2019 Spring ME492 Final Report

Team I
Seungeon Lee, 20140416
Minseok Jang, 20160558
Thanh Nguyen, 20150846
Chaeree Park, 20160275
Taehwan Kim, 20150861

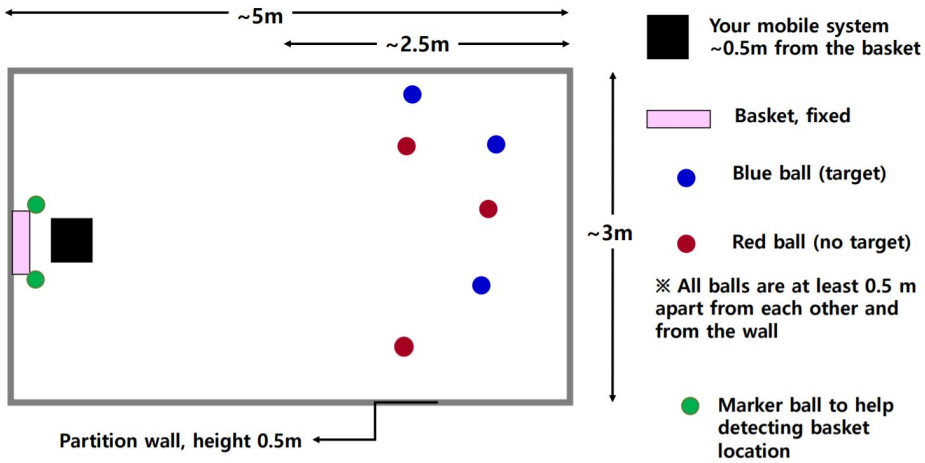
2019.06.16

1. Preface

1. Introduction

The primary concern before developing the ROS integration algorithm is finding the appropriate link to the given hardware. Based on its needs, we are to develop a well-planned algorithm to indicate the target; publish precise, accurate commands; and optimally finish the given task.

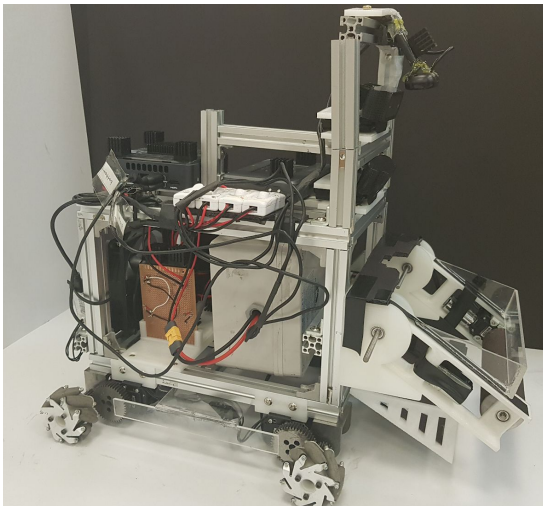
1.1 Task



<Figure 1. Final demo task>

The overall task of Capstone Design I is to pick up 3 blue balls in an arena with 3 blue and red balls each. In completing the demo, we aim to optimize our system's path in picking up the blue balls and avoiding the reds. It sounds like a simple task to fulfill; however, building a rigorous algorithm requires a lot of detailed deliberations, considering the numerous cases our vehicle may face during the task. This report will go through our group's initial logic plans and how we have debugged problems we have faced during the development of our final algorithm for the demo.

1.2 Hardware design

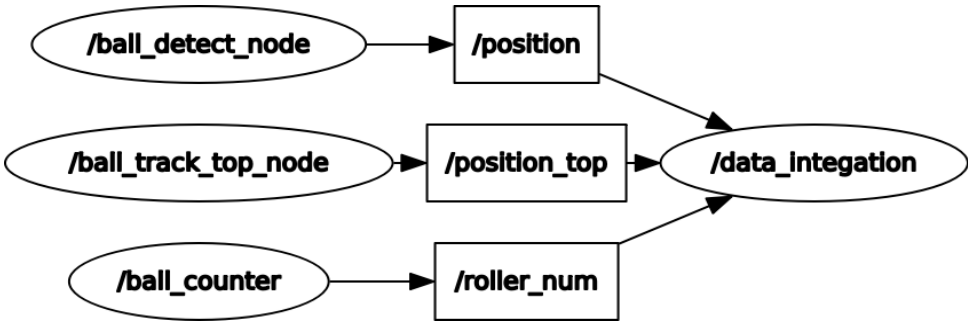


<Figure 2. Hardware>

Figure 2 is a picture of our hardware. We used roller as pick-up module. One big advantage of roller is that picking up algorithm finished just by turning on the roller. Also, the releasing algorithm is that turning on the roller by opposite direction. With this outstanding hardware and following codes, we finished the task for shortest time at final demo. Now we will explain about our codes.

