for origin the starting position of the robot. We essentially keep track of the robot pose (x, y, theta) by computing the speed of the center of the robot (speedCenter, on line 136 in figure 12) and the angular velocity (angVel, on line 135 in figure 12) at which the heading of the robot is changing. Then, the current heading, theta (refer to line 138 in figure 12), was always determined as the sum of the previous heading and the current angular velocity times time, but we did not need to multiply by the time difference because the speed of the wheels that we used were in centimeter per interval and this function was called every interval. To reduce the inaccuracy of discretization of the computations for the x and y positions, the corresponding computations were done

based on an angle that is the average of the previous and the current angle. To use some course terminology, this was a second-order approach. As line 140 in figure 12 shows, the position of the robot along the x-axis was computed as the sum of the the previous horizontal position and the distance driven along the x-axis, which is geometrically determined from the speed of the center of the robot (speedCenter) (no need to multiply by time here again because we are calling the updatePose every time interval). The position of the robot along the y-axis is computed analogically (refer to line 141 in figure 12).

```
130
131  void Chassis::updatePose(void) {
132
133          float ticks_per_cm = ticks_per_rotation / (PI * wheel_diam);
134          float dt = timestepMS / 1000;
135          float angVel = (speedRight - speedLeft) / (wheel_track * ticks_per_cm); //in rads per interval
136          float speedCenter = (speedRight + speedLeft) / (2.0 * ticks_per_cm); //in cm per interval
137          float oldTheta = theta;
138          theta = oldTheta + angVel;
139          float thetaStar = (oldTheta + theta) / 2.0;
140          x += speedCenter * cos(thetaStar); //in cm
141          y += speedCenter * sin(thetaStar); //in cm
142
143  }
```

Figure 12: Function to keep track the robot's position and orientation from chassis.cpp

Moving to point using pose

To move to a given point, we compute the speed of the left and right wheel based on the target pose and the robot's current pose (updated by a call to updatePose()).

For starters let's assume that the final heading of the robot does not matter. So, the goal is to get the center of the robot to the target position. The strategy being to follow a circular trajectory, we need to compute the distance between the current center of the robot (again, updated by updatePose) and target position, and the angular difference between the heading of the robot and the target position T. Figure 13 demonstrates how we found the angular difference θ.

