2019 Spring ME492 Final Report

Team G
Hyeonjae Gil, 20160062
Byeongju Lee, 20160456
Jongwon Lee, 20140447
Joosung Lee, 20150600
Yoonseop Lim, 20160554
Seungyeon Jang, 20150658

2019.06.16

1. 서론

1.1 코드의 철학

우리 팀은 어떠한 공 배치에 대해서도 미션을 성공시킬 수 있고, 간단하고 직관적인 코드를 만드는 것을 기본 철학으로 삼았다.

서식 있음: 줄 간격: 배수 1.15 줄

1.2 아이디어

우리의 기본 철학에 기반하여 코드를 작성하는 과정에서 크게 두 가지 아이디어가 제시되었다.

1.2.1 함수의 모듈화, 위계화

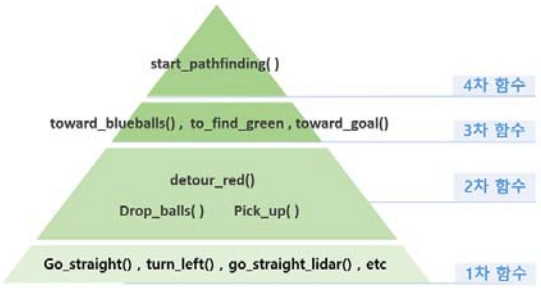


그림 1 우리 코드의 함수 위계구조

간단하고 직관적인 코드를 만들기 위해 제시된 가장 우선적인 아이디어는 함수를 모듈화한 후 function hierarchy를 만드는 것이다. 함수를 모듈화하면 함수별로 특정 기능을 독립적으로 수행할 수 있으며, 각 함수는 독립적으로 구현될 수 있다. 따라서 코드의 각 계층의 오류를 수정하거나, 기능을 향상시켜야 할 때 전체 시스템을 수정하지 않고 해당 계층의 모듈만 수정하는 것으로 튜닝과 디버깅을 완료할 수 있다. 예를 들어, detour_red()함수가 잘못된 것을 발견했다면 1차 함수들은 수정할 필요 없이 detour_red()함수만 수정하면 된다. 실제로 하드웨어와 소프트웨어를 통합하는 과정에서 여러 가지 문제점이 발생했는데, 우리의 코드는 디버깅이 용이하여 어느 파트에서 오류가 발생한 것인지 쉽게 찾을 수 있어 작업의 효율을 높일 수 있었다. 총 4개의 층으로 코드를 작성했으며, 각 상위 단계는 하위 단계의 함수들의 조합으로 실행된다. 결과적으로, 실제 데모에서는 이를 모두 아우르는 4차 함수인 start_pathfinding() 함수만을 실행하였다.

서식 있음: 줄 간격: 배수 1.15 줄

1.2.2 “3 Rule operation”

함수 단위의 코딩은 디버깅이 용이하다는 분명한 장점이 있었지만, 알고리즘이 일관된 규칙을 따르지 않을 경우 공 배치가 달라질 때마다 코드를 수정해야 하는 문제가 발생하였다. 따라서 우리는 세 가지 절대적인 규칙을 만들었고, 이를 “3 Rule operation”이라고 이름 붙였다. 그 규칙들은 다음과 같다.

Rule 1. 로봇의 시야에서 가장 오른쪽에 있는 파란색 공을 타겟으로 정한다.

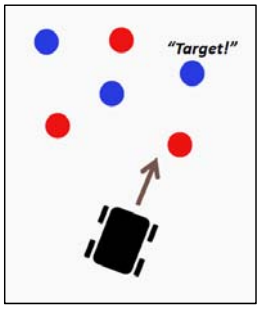


그림 2-1 첫 번째 규칙

Rule 2. 파란색 공을 줍고, 그 다음 파란색 공을 주워야 하는 상황에서 로봇의 시야에 파란색 공이 보이지 않는 경우 로봇이 공을 찾기 위해 반시계 방향으로 회전한다.