2. Code explanation

2.1.1 General logic (rqt graph)



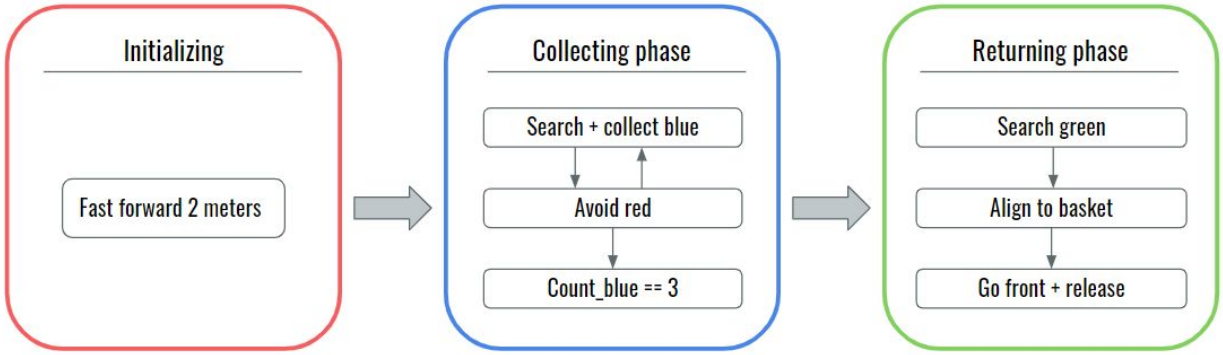
<Figure 6. rqt_graph>

Our system consists of 4 main nodes: /ball_detect_node, /ball_track_top_node, /ball_counter, and /data_integration. As seen in the rqt_graph, /ball_detect_node, /ball_track_top_node, /ball_counter nodes publish /position, /position_top, and /roller_num topics that are used in the /data_integration node.

To give a brief explanation of the functions for each node, /ball_detect_node processes the data from the bottom webcam, /ball_track_top_node processes the data from the top webcam, /ball_counter node processes the data from the ball counter webcam, and the /data_integration subscribes to all the topics from the three other nodes, and compiles the final processes in order to operate the vehicle.

In this report, we will be delving into the /data_integratio node since the other nodes are mainly OpenCV algorithms.

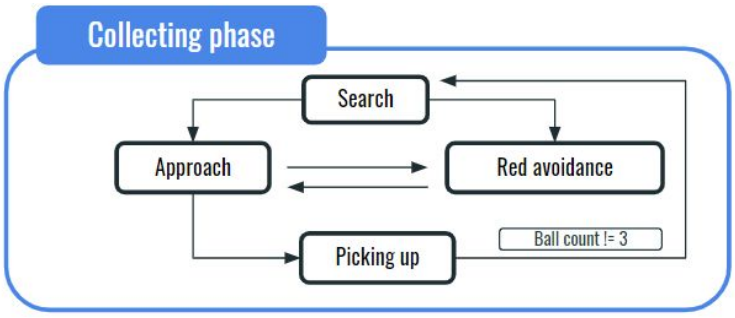
2.1.2 General logic (Overall system)



<Figure 7. Overall system>

Our overall system is consist of 3 phases. First in Initializing phase, our robot go fast forward 2 meters. Then, the phase is changed to Collecting phase. After collecting 3 blue balls (or timeout), the phase is changed to Returning phase and go to basket and finally release all the blue balls.

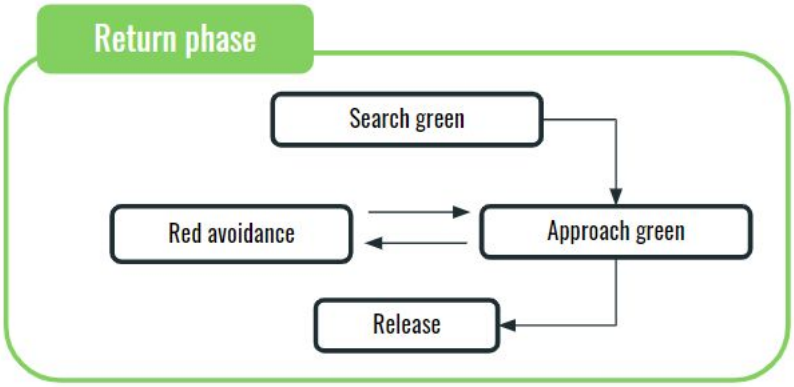
2.1.3 General logic (Collecting phase)



<Figure 8. Collecting phase>

In Collecting phase, there are four states. Search state is that find and align to target blue ball. Then, the robot tries to approach target. But if there are red balls on the path, Red avoidance state activated to avoid them. Finally if the robot reach to target closely, Picking up state activated and collect the blue ball. And this algorithm is repeated until the Ball count became 3.

2.1.4 General logic (Return system)



<Figure 9. Return phase>

In return phase, there are four states. In Search green state our robot find and align to green balls. Then, the robot tries to approach green balls. In the same manner of Collecting phase, Red avoidance state will be activated if there are red balls on the path. And finally if the robot reach to target closely, and aligned perfectly, all the balls would be released.

2.2 Code details

2.2.1. mainv2.cpp(/data_integration)

1 ~ 128: Include header files that are necessary for this code, and define values.

This is the summary of including header files and defining values. In line 19 and 20, “core_msgs/ball_position.h” and “core_msgs/roller_num.h” are included for subscribing webcam data. In line 22, “sensor_msgs/LaserScan.h” is included for subscribing lidar.

In line 28, POLICY is the variable that selects target. It is initially set as LEFTMOST. The target can be changed based on the value of POLICY. If POLICY is 1 or 2, for instance, target is set to centermost blue ball or closest blue ball.

DISTANCE_TICKS, ROTATE_CONST_SLOW, TRANSLATE_CONST_SLOW, DURATION, COLLECT_THRESH_FRONT, DISTANCE_TICKS_CL are decided based on movement of our Robot.