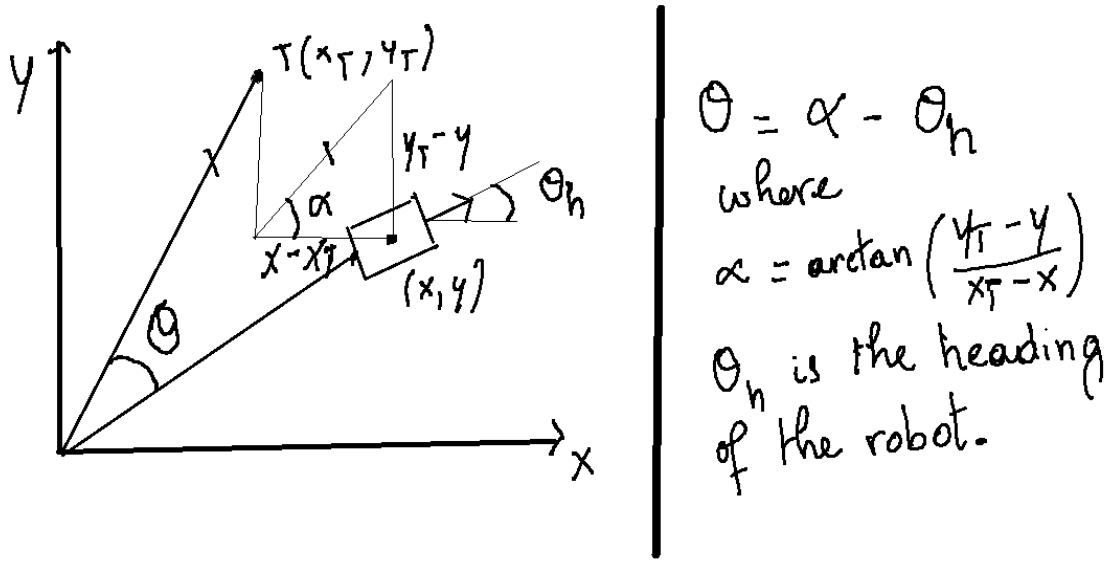


Figure 13: Geometric reasoning to find the angle θ between the robot's heading the target position

```
151 void Chassis::MoveToPoint(void) {
152     float errorDistance = sqrt(pow((x - x_target), 2) + pow((y - y_target), 2));
153     float internalTargetTheta = atan2(y_target - y, x_target - x);
154     float errorTheta =  internalTargetTheta - theta;
155     float errorHeading = th_target - theta;
156 
157     targetSpeedLeft = kpD * errorDistance - kpTheta * errorTheta + kpFinalHeading * errorHeading;
158     targetSpeedRight = kpD * errorDistance + kpTheta * errorTheta - kpFinalHeading * errorHeading;
159 
```

Figure 14 : Function that enables to move to a given point from chassis.cpp

In the code snapshot above (figure14), errorTheta corresponds to θ in our geometric reasoning (internalTheta is α and theta is θ_h). ErrorTheta (on line 154 in figure14) is used to make the robot go in a circle by adding proportional value to one of wheel speed and subtracting it from the other (line 157 and 158). errorDistance is the distance between the

current robot's position and its target position and is being used to set the initial speed of the wheels.

Moving away from our initial assumption, to reach a certain target heading, we compute errorHeading as the difference between the current heading of the robot and the target heading.errorHeading is multiplied by KpFinalHeading and added or subtracted to the wheel speed in order to make the robot turn and reach its final heading. This algorithm works partly because the error in the angle, errorTheta, is given a lot more weight than the error in the distance and the error in the heading (KpTheta >> KpFinalHeading and KpTheta >> KpD).

Drive in a Circle / Circle Maneuver

The circle maneuver in the first scene is one of the core functionalities of the project. To implement this, we created a function in the Chassis class called **driveCircle()**, which takes the radius in cm and the speed in ticks per interval.

```
void Chassis::driveCircle(float radius, float speed) {
    float speedRight = ((wheel_track * speed) / (2 * radius)) + speed;
    float speedLeft = (2 * speed) - speedRight;

    targetSpeedRight = speedRight;
    targetSpeedLeft = speedLeft;
}
```

Figure 15: Definition of **driveCircle()** function, called to perform a circle maneuver.

This function utilizes two equations for **speedRight** and **speedLeft** derived using the equations based on the instantaneous center of curvature. Fig. 15 shows the two derived equations as part of the function definition. It then sets the **targetSpeedLeft** and **targetSpeedRight** of the Speed Controller to the results of **speedLeft** and **speedRight**, respectively.

Tap Detection

We used the code provided by our instructor. The code in question makes use of the built-in tap detection mechanism of the LSM6 IMU, which requires the setting of register 59h and 58h.

We had to edit the register settings to make tapping work more consistently with the different robots, and speed configurations. Through a process of trial and error, we determined that register 59h, which is used to set the tap threshold, was set to a threshold of 4 (the unit is unknown). It also turned out that a full-scale range value of 2 and an output data rate of 104 Hz combined with the value chosen for register 59h provided enough sensitivity for the various scenes.

Servo Control

Servo control was not the largest obstacle to overcome in this project, as it is only used briefly. We decided to use a servo to control the mechanism that dumps poison onto Romi for the final death scene.

The servo we used has two pinouts, plus voltage and ground. One is for command signal, which we set to pin 5, and the other is position feedback, set to pin 11. The code for this section of the project was rather simple, as the servo only needed to achieve two different positions; holding poison, and poison spilt. Using servo.Write() we can tell the servo to move to whatever position we desire. Once we found which Write() amount corresponds to which position, we defined them in their own class. Once they are defined, we can call these positions in any function by using the servo class (named poison).

We also created a mechanism to attach to the servo and hold the poison. Two separate parts were created in CAD; a mount for the servo, and a bucket to hold the poison and attach to the servo arm.