서식 있음: 줄 간격: 배수 1.15 줄

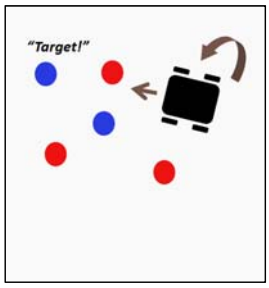


그림 2-2 두 번째 규칙

Rule 3. 그러나, 로봇의 시야에서 가장 오른쪽에 있는 공이 로봇으로부터 가장 가까이 있는 공이 아닌 경우에는 가까운 공부터 줍는 것이 시간상 효율적이다. 따라서 이러한 경우에는 가장 가까운 공을 타겟으로 하고, 시야에서 가장 오른쪽에 있는 공을 다음 타겟으로 한다. 이러한 방식으로 타겟의 우선순위를 설정하는 알고리즘을 'checker'라고 명명하였다.

서식 있음: 줄 간격: 배수 1.15 줄

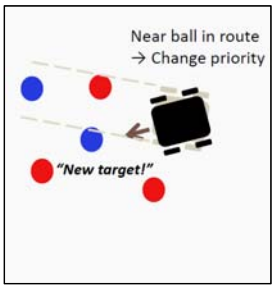


그림 2-3 세 번째 규칙

서식 있음: 줄 간격: 배수 1.15 줄

2.1 Callback 함수 및 main 함수

2.1.1 설명

우리의 로봇은 RGBD카메라와 RPLIDAR를 사용하였다. 따라서 camera_Callback 함수와 lidar_Callback 함수를 main 함수에서 실행함으로써 각 센서가 발신하는 메시지를 받는다.

서식 있음: 글꼴: (영어) 바탕체, (한글) 바탕체

서식 있음: 글꼴: (영어) 바탕체, (한글) 바탕체

2.1.2 camera_Callback(const core_msgs::ball_position::ConstPtr& position)

```
void camera_Callback(const core_msgs::ball_position::ConstPtr& position)
{
  blue_number = position->blue_size;   // put number of blue balls
  near_blue = 0;                       // initialize index
  near_center_blue = 0;
  rightmost_blue = 0;
  for(int i = 0; i < blue_number; i++)
    {
        blue_X[i] = (position->blue_x[i])+110;  // revise the value of x coordinate of blue ball correctly
        blue_Y[i] = position->blue_y[i];        // put y coordinate of blue ball
        blue_distance[i] = position->blue_distance[i];    // put distance of blue ball
        if(blue_distance[i]<blue_distance[near_blue]){    // put index of the nearest blue ball
          near_blue = i;
        }
        if(abs(blue_X[i])<abs(blue_X[near_center_blue])){ // put index of blue ball at center
          near_center_blue = i;
        }
        if(blue_X[rightmost_blue]<blue_X[i]){             // put index of rightmost blue ball
          rightmost_blue = i;
        }
    }
  red_number = position->red_size;     // put number of red balls
  near_red = 0;                        // initialize index
  for(int i = 0; i < red_number; i++)
    {
        red_X[i] = (position->red_x[i])+110;          // revise the value of x coordinate of red ball correctly
        red_Y[i] = position->red_y[i];                // put y coordinate of red ball
        red_distance[i] = position->red_distance[i]; // put distance of red ball
        if(red_Y[i]<red_Y[near_red]){                // put index of the nearest red ball
          near_red = i;
        }
    }
  green_number = position->green_size;               // put number of green balls
  rightmost_green = 0;                               // initailize index
  for(int i = 0; i < green_number; i++)
    {
      green_X[i] = (position->green_x[i])+110;         // revise the value of x coordinate of gree ball correctly
      green_Y[i] = position->green_y[i];               // put y coordinate of green ball
      green_distance[i] = position->green_distance[i]; // put distance of green ball
      if(green_X[rightmost_green]<green_X[i]){         // put index of the nearest green ball
          rightmost_green = i;
        }
    }
}
```