In line 49 and 50, the status of machine are initially set as INIT which phase is approaching 2m at beginning. Then, leftmost blue ball is set as a target ball in line 51. In line 58 to 121, variables that are used in this code are defined.

131 ~ 136:

Main function starts from line line 131. In line 133, the function signal prevents program from terminating when wrong indices are referred. And “data_integration” node is initiated in line 135.

138 ~ 206: Declare offset argument, and designate values in flag.

Position offsets are defined to minimize the error from initial position, and timeout is defined to ensure that robot is able to return to basket at the assigned time.

ROS nodes are capable of accepting arguments, like any other c/c++ executables. C/C++ supports several argument parsing functions such as the following.

int getopt(int argc, char** argv, const char* optstring)

In line 162, getopt(argc, argv, "x:z:X:Z:mT:"), the meaning of this line is that -x, -z, -X, -Z, and -T will be followed by one additional argument and *-m* will be solely used. Therefore, issuing the command

rosrun data_integration data_integrate_node -x -0.013 -z 0.18 -T 60 -m

sets x_offset = -0.013, z_offset = 0.18, timeout = 60, and use_myrio = false.

The variable flag is to check what arguments have been assigned. If ‘x’ and ‘T’ have been assigned in command line, for instance, binary number ‘1 0 0 0’ and ‘1 0 1 0 0 0’ are compared with flag. Therefore, flag becomes ‘1 0 1 0 0 0’, and it is possible to know which arguments have been assigned from flag.

209 ~ 213: Subscribing topics from webcams.

Subscribe to position topics “/position”, “/position_top”, and “/roller_num”. Each subscriber represents main camera, top camera, and ball counter camera.

216 ~ 235: Reset socket values in c_socket, and communicate with Myrio.

If Myrio is defined, make c_socket by socket function and designate IPADDR and PORT. Then, try to connect and if it is failed to connect, print message, close the socket, and return -1.

236~257: Timeout setting

Before getting into the main routine, we print out that our system has started and is initialized. We set the current_ticks = timer_ticks to mark the start of the program. Based on a pre-defined time frame as explained in Appendix A.1.1, we have added a timeout setting as a backup plan for blue ball counter error. This forces our machine_status to automatically change to the return phase when a certain amount of time (input) has passed since program start and system initialization.

258~603: Overall switch function explanation

These codes are used to switch our machine_status based on predefined conditions and current machine data output from the webcam. In case of the machine_status, we have 8 main states in order to operate our vehicle all throughout the given task.

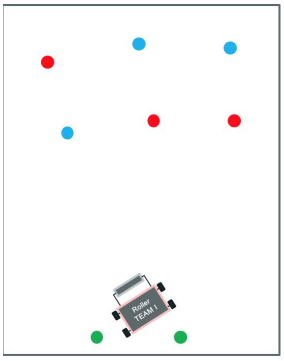
This code always starts with an initializing phase to move our vehicle closer to the ball zone within the arena. From there, we search for the blue balls and start to pick up three blue balls in the arena and avoid red balls when we need to. When we collect all three balls, we return to the basket based on the green ball positions and release our balls. The details of the algorithm will be discussed below.

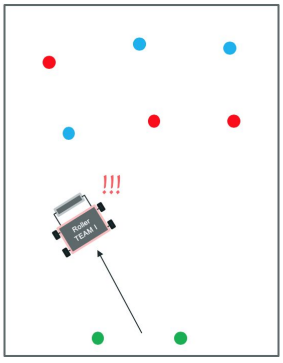
Throughout this state determining algorithm, we have added a predefined print function MSGE in order to feed the current status for debugging purposes. The MSGE function was created so that repetitive message printing based on our duration time of 1/40 seconds will not occur, but rather at a rate of ¼ seconds.

259 ~ 268: INIT state

In the initializing state, the vehicle is to move forward for 2 meters, then change to SEARCH state. Notice that our starting position is towards the left because our general search logic is to pick up the leftmost ball and rotate clockwise.

We added this state because we faced some problems when we started the initializing state from the starting position. The vehicle would fall into an infinite loop during the search phase while deciding the eftmost_blue ball. There were several ways to debug this issue, but the most straightforward solution to this problem was to move our vehicle closer to the balls for steady recognition from OpenCV. Based on the conditions of the arena and ball locations, this solution was readily viable and therefore we decided to add the initializing state.





269 ~ 304: SEARCH state

In the search phases, we are to find the blue ball target we are going to pick up. The general philosophy behind this state is to first find the "closest ball" using the bottom camera, and if a ball is present, the vehicle aligns itself and the variable 'machine status' switches to APPROACH. If not, the top camera is used to search for the left most ball in screen and approach the ball so that the ball can be seen with the bottom camera. If no ball is seen in both cameras, the vehicle is to rotate clockwise until a ball is in sight.

The logic starts with an if statement that checks whether there is a blue ball on the bottom camera

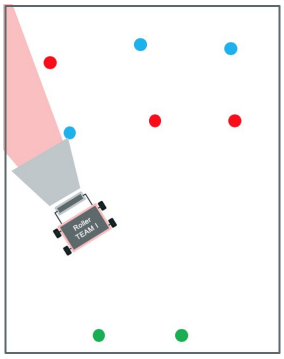
If no blue ball on bottom, we move to the next if statement checking the presence of blue ball with the top camera.