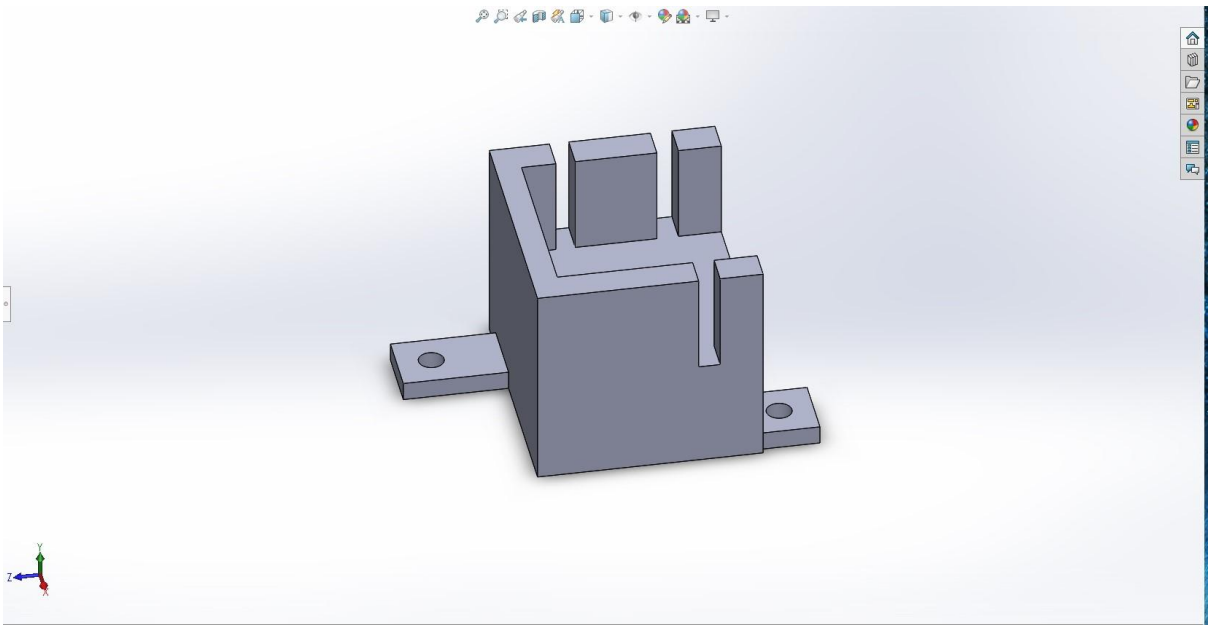


Figure 16: The servo mount in CAD

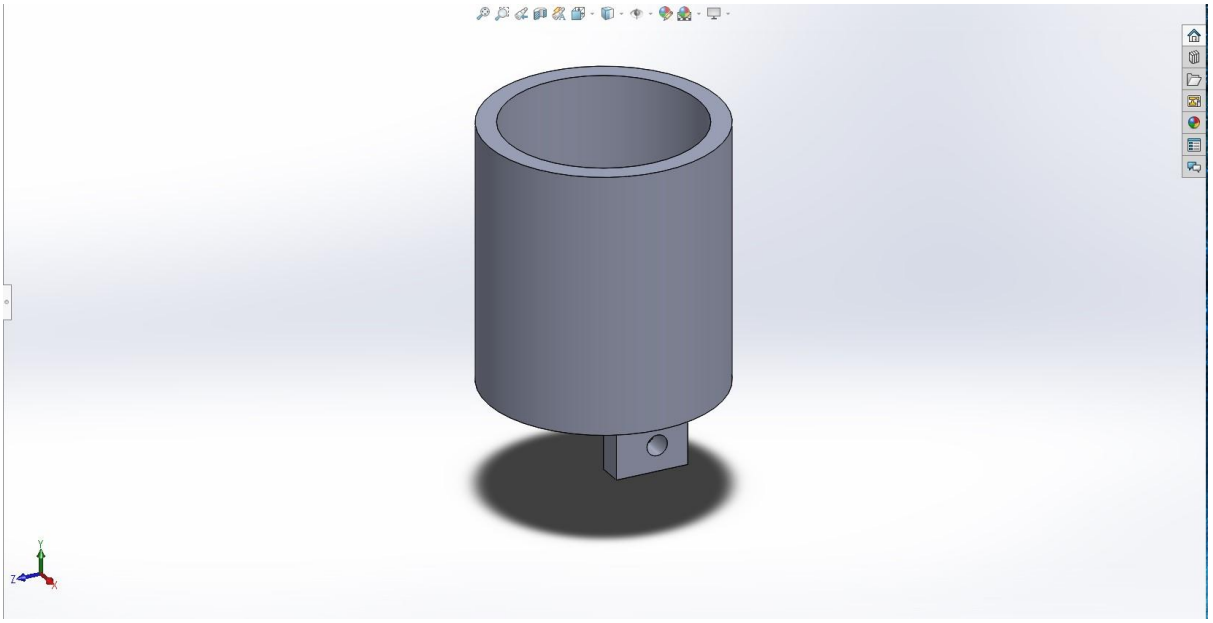


Figure 17: The poison bucket in CAD

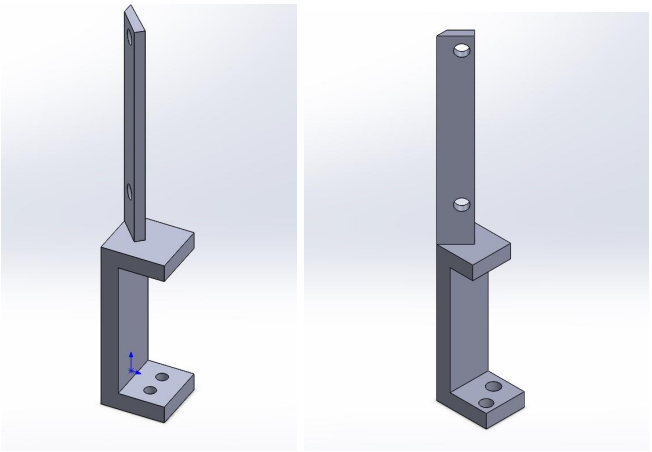
Wall Follow

Originally, we were going to use wall following in most of the scenes for the final project, but as we kept experimenting, we found other ways to drive straight to our target. In the end, we decided to use the wall following on the balcony scene alone, when Juliet drives

up the ramp then back down. We needed a way for Juliet to stay on the ramp while driving up and down, and we decided that wall following was the best solution to implement.

For our wall following we used a rolling average of distances from the wall to compute error distance and to filter out noise from previous readings. The rolling average takes the average distance from the last 5 readings of the sensors, meaning it updates constantly to give us the most accurate reading possible. This average is used to give us our distance error, which we use to correct wheel speeds by setting target speed left and right based on PID control.

Wall follow can be done using different types of sensors, but after experimenting with the IR sensors so much we came to the conclusion that they were our best option. We also found that using two IR sensors instead of one would allow us to wall follow both up and down the ramp, so two mounts were printed to point in positive and negative 45 degrees off the x-axis on the Romi;



Figures 18: The two IR Sensor mounts in CAD

We found that angling the mounts gave us better readings and a more accurate wall following as it kept the signals from the two IR sensors from interfering with each other. After the sensors were mounted, we used PID control to create and sustain a target distance from the walls. Some experimenting was necessary to find the correct constants for the PID

control, but once we found the best value, our wall following worked perfectly for the balcony scene.

AprilTags Detection and Following

For AprilTag detection and following, we used the OpenMV Cam H7 and the base code provided to us by the instructor. We uploaded the code for AprilTag processing onto the camera and we were then able to use I2C to communicate and get the camera data onto the 32U4 for higher level processing. We also used the code C++ code to get the different data about the AprilTag, such as the tag id, width, height, cx and cy (x and y pixel coordinates), etc. This was critical data to be able to implement the follow and detection of AprilTags.

Firstly, we derived an equation that we used to determine the distance from an AprilTag by using the returned tag width from the I2C communication. To derive this equation, we used concepts of angular field of view and pixels, as well as calibration values.