이 함수의 주 목적은 카메라의 시야에서 본 세 가지 색깔의 공들의 x, y 좌표와 거리 정보를 담고 있는 core_msgs::ball position 형의 메시지를 받아 전역 변수로 저장하는 것이다.
또한, 다른 함수에 필요한 특정 값도 전역 변수에 입력한다. 크게 파란 공, 빨간 공 그리고 초록 공 각각에 대한 변수를 입력하는 세 개의 부분으로 나눌 수 있다. 공통적으로 먼저 값을 입력하기 전에 배열의 크기를 설정하고 전역변수들을 초기화한 다음 x, y 좌표와 거리의 값을 입력한다. 이때 x 좌표를 입력하는 부분에서 110을 더해주는데 이는 calibration 과정에서 원점을 로봇의 중앙이 아닌 pickup module의 가장 오른쪽으로 설정하였기 때문이다. 따라서 로봇의 중심축을 기준으로 이 메시지를 다루고자, 로봇의 중앙 축과 pickup module의 가장 오른쪽 부분 사이의 간격인 110mm을 더하였다. 공통적인 부분을 제외한 각 값들은 if 문을 사용하여 가장 조건에 맞는 공의 index를 찾았다.

2.1.3 lidar_Callback(const sensor_msgs::LaserScan::ConstPtr& scan)

서식 있음: 줄 간격: 배수 1.15 줄

```
void lidar_Callback(const sensor_msgs::LaserScan::ConstPtr& scan)
{
  map_mutex.lock();  // protect collision of two callback functions
  int count = scan->scan_time / scan->time_increment; // compute lidar size by scan message
  lidar_size=count;  // put size of lidar angle data
  zero_degree = 0;   // initialize zero_degree
  least_distance = 0;  // initialize least_distance
  for(int i = 0; i < count; i++)
    {
      lidar_degree[i] = RAD2DEG(scan->angle_min + scan->angle_increment * i);  // put real angle of robot that unit is degree
      if(static_cast<int>(lidar_degree[i]+0.5) == 0){zero_degree = i;}  // check that the angle is zero
      lidar_distance[i]=scan->ranges[i];  // put distances of robot at each angles
      if(lidar_distance[i]<lidar_distance[least_distance]){least_distance = i;}  // put the least distance of robot
    }
  if(isinf(scan->ranges[zero_degree])){   // check range is infinite
    lidar_zero_degree = lidar_zero_degree;
  }
  else{
    lidar_zero_degree = scan->ranges[zero_degree];  // put the distance of robot when is zero
  }
  map_mutex.unlock();
}
```

이 함수의 주 목적은 RPLIDAR가 메시지로 전달한 각도와 각도마다의 거리 값을 받아 다른 함수에 사용할 수 있도록 전역 변수에 입력하는 것이다. 먼저 camera_Callback 함수의 값과 충돌이 일어날 수 있어 lock 함수로 이를 막았다. 그 다음 RPLIDAR 메시지의 배열 크기를 계산하여 lidar_size에 대입하고 다른 값들을 초기화한다.
이후 lidar_degree와 lidar_distance 변수에 각도와 각각의 각도에서 바라보는 벽까지의 거리를 입력한다. 이때 로봇이 보고 있는 방향인 0도인 index를 zero_degree에, 벽까지의 최소의 거리인 index를 least_distance에 입력한다. 그 다음 0도에서의 거리가 무한인지 조건문으로 확인하고 유한한 값이라면 그 값을 lidar_zero_degree에 입력한다.

2.1.4 main 함수

```
int main(int argc, char **argv){

  ros::init(argc, argv, "data_integrate_node");
  ros::NodeHandle nh;

  ros::Subscriber sub1 = nh.subscribe<core_msgs::ball_position>("/ball_position", 100, camera_Callback); // subscribe ball position data by camera
  ros::Subscriber sub = nh.subscribe<sensor_msgs::LaserScan>("/scan", 1000, lidar_Callback);  // subscribe robot position data by lidar

  c_socket = socket(PF_INET, SOCK_STREAM, 0);  // TCP/IP
  c_addr.sin_addr.s_addr = inet_addr(IPADDR);  // change IP address type from string to long int
  c_addr.sin_family = AF_INET;                 // put IPv4 internet protocol to address family
  c_addr.sin_port = htons(PORT);               // change short memory value from Host byte of PORT to Network byte
  connect(c_socket, (struct sockaddr*) &c_addr, sizeof(c_addr));  // request to connect server through c_socket

  start_path_finding();   // do start_path_finding
}
```

이 함수는 최종적으로 메시지를 받는 콜백 함수를 실행하고 TCP/IP로 서버에 연결한 다음 start_path_finding 함수를 실행시킨다.