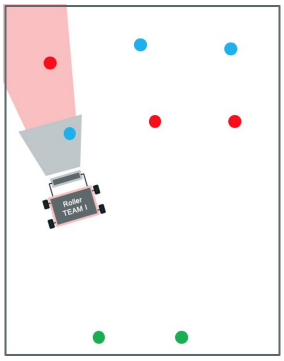
If there is a blue ball in sight, we take the ball position from the top webcam to align the vehicle towards the ball within an adjusted uncertainty range. After alignment, the vehicle moves forward until the ball is seen with the bottom webcam.

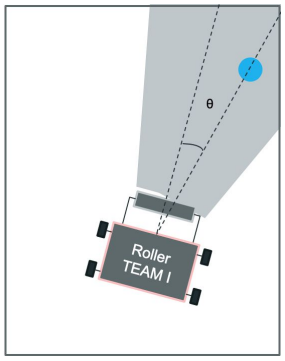
If there is no blue ball in sight, which means that no balls are within both camera ranges, we need to rotate our orientation. Here we rotate the vehicle clockwise until a blue ball comes into sight. Then based on where the ball is seen (top/bottom camera) the if statement will repeat

If the blue ball is seen from the bottom camera, we first align the vehicle towards the blue ball. Here we use a more precise uncertainty value determined based on maximum distance from the vehicle to the ball and the roller width. With these values, we will always be able to collect the ball.

During the search phase, it is definitely possible to have a red ball come into sight. Therefore, we have added a code to switch the machine_status to RED AVOIDANCE if the red ball is closer to the vehicle than the target blue ball.





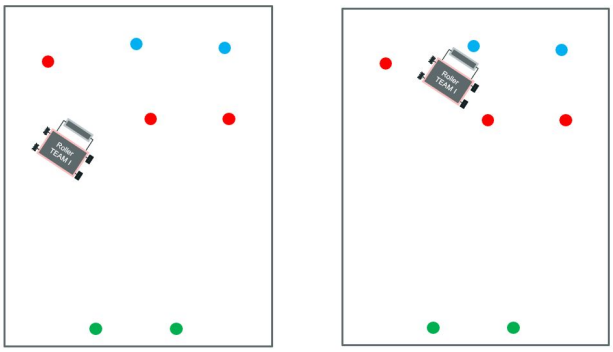


305 ~ 337: APPROACH state

In the approach state, we must realize that the blue ball and the vehicle is already aligned before the phase switch; therefore, the vehicle is ordered to move forwards. However we have considered several conditions that can occur during this phase.

First, if there are more than one blue ball in the bottom camera sight, we have organized an if statement to always target the closest blue ball.

Second, if the vehicle and the ball have been misaligned due to inaccurate movement or due to a target change as discussed above, we must switch to SEARCH phase to restore correct alignment to the new target.



Lastly, as we continue with correct forward movement, if the vehicle to ball distance becomes less than 20 cm, we move to the COLLECT phase.

And of course, during the approach phase, RED AVOIDANCE algorithm is implemented in such a way that it pops up when a red ball is present in sight range. We also added a code for error detection in case a ball is "accidentally" recognized as a red ball during movement due to vibration that makes webcam data slightly unstable in certain cases.

338 ~ 362: RED_AVOIDANCE state

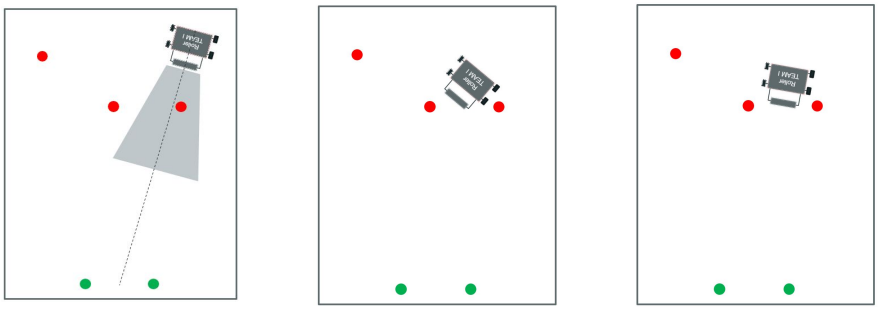
In RED_AVOIDANCE state, there are two parts: rotating and moving forward.

Considering the case when the red ball is not in front of the vehicle and red_phase2 is false, this means that we have already turned away from the red ball, but red_phase2 needs to be turned on for the vehicle to move forward. Then we simply need to change the red_phase 2 boolean to true. At this point we save current_ticks from the running timer_ticks to spot the time at which the vehicle starts to move forward.

Next, if only red_phase2 is false, we want our vehicle to turn left to avoid the red ball. If the first if statement containing !red_in_range is unsatisfied, this means that red ball is in sight. The vehicle would rotate until the red ball becomes out of range, which then means the if statement above will be activated.

Lastly, when the red_phase2 becomes true, this means that we have avoided the red ball and the vehicle is ready to move forward. Therefore we move the vehicle forward. In order to stop moving forward we must change the red_phase2 boolean back to false. The condition at which we stop moving is when the timer_ticks - current_ticks becomes greater than the distance_ticks. Distance_ticks is predefined as a variable of 65/40 seconds obtained based on both analytical and experimental tests.