Consider the first case below, where we knew the tag (half) width calibration $l_{calibration} = 3.05\ cm,$ tag distance calibration $d_{calibration} = 20\ cm,$ tag width calibration $n_{calibration} = 45\ "pixels"$. From this, we were able to calculate ϕ and the ratio between ϕ and $n_{calibration}$. This ratio is also the ratio between the camera's angular field of view and pixels in y-direction, which is constant.

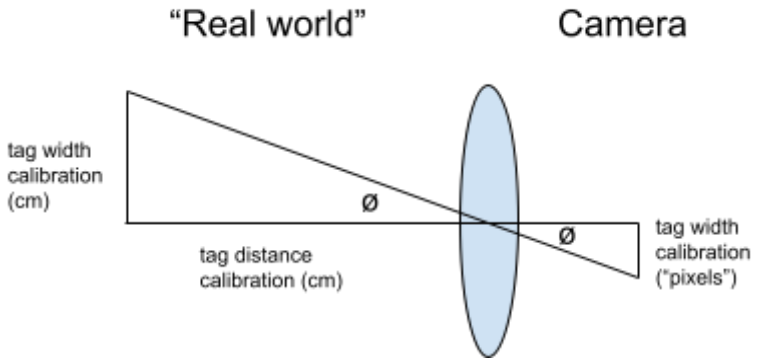


Figure 19: Diagram of initial set up of camera and tag, with calibration constants and angular field of view.

Now, consider a new set up below, with a new, unknown distance from the AprilTag and with a new tag width reading returned. From the established relationship between the camera's angular field of view and pixels, we determine ϕ_2 and from that we could calculate tag distance using trigonometry.

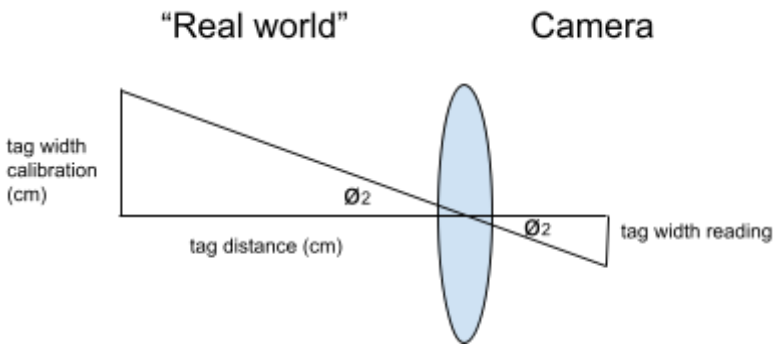


Figure 20: Diagram of final set up of camera and tag, used to determine the tag distance.

After doing the calculations, we determined an equation for distance based on the known values, as shown in the figure below .

```
float distance = (actualWidth / 2) / (tan(tagW * (atan2((actualWidth / 2), calDistance) / calWidth)));
```

Figure 21: Code snippet of camera distance equation, with calibration constants and tag width reading.

All variables except **tagW** are calibration constants, and **tagW** is the reading from the camera. We used this final equation in the **getDistanceCam()** function in AprilTags.cpp, which returns the distance from the AprilTag.

Apart from distance, for our FollowAprilTag() function, we needed to know the robot's (i.e. camera mounted on robot) translation in the x direction from the center of the AprilTag. We will expand on why we needed to know this shortly. For this, we created another function in the AprilTags.cpp called **getDeltaXCam()**.The function calculates and returns the offset in x-direction of the current x-direction pixel value (i.e. tagCX) from the center pixel, which has a value of 80.

Throughout our code, we make use of three functions in the Chassis class that are related to following, detecting and getting a distance from an AprilTag. The robots use the **FollowAprilTag(float targetDistance)** function to follow an AprilTag (or a robot with a mounter AprilTag) at a set target distance from it, which is passed as a parameter. It does this by calculating the error distance by reading the distance from the AprilTag — through the method described previously — and subtracting the target distance from it. Additionally, the function calculates the error in x-translation from the center of the AprilTag. Given these two errors, **errorDistance** and **errorXTranslation**, the function sets **targetSpeedLeft** and **targetSpeedRight** through proportional control.

```
targetSpeedLeft = errorDistance * kp_distance - errorXTranslation * kp_alignment;
targetSpeedRight = errorDistance * kp_distance + errorXTranslation * kp_alignment;
```

Figure 22: Code snippet of target speed left and right equations based on error distance and error in x-translation of camera from tag.