2.2 1단계 함수

2.2.1 설명

1단계의 함수는 로봇이 받는 신호를 초기화하고 회전, 직진과 같은 로봇의 기초적인 이동 기능을 수행하는 함수이다. 카메라의 피드백 제어를 통해 움직이는 함수들과 RPLIDAR를 통해 움직이는 함수로 나뉘어 있다. 덧붙여, 로봇을 공에 정밀하게 정렬하고자 느리게 회전하는 함수도 구현하였다.

2.2.2 dataInit()

서식 있음: 줄 간격: 배수 1.15 줄

```
void dataInit(){
  data[0] = 0; //lx*data[3];
  data[1] = 0; //ly*data[3];
  data[2] = 0; //GamepadStickAngle(_dev, STICK_LEFT);
  data[3] = 0; //GamepadStickLength(_dev, STICK_LEFT);
  data[4] = 0; //rx*data[7];
  data[5] = 0; //ry*data[7];
  data[6] = 0; //GamepadStickAngle(_dev, STICK_RIGHT);
  data[7] = 0; //GamepadStickLength(_dev, STICK_RIGHT);
  data[8] = 0; //GamepadTriggerLength(_dev, TRIGGER_LEFT); lb
  data[9] = 0; //GamepadTriggerLength(_dev, TRIGGER_RIGHT); rb
  data[10] = 0; //GamepadButtonDown(_dev, BUTTON_DPAD_UP); //horizontal align(pickup)
  data[11] = 0; //GamepadButtonDown(_dev, BUTTON_DPAD_DOWN);
  data[12] = 0; //GamepadButtonDown(_dev, BUTTON_DPAD_LEFT);
  data[13] = 0; //GamepadButtonDown(_dev, BUTTON_DPAD_RIGHT);
  data[14] = 0; //GamepadButtonDown(_dev, BUTTON_A); // duct on/off
  data[15] = 0; //GamepadButtonDown(_dev, BUTTON_B); //우회전
  data[16] = 0; //GamepadButtonDown(_dev, BUTTON_X); //좌회전
  data[17] = 0; //GamepadButtonDown(_dev, BUTTON_Y); //직진
  data[18] = 0; //GamepadButtonDown(_dev, BUTTON_BACK);
  data[19] = 0; //GamepadButtonDown(_dev, BUTTON_START);  //느린 회전 모음
  data[20] = 0; //GamepadButtonDown(_dev, BUTTON_LEFT_SHOULDER); //pick up
  data[21] = 0; //GamepadButtonDown(_dev, BUTTON_RIGHT_SHOULDER);
  data[22] = 0; //GamepadButtonDown(_dev, BUTTON_LEFT_THUMB);
  data[23] = 0; //GamepadButtonDown(_dev, BUTTON_RIGHT_THUMB);
}
```

로봇이 움직이기 위한 모터 제어 신호를 초기화하는 함수이다. 총 24개의 신호를 줄 수 있는데, 이 dataInit() 함수에서는 일단 이 값을 모두 0으로 초기화한다.

서식 있음: 줄 간격: 배수 1.15 줄

2.2.3 turn_right(), turn_right_slow(), turn_left(), turn_left_slow()

```
void turn_right(){
  dataInit();
  data[15]=1; //오른쪽으로 이동
  write(c_socket, data, sizeof(data));
  ros::Duration(0.025).sleep();
  ros::spinOnce();
}

void turn_right_slow(){
  dataInit();
  data[15] = 1; //오른쪽으로 이동
  data[19] = 1; //느린 이동
  write(c_socket,data,sizeof(data));
  ros::Duration(0.025).sleep();
  ros::spinOnce();
}
```

```
void turn_left(){
  dataInit();
  data[16]=1; //왼쪽으로 이동
  write(c_socket, data, sizeof(data));
  ros::Duration(0.025).sleep();
  ros::spinOnce();
}

void turn_left_slow(){
  dataInit();
  data[16] = 1; //왼쪽으로 이동
  data[19] = 1; //느린 이동
  write(c_socket,data,sizeof(data));
  ros::Duration(0.025).sleep();
  ros::spinOnce();
}
```

Turn_right는 로봇을 오른쪽 방향으로 회전시키는 함수이다. 먼저 각 신호를 초기화한 다음 오른쪽으로 이동하는 신호인 15번을 활성화하고 이 신호를 c_socket에 보낸다. 그 다음 이 함수를 다시 받을 때까지 기다린다. Turn_right_slow는 turn_right와 같이 오른쪽으로 회전함과 동시에 속도를 제어하는 19번 신호를 주어 느리게 움직이게 만드는 함수이다.