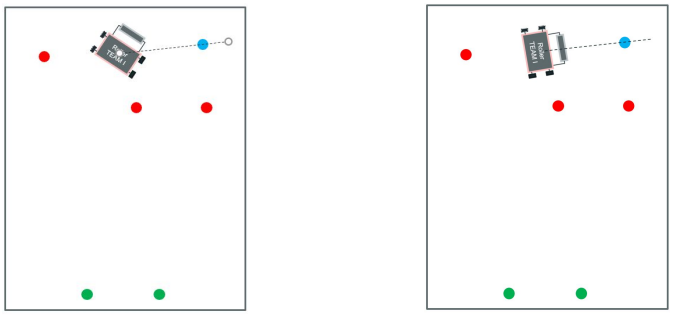
Since the red_phase2 is back to false and we have avoided the red ball, we have to return back to search (during pickup mode) or search green (during return mode).



363 ~ 406: COLLECT state

In the COLLECT state, robot is aligned to target blue ball and roller is turned on. This state is until alignment and collection will be conducted in collect2 state. The transition to collect2 state occurs if the target ball is presented in the collecting range which is between -0.013m and 0.013m of the roller. If it's not, robot slowly turns and goes front to align the target ball with the roller until it matches, then changes to collect2 state. If red ball is in range, the state changes to RED_AVOIDANCE state at any moment.

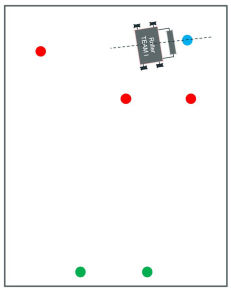
When we implemented collect state, there was an issue that robot collides with close ball when it heads to leftmost blue ball which is located further. To solve this issue, we used the closest_ball function which is explained in approach state. Thus, if there's a blue ball on the way to collect leftmost ball, robot will collect the closest ball first. Moreover, we encountered another situation that blue ball is out of sight because of imprecise alignment to target ball or the target ball that is hidden by roller. In this situation, the state of robot is converted to collect2 state to pick up the hidden ball.

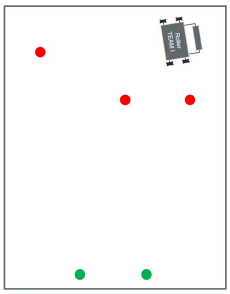


407 ~ 424: COLLECT2 state

In the collect2 state, we made our robot to go toward the target ball with roller on. And because we need some time to collect the ball completely, we should wait with roller on. So, the state would change to SEARCH after the difference of timer_ticks and current_tics become 75 not directly after stop. The number 75 is defined as DISTANCE_TICKS_CL and we choose this number with many experiments.

Our initial plans dealt with using a blue ball counter during the COLLECT2 state. Assuming correct alignment and proper vehicle movement, the blue ball is collected as it becomes out of sight behind the roller (our bottom webcam is set to see until the boundary of the roller). However, we had some overcount problems due to overshoots and boundary considerations. Even with considerable changes, we thought that to keep a rigorous and safe ball count system, we must have a separate camera to do the ball counts.





424 ~ 463: LIDAR_RETURN state

If the number of collected ball is equal to 3, then change to the LIDAR_RETURN. If the lidar is not utilized, then immediately change to the SEARCH_GREEN state. If red ball presents, changing to RED_AVOIDANCE state at any moment.

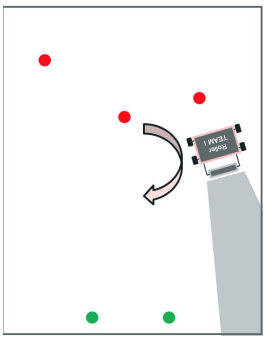
If the lidar is utilized, based on the absolute coordinates estimated from measurement of lidar, move the robot to the center line aligning with the basket and facing the basket. If the absolute angle of the robot is smaller than 170 degree or larger than 190 degree, adjust robot's direction back to near 180 degree. And if the robot is away from the center line 2 cm, adjusting the lateral position of the robot to the center line. Then change to the SEARCH_GREEN state.

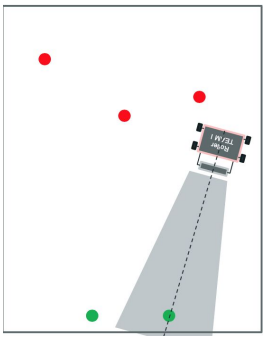
464 ~ 556: SEARCH_GREEN state (속도조절)

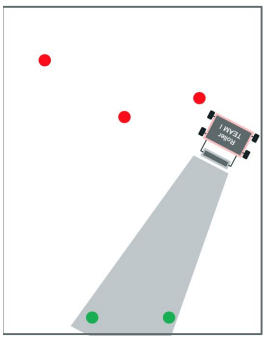
Case 0: if no green ball is detected by the cameras, robot keeps turning right until detecting.