One thing to note is that **errorDistance** and its corresponding proportional constant correct the distance from the AprilTag and **errorXTranslation** and its corresponding proportional constant correct the alignment with the AprilTag (i.e. makes the robot turn towards the AprilTag). There is also an overloaded **FollowAprilTag(float targetDistance, float**

maxSpeed) function that does the same as the former, but it also sets a maximum speed for following the tag. We implemented this primarily to be able to follow a tag from longer distances but without the robot slipping while trying to correct to the target distance, since a large error distance without a cap in speed caused the robot to accelerate really fast. Both of these functions have a timeout of 500 ms, which causes the robot to stop when it doesn't detect an AprilTag for more than that time interval.

Another function that we implemented was the **DetectAprilTag()** function, which returns the tag ID of the detected tag; when no tag is detected, it returns -1. This allowed us to set a constant speed to the motors until the camera detected a tag (i.e. until tag was in range of the camera); from there onwards, we would use the **FollowAprilTag()** function. Finally, the last camera/AprilTag related function we implemented in the **Chassis** class was the **getDistanceCamera()**, which it essentially does what **getDistanceCam()** function in AprilTags.cpp does, but we implemented a similar function in Chassis so that it was accessible to the other classes by just including Chassis. Also, this new function also takes into account the **CAMERA_OFFSET** from the robot chassis and the **TAG_OFFSET** from the robot chassis.

Estimating Pitch and On Ramp Hysteresis

For the balcony scene, Juliet must drive up a ramp and stop once she detects there is no pitch and she is on level ground. To do this, we created a complementary filter as we had done in a previous lab to estimate pitch. This filter used the built in IMU on the Romi, specifically the accelerometer, to give an estimate on what pitch the robot was driving on. The main factor for this estimation is the acceleration of gravity, as we can use the direction of the gravitational acceleration vector to determine the angle of the robot relative to the ground.

Our complementary filter uses two main functions; observed pitch and predicted gyro. These two functions are dependent on each other, along with data rate, sensitivity, offset, bias, and individual IMU readings;

```
float predictGyro = estimatedPitchAng + (dataRateSec * (imu.g.y - Bias) * senseRad);
float obsPitch = atan2((double)(imu.a.x - accXoffset),(double)(imu.a.z));
estimatedPitchAng = kappa * predictGyro + (1 - kappa) * obsPitch;
Bias = Bias + E*(predictGyro - obsPitch);
```

Figure 23: Our complimentary filter in the Final code

As you can see from the figure above, we used multiple equations given to us in Lab 4 to calculate the estimated pitch angle. Observed pitch is the tangent of the x and y axis accelerometer readings, with an offset in the x-axis. Predict gyro is an angular value calculated by adding the previous estimated pitch angle to the sensitivity of the gyroscope multiplied by the time step and the y-axis gyro reading minus the ever-changing bias. This all leads to the updated pitch angle, calculated by multiplying the K value to the predicted gyro value and adding it to the decrement of the K value multiplied by the observed pitch value.

This estimation by itself is not enough to complete the ramp maneuver, however. This will tell you the pitch, but there is a significant amount of noise that needs to be filtered out in some way. To address this problem, we decided to add a hysteresis band to the pitch detection code. A hysteresis band entails that the robot will be able to determine exactly when it is on or off the ramp, and stop accordingly.

Remote Detection

To detect the remote we utilized the IRdecoder class from the code library. We also created a remote constants file to compare the keycode to. To read the remote value every loop we would store the keyCode we were receiving in a remote code variable. If we were waiting for a specific remote button to be pressed, we would compare the value of our

remote code variable to the corresponding value of a remote button, stored in the remote constants file.