Turn_left는 turn right와 같은 알고리즘이지만 16번의 신호를 주어 왼쪽으로 이동한다. 그리고 turn_left_slow도 turn_right_slow와 같이 turn_left에서 19번의 신호도 주어 느리게 움직이게 만드는 함수이다.

2.2.4 go_straight(), move_backward(), move_backward(float distance)

```
void go_straight(){
  dataInit();
  data[17]=1; //직진
  write(c_socket, data, sizeof(data));
  ros::Duration(0.025).sleep();
  ros::spinOnce();
}

void move_backward(){ //Used for no blue ball found
  dataInit();
  data[14]=1; //후진
  write(c_socket, data, sizeof(data));
  ros::Duration(0.025).sleep();
  ros::spinOnce();
}
```

```
void move_backward(float distance){
  for(int i = 0;i<12;i++){  //12/40초 동안
    dataInit();
    data[14] = 1;
    write(c_socket,data,sizeof(data));
    ros::Duration(0.025).sleep();
  }
  dataInit();
  write(c_socket,data,sizeof(data));
  ros::Duration(0.3).sleep();
}
```

Go_straight() 함수는 로봇을 앞으로 직진시키는 함수이다. 먼저 각 신호를 초기화한 다음 직진 신호인 17번을 활성화시키고 이 신호를 c_socket에 보낸다. 그 다음 이 함수를 다시 받을 때까지 기다린다. Move_backward()는 go_straight()와 같이 후진하는 14번 신호를 주어 뒤로 움직이게 만드는 함수이다.
Move_backward(float distance)는 move_backward()에 for 반복문을 사용하여 로봇이 12/40초 동안 뒤로 움직이게 만드는 함수이다.

2.2.5 turn_lidar(float angle, int direction), go_straight_lidar(float distance), go_pickup_lidar(float distance)

```
void turn_lidar(float angle, int direction){//direction = 1 좌회전, direction = 0 우회전 1
  ros::spinOnce();
  if(angle>90){
    turn_lidar(angle/2,direction);
    turn_lidar(angle/2,direction);
  }
  float initial_angle = lidar_degree[least_distance];
  float moved_angle = lidar_degree[least_distance] - initial_angle;
  while(abs(moved_angle)<angle){
    ros::spinOnce();
    dataInit();
    data[15+direction] = 1;
    write(c_socket,data,sizeof(data));

    moved_angle = lidar_degree[least_distance] - initial_angle;
    if(abs(moved_angle)>180){
      if(moved_angle<0){
        moved_angle = moved_angle + 180;
      }
      if(moved_angle>0){
        moved_angle = moved_angle - 180;
      }
    }
    ROS_INFO("Turned : %f",abs(moved_angle));
    ros::Duration(0.025).sleep();
  }
}
void go_straight_lidar(float distance){
  ros::spinOnce();
  cout<<"lidar_moving : "<<distance<<endl;
  float initial_distance = lidar_zero_degree;
  float moved_distance = initial_distance-lidar_zero_degree;
  int t = 0;
  while(moved_distance < distance){
    t++;
    moved_distance = initial_distance-lidar_zero_degree;
    cout<<"lidar moved : "<<moved_distance<<endl;
    dataInit();
    data[17] = 1;
    write(c_socket, data, sizeof(data));
    ros::Duration(0.025).sleep();
    ros::spinOnce();
  }
}
void go_pickup_lidar(float distance){
  ros::spinOnce();
  cout<<"lidar_moving : "<<distance<<endl;
  float initial_distance = lidar_zero_degree;
  float moved_distance = initial_distance-lidar_zero_degree;
  int t = 0;
  while(moved_distance < distance){
    t++;
    ros::spinOnce();
    moved_distance = initial_distance-lidar_zero_degree;
    cout<<"lidar moved : "<<moved_distance<<endl;
    dataInit();
    data[17] = 1;
    data[10] = 1;
    write(c_socket, data, sizeof(data));
    ros::Duration(0.025).sleep();
  }
}
```