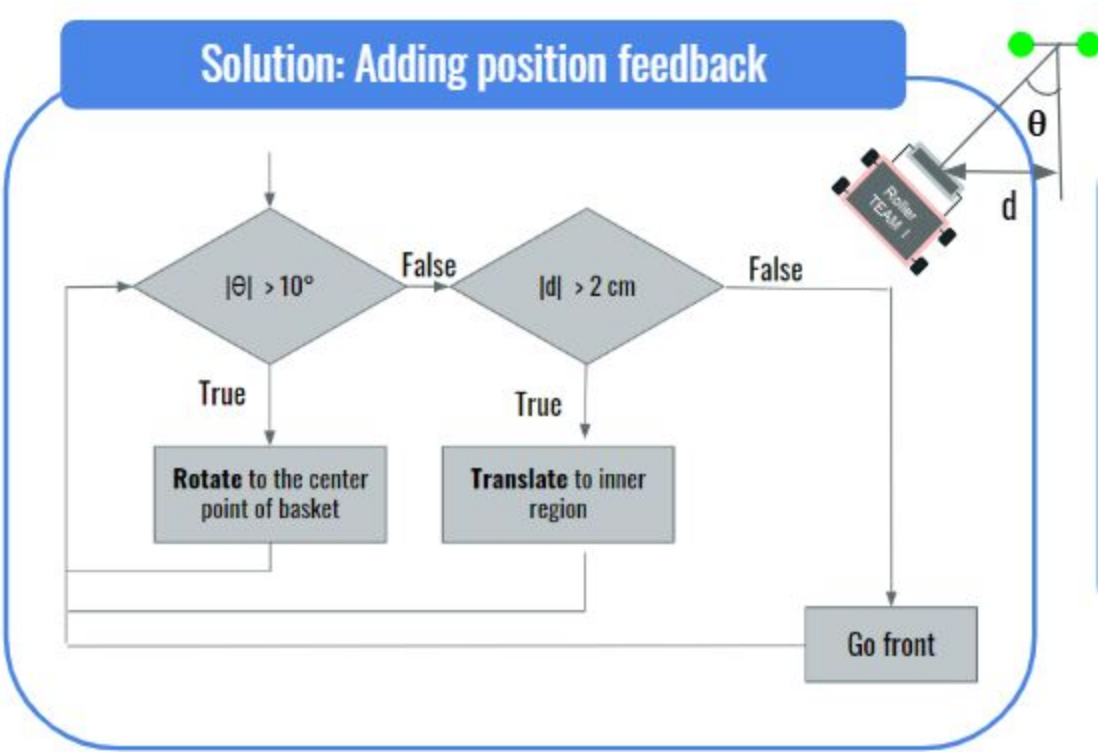
Case 1: in this case, only one green ball is detected. If the distance to the detected green ball is larger than 2 meters, then close this distance while maintaining the direction toward the ball. After reaching this distance, robot keeps turning slowly to detect the other green ball.

Case 2: In this case, both two green balls are detected. A feedback loop is implemented to align the robot with basket. The details of the scheme as presented in the flowchart below:







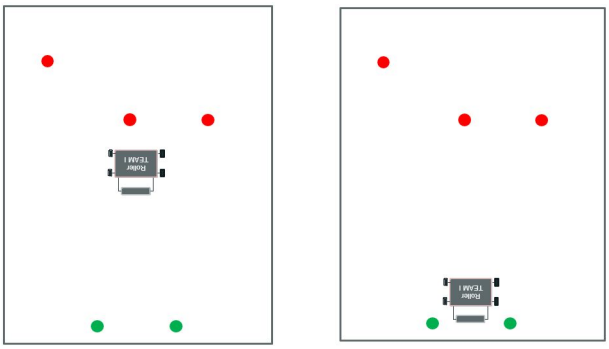


When the robot is perfectly aligned with the basket and 80 cm away from the basket, change to the RELEASE state. If red ball presents, changing to RED_AVOIDANCE state at any moment.

The explanation above discusses the general flow of how we search and align our vehicle towards the basket based on the positions of the green balls. In order to optimize the aligning time and accuracy during this phase, our LABVIEW team and ROS set up speed changes based on the angle deviation and horizontal distance deviation. At higher angle deviations and horizontal distances, we moved quickly disregarding the overshoot it can result in. When the angle deviation and horizontal distance comes into a range in which delicate alignment and precise movement is required, we slowed the vehicle down. This allowed our vehicle to be accurate and fast.

558 ~ 600: RELEASE state

If lidar is not used, robot goes front based on the distance to basket calculated in SEARCH_GREEN state, then reverse the roller to release the ball into the basket.



608 ~ 617: send data to LabView via TCP/IP

If the way to move is decided, send 24 bit 1 array named 'data' to Myrio by 'write' function. Then, print message to identify whether TCP/IP is working well.

619 ~625: ROS sleep, spin, shutdown + close socket

We used ros.sleep function to make timer_ticks to increase regularly. The duration is 1/40sec so timer_ticks increase 1 per 1/40sec. Finally, we can use timer_ticks to make our robot to go or turn about certain amount exactly.

631 ~ 685: Main (bottom) camera callback function

If the message from webcam node is arrived, this callback function is implemented. This function is for bottom camera and it updates the position and the number of balls of each color. First of all, we get the number of blue ball and the position of each blue ball. Then, because the position of webcam and roller is different, we should choose proper offset to chance coordinate to roller. This procedure is repeated for red balls and green balls.

687 ~ 734: Top camera callback function

If the message from webcam node is arrived, this callback function is implemented. This function is for top camera and it updates the position and the number of balls of each color. First of all, we get the number of blue ball and the position of each blue ball. Then, because the position of webcam and roller is different, we should choose proper offset to change coordinate to roller. This procedure is repeated for red balls and green balls.