Performance

Fight Scene

Many portions of this scene performed very well. We were especially satisfied with our speed controller and pose based circle maneuver. We were initially concerned that not referencing the outside world would lead to wildly inconsistent circles, collisions of the robots, and variable endpoints. In the end, however, the robots drove in very consistent circles, never collided, and were very precise in where they ended their circles. The combination of the speed controller and updating pose to know the stop location satisfied our expectations in this regard. Our least consistent element of the project came from this scene, in the form of tap detection. We found that we were either having issues with the robot detecting its own change in acceleration as a tap, or with not detecting collisions depending on which way we tuned the tap detection. In the end we solved this by tuning the tap detection towards the more sensitive side and only checking if we had detected a tap when the robot was within a certain distance from the robot, which we measured using the camera. This brings us to our implementation of the camera, which we also felt went quite well. The camera was able to consistently detect and follow AprilTags as well as return an accurate distance from the AprilTag. We certainly had some challenges implementing the camera as was discussed previously, but the camera tracking as it was implemented for the performance worked accurately and consistently. To improve this scene we would improve our tap detection. One of the main issues here was that as the camera turned to track it's target, the IMU detected that as a tap leading to a false positive. To solve this, we believe we could have the robot turn until it was lined up with it's target and then charge. This charge would involve ramping up the speeds, and only turning on tap detection part of the way through. This way we would be colliding with more force and could tune the tap detection based on that and the ramping up of speeds, which would prevent false positives. We think

that this could make the tap detection more consistent, but we decided not to implement it for the final performance as we are content with the current consistency of our robots.

Balcony Scene

Our robots performed remarkably well during the balcony scene. Juliet's wall following up the ramp was very consistent after tuning and we never saw Juliet get misaligned. Juliet was also able to very consistently measure her pitch and determine when she was and was not on the ramp. Juliet's other motions in this scene were either turning using pose or driving to a specific point also using pose. These motions were also very consistent, leading to Juliet's performance being extremely repeatable. In this scene Romi utilized the camera and updating pose to do most of his motions. One interesting part of this scene is that when Romi detects Juliet's AprilTag he gets excited and jitters back and forth very fast. This is interesting as it leads to Romi skidding slightly resulting in his pose not being entirely accurate. This makes it so that when Romi drives to the bottom of the ramp his position is slightly varied, and makes it so that Romi's performance in this scene is dependent on his following of Juliet's AprilTag. We are also very satisfied with Romi's use of the camera in this scene, as it was very consistent and had very few issues. The only notable issue with this scene is that Romi drives on stage by driving for a certain distance. He drives for a very consistent distance and it's not important that he starts in exactly the right place as he relies on the camera for most of this motion, leading to this not being much of an issue. However, if Romi is started egregiously in the wrong position, he can be so misaligned that he won't detect Juliet at the top of the ramp. One thing we could implement to improve this scene is either the reflectance sensor so that Romi drives forward until he reaches a stage queue on the ground, or have Romi drive forwards towards an AprilTag that is part of the stage but disguised as a wall of vines or something similar. We decided not to implement this as it was difficult to set up Romi in the wrong location and we also liked that

Romi's location was slightly variable as it meant that he had to rely more on his camera for positioning. Overall, our Balcony scene was very consistent and we are very pleased with it.

Final Scene

This scene implemented the tap detection, speed controller, and tap detection from previous scenes. These all performed quite well. If we wanted to improve our performance in this scene we would most likely implement the fixes for tap detection we discussed in the fight scene. Overall, we thought it was still performing quite well even if only at about eighty percent consistency. The only new functionality in this scene was using the servo motor to dump poison onto Romi. The only issue we ran into here was that we did not originally secure the wires of the servo motor. This meant that occasionally the servo would unplug during normal operation, which was not ideal. After fixing the wiring issue the servo performed flawlessly, leading to our overall performance in this scene being very consistent.

Overall Performance

We feel that our robots performed very consistently across our three scenes. With our robots we have completed our original goal of truly bringing Verona into the RBE lab. And as we complete this project, a glooming peace this morning brings. The sun, for exhaustion, will not show it's head. We go hence, to have more talk of these learned things. And to exult in successful completion of hard work. For never was a story of more glee, than that of Juliet and Romi.