앞선 함수들이 모두 시간 기반으로 모터를 동작한다면, 이 함수들은 모두 RPLIDAR로부터 발신되는 주위 환경의 정보를 사용해 피드백하며 움직이는 함수이다.
turn_lidar 함수는 'angle' argument만큼 좌회전 또는 우회전하는 함수이다. Lidar를 통한 회전 알고리즘은 RPLIDAR의 중심이 회전축과 일치하다는 것을 전제로 하여, 처음 시점에 기준이 될 수 있는 가장 짧은 거리에 있는 벽의 각도를 저장한다. 그리고 15번 신호나 16번 신호를 주어 회전하면서 그 각도의 차이인 moved angle을 계산해 주는 것으로 원하는 각도를 회전한다. 그러나, 회전 각도가 90도가 넘어갈 경우 lidar data의 경계에서 문제가 생기기 때문에, 그 경우에만 각도를 절반으로 나누어 재귀적으로 실행한다. 그리고 조건문을 활용하여 moved_angle 값을 180, 또는 -180을 더하여 -180도 이상 180도 이하로 맞춘다.
Go_straight_lidar 함수는 'distance' argument만큼 로봇을 앞으로 직진시키는 함수이다. 먼저 로봇 기준으로 0도에서의 거리를 initial_distance로, 움직인 거리를 moved_distance로 정의한다. 이후 움직인 거리가 목표하는 distance보다 클 때까지 move_distance를 갱신하면서 직진한다.
Go_pickup_lidar 함수는 go_straight_lidar 함수에서 10번의 신호를 주어 공을 줍기 위해 로봇의 blade를 지면과 나란히 유지하는 과정이 추가되었다. 나머지는 go_straight_lidar와 알고리즘이 동일하다.

2.3 2단계 함수 - Detour red, Pickup & Drop
2.3.1. 설명
이 단계에서 설명하는 함수들은 파란 공을 줍고, 빨간 공을 피하고, goal position에 파란 공을 놓는 기능을 수행한다. 각 함수들은 모두 시간 기반 모터 제어를 기본으로 하되, detour_red() 함수에서 빨간 공을 안전하게 피할 수 있는 방향으로 회전하는 단계에서는 카메라로부터의 시각 데이터로 계속 주위 공들의 위치를 확인하며 회전하는 피드백 제어를 거친다. 시간 기반으로 모터 제어를 하게 되면 지형의 굴곡이나 마찰계수 등 주위 환경의 변화에 대응할 수 없다는 단점이 있으나, 창의적시스템구현의 시연 환경은 엄격히 통제되기에 시간 기반으로 각 움직임을 제어해도 크게 무리가 없음을 확인했다.

2.3.2. 위치 기반 제어가 아닌 시간 기반 제어를 사용한 이유
처음 우리는 RPLIDAR를 기반으로 외벽으로부터 로봇의 위치를 계산해 제어 명령을 내리고자 했으나, 이는 오차가 상당히 크다는 것을 알았다. 빨간 공을 피해 파란 공을 정확히 줍기 위해서는 수 cm 이내의 정밀한 제어가 필요한데, RPLIDAR는 주파수가 작고 오차도 있어 로봇의 위치를 미세하게 조정하기는 어렵다는 것을 느꼈다. 이에 우리는 시간 기반의 모터 제어 명령을 내려 회전이나 직진 움직임의 신뢰성과 정밀성을 동시에 잡고자 하였다.
다만, 다음 장에서 언급하겠지만 파란 공 세 개를 다 줍고 난 뒤 goal position으로 돌아가기 위해서 로봇의 위치를 정확히 추정할 필요가 있는데, 이 과정에서는 RPLIDAR가 제 역할을 온전히 수행한다.