735 ~ 753: Counter camera call back function

If the message from webcam node is arrived, this callback function is implemented. This function is for counter camera and it updates whether the number of collected blue balls are 3 or less. Because the collected balls are not separated, the webcam cannot count the number of balls. So instead of counting the number we just get the radius of total enclosing circle and if the radius is bigger than 191, we regard it as the number of collected balls is 3 so the roller node will publish 1 and if it is less than 191, the node will publish 0. Serious problem was that when two blue balls are collected and seperated little bit, the radius of enclosing circle become sometimes over 191. To solve this problem, we defined a new variable named 'dupACKcnt' and if the subscribed number is 1 over three times, we made the machine_status to LIDAR_RETURN. In other words, the situation that the radius become over 191 because of vibration when the number of collected ball is two would not change the status to LIDAR_RETURN.

755 ~ 772: ball_in_range function (boolean)

In case that the color of detected ball is blue, if the position of detected blue ball is inside the range of +/- 0.034 m lateral and 0.3 m longitudinal, return true, conforming that blue ball is in range.
In case that the color of detected ball is red, change to red_in_range function.

773 ~ 780: red_in_range function (boolean)

The detected red ball is considered as inside the range when red ball's lateral position is different from the center less than +/- 0.25 m, and red ball's longitudinal position is less than 0.3 in front of the robot.

782 ~ 840: leftmost_green(_top), rightmost_green (redundant, NOT USED)

843 ~ 863: leftmost_blue

From the lateral positions of blue balls which are detected by the main camera, we sort out the ball with minimum **lateral** position value. That will be the leftmost blue ball.

864 ~ 885: leftmost_blue_top

From the lateral positions of blue balls which are detected by the top camera, we sort out the ball with minimum **lateral** position value. That will be the leftmost blue ball.

885 ~ 920: target_blue policy (LEFTMOST, CENTERMOST, CLOSEST)

Define which policy amongst LEFTMOST, CENTERMOST, CLOSEST to be executed for the main camera and top camera.

923 ~ 942: centermost_blue(), return index of centermost blue ball amongst visible ones

From the lateral positions of blue balls which are detected by the main camera, we sort out the ball with minimum **absolute lateral** position value. That will be the centermost blue ball.

944 ~ 965: closest_blue_top(), return the index for the closest blue ball array seen by top camera

From the longitudinal positions of blue balls which are detected by the top camera, we sort out the ball with minimum **longitudinal** position value. That will be the closest blue ball seen by top camera.

967 ~ 1030: closest_ball(enum color ball_color), return the index for the closest (blue, red, green) ball seen by bottom camera

Depending of the color variable, from the longitudinal positions of variable balls which are detected by the main camera, we sort out the ball with minimum **longitudinal** position value. That will be the closest variable ball seen by main camera.

1032 ~ 1049: furthest_green(REDUNDANT, NOT USED)

1053 ~ 1067: lidar_Callback function, receive x, y, θ values from lidar

Subscribe to lidar topic and get absolute fixed ground coordinates of the robot: lateral position xpos, longitudinal position ypos, orientation position theta_abs.

1069 ~ 1095: dataInit, reset data from index 0 to index 23

2.2.2. util_rtn.hpp(header file) - key variables

81: bool red_in_range()

if red ball position x= +-25 cm, z<=30 cm 일때 true , else false

82: bool ball_in_range()

if blue ball x= +-3.4 cm, z<=30 cm 일때 true, else false
elseif red ball --> red_in_range
else false

87: int closest_ball(enum color ball_color)

Return index of array about ball having color which is the closest seen from the bottom webcam. If nonexistent, return -1

88: int centermost_green()

Returns index of array about the centermost green ball seen from bottom webcam. If nonexistent, return -1

89: int leftmost_green()

Returns index of array about the leftmost green ball seen from bottom webcam. If nonexistent, return -1

90: int rightmost_green()

Returns index of array about the rightmost green ball seen from bottom webcam. If nonexistent, return -1

91: int leftmost_green_top()

Returns index of array about the leftmost green ball seen from top webcam. If nonexistent, return -1

92: int closest_green()

Returns index of array about the closest green ball seen from bottom webcam. If nonexistent, return -1

94: int target_blue(int policy)

Returns index of blue ball located leftmost, center, and rightmost

96: int leftmost_blue()

Returns index of array about the leftmost blue ball seen from bottom webcam. If nonexistent, return -1

97: int centermost_blue()

Returns index of array about the centermost blue ball seen from bottom webcam. If nonexistent, return -1

98: int leftmost_blue_top()

Returns index of array about the leftmost blue ball seen from top webcam. If nonexistent, return -1

2.2.3. openCV nodes - key variables

2.2.3.1. main.cpp

Publish ball_position.msg in core_msgs. There are 12 variables that 4 variable for blue, red, green ball each. Int32 size_r represents the number of red balls in bottom camera, and float32[] imag_x_r, img_y_r, and img_z_R is the array of x ,y, z positions of red balls. Other variables for green, blue balls are same.

2.2.3.2 main_top.cpp

This is same with main.cpp but the only difference is that this node uses top camera. Publish ball_position_top.msg in core_msgs. There are 12 variables that 4 variable for blue, red, green ball each. Int32 size_r represents the number of red balls in top camera, and float32[] imag_x_r, img_y_r, and img_z_R is the array of x ,y, z positions of red balls. Other variables for green, blue balls are same.

2.2.3.3 roller.cpp

Publish binary value whether the number of collected blue balls are 3 or less. Because the collected balls are not separated, the webcam cannot count the number of balls as using min enclosing circle function that is the most stable function to find blue circle. So instead of counting the number we just get the radius of total enclosing circle and if the radius is bigger than 191, we regard it as the number of collected balls is 3 so the roller node will publish 1 and if it is less than 191, the node will publish 0.

3. Philosophy and ideas

3.1 Philosophy:

- Keep everything as simple as possible. We try to keep everything from logic thinking to code organization in the most minimal style.

- Keeping that mindset, our logic started from dividing the main routine of robot into phases in order to easily execute and debug, which included Search-Collect blue balls, Avoid red ball, Return-release balls. The success in each phase was essential to a complete success of the whole project. After achieving the success in each phase, integrating them together was a much easier task. In short, our simple strategy "divide-and-conquer" gave us the best results amongst teams.

- Secondly, in code organization, unlike several previous year teams, we created a separate header file to store macro and function prototypes. Moreover, we always kept our codes organized in block style. These simple, yet efficient techniques helped us tremendously in adding more functions, variables or parameters, and, most importantly, debugging.

3.2 Ideas:

- States construction: Based on only separate basic movements: go front, turn left-right, move aside, Turn on-off roller motor, a combination of some of these movements created fundamental states, including INIT, SEARCH, APPROACH, RED_AVOIDANCE, COLLECT, COLLECT2, LIDAR_RETURN, SEARCH_GREEN, RELEASE .

- Switching between states following the feedback received from vision: Our robot relied on vision to decide whether it achieved the success of each phase. Therefore, it was critical to have a clear decision line between states in order to keep the system running reliable and stable (not randomly swing back and forth between states).

- Back-up plans for unexpected problems: even the hardware have worked hard on creating a stable system, but several unexpected external problems might come up during the development of the project. Therefore, it is important to have a back-up solution in case of dealing with an unstable system. In addition, since finishing time is one of the main criteria even without completing the mission of pick up, in our opinion, a feature that makes the robot return after a certain time would be necessary. As a result, time-out function should be always included in the routine.

4. Desired coding schedule

(2/28/2019 - 5/31/2019)		Week 2	Week 3	Week 4	Week 5	Week 6	Week 7	Week 8	Week 9	Week 10	Week 11	Week 12	Week 13	Week 14
ROS basic study														
Algorithm														
	Idea generation													
	Simulation													
System integration														
	Integrating with LabView													
	Integrating with OpenCV													
	Integrating with Lidar													
Coding implementation														
	Searching for blue ball													
	Approaching target ball													
	Collecting target ball													
	Avoiding red ball													
	Searching green balls and approaching basket													
	Releasing ball													
Testing the complete code and revising														

Appendix

A.1.1. Timer features

(1) Overview

A global timer, *timer_ticks* is defined. The value is incremented by 1 on each subscription loop.

data_integration/src/util_rtn.hpp

```
46  /* For timer features */
47  static uint32_t timer_ticks = 0;
48  static uint32_t current_ticks = 0;
```

data_integration/src/mainv2.cpp

```
806         ros::Duration(DURATION).sleep();
807         ros::spinOnce();
808         timer_ticks++;
809       }
810
```

DURATION = 25msec in our implementation. Therefore, 40 timer ticks is equivalent to 1sec(= 40 x 25ms).

(2) Open-loop control

The key idea is that speed is constant, so distance is proportional to duration. All open-loop control uses this feature by evaluating equivalent timer ticks corresponding to desired distance.

In our implementation, there are 3 cases using this feature:

1) RED_AVOIDANCE : In the second step of the phase, goes front for DISTANCE_TICKS.

```
315           case RED_AVOIDANCE:
316           {
317
 …
325               GO_FRONT
326
327               if(timer_ticks - current_ticks > DISTANCE_TICKS) {
328                 red_phase2 = false;
329                 machine_status = (return_mode)? SEARCH_GREEN : SEARCH;
```

2) COLLECT2 : Turns on roller, goes front for DISTANCE_TICKS_CL ticks.

```
368           case COLLECT2:
369           {
370             if(timer_ticks - current_ticks <= 40) GO_FRONT
371             ROLLER_ON
372             if(timer_ticks - current_ticks > DISTANCE_TICKS_CL) {
373               machine_status = SEARCH;
```

3) RELEASE : Go front for equivalent ticks evaluated from SEARCH_GREEN.

```
756           case RELEASE:
757           {
758             #ifndef LIDAR
759             uint32_t goal_front_ticks = (uint32_t) (90.0f * (goal_z));
```

(3) Mechanism

1) Save current_ticks at the start of the phase.

```
354                 current_ticks = timer_ticks;
355                 machine_status = COLLECT2;
```

2) Compare timer_ticks and current_ticks.

(time_elapsed_ticks) = (timer_ticks) - (current_ticks)

```
372             if(timer_ticks - current_ticks > DISTANCE_TICKS_CL) {
```

3) Multiplex through time_elapsed_ticks.

```
761             if(timer_ticks-current_ticks < goal_front_ticks) {
762               MSGE("RELEASE - go front")
763               GO_FRONT
764             } else if(timer_ticks - current_ticks < 100 + goal_front_ticks) {
765               MSGE("RELEASE - roller_reverse")
766               ROLLER_REVERSE
767             } else {
768               printf("(%s) elapsed time = %.4f sec\n", TESTENV, 0.025 * timer_ticks);
769               PANIC("RELEASE_TERMINATE : should have released 3 balls.")
770             }
```