2.3.3 detour_red()

```
void detour_red(){
  ros::Duration(0.1).sleep();
  ros::spinOnce();
  int near_red_X = red_X[near_red]; // The nearest red ball's x-coordinate wrt. the robot
  int near_blue_X = blue_X[near_blue]; // The nearest blue ball's x-coordinate wrt. the robot
  int t = 0; // Measure how much the robot have turned to detour a ball
```

주요 변수들을 선언한다. 현재 로봇으로부터 가장 가까이 있는 빨간 공(즉, 우회해야 할 대상)의 x좌표를 int 형 변수인 near_red_X로, 빨간 공을 피해 로봇이 우회한 척도를 나타내는 변수인 int t를 초기화한다. near_red는 가장 가까이 있는 빨간 공의 index이다.

```
if(near_red_X>0){ // If the nearest red ball is on the right side of the robot...
  int red_X_prev = near_red_X; // Initialize red_X_prev (the detoured ball's x-pos) as near_red_X
  while(red_number != 0 && red_X_prev<185){ // While any red ball is visible and red_X_prev is in the range able to collide...
    int now = -1; // The variable should indicate the 'detoured ball'. i.e. red_X[now] denotes the x-pos of the ball which is detoured
    int smallest_delta = 100;
    for(int j = 0;j<red_number;j++){ // For all elements in red_X[] ...
      int delta_X = (red_X[j]-red_X_prev); // Calculate the distance wrt. red_X_prev (the detoured ball's x-pos in the previous frame)
①   if(abs(delta_X) < smallest_delta){ // Choose the most similar value wrt. red_X_prev
        now = j; // And save its index so that red_X[now] means the x-pos of the ball which is detoured
        smallest_delta = abs(delta_X);
      }
    }
    if(now == -1){ // If the tracked ball (= detoured ball) is no more visible or etc ...
      ros::spinOnce();
②     ros::Duration(0.1).sleep();
      cout<<"now is -1"<<endl;
      break; // Break the while loop
      }
    red_X_prev = red_X[now]; // Update the the tracked ball (= detoured ball)'s x-pos
    cout<<now<<endl;
③   cout<<"red_X : "<<red_X_prev<<endl;
    turn_left(); // Turn left
    cout<<"After turning left, red number is "<<red_number<<endl;
    t++; // check the step how much the robot have turned so far
  }
}
```

만일 우회해야 할 빨간 공이 로봇을 기준으로 오른쪽에 있을 때 실행되는 조건문이다.

① 로봇은 빨간 공을 우회 하기 위해 while문의 실행 조건에 따라 1) 시야에 빨간 공이 더 이상 보이지 않거나 (red_number != 0) 혹은 우회하고 있는 공을 안전하게 회피할 수 있는 방향까지(red_X_prev<185) 왼쪽으로 회전하며, 시야에 보이는 가장 가까운 공의 index 정보를 전역변수인 near_red로 읽어 들인다. 그러나 이 과정에서 로봇의 시야각이 바뀌며 기존의 near_red와 현재의 near_red가 달라지는 경우가 생긴다(그림 3 참조). 이를 막고자 이전 프레임에서 피하고자 하는 빨간 공의 위치(red_X_prev)와 현재 프레임에서 보이는 모든 빨간 공(red_X[j]) 사이의 변위(delta_X)를 계산하여 지금 회피 중인 공의 index 정보를 추적, int형 변수 now에 저장해 준다(now = j, j = argmin(|red_X[j]- red_X_prev|), 그림 4 참조).

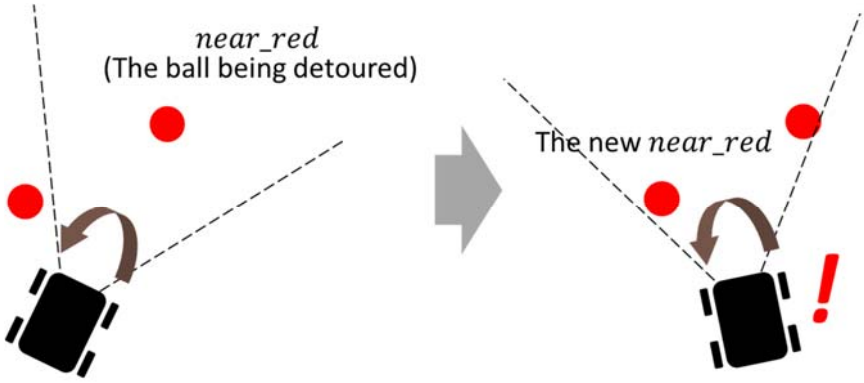


그림 3 로봇이 회전하며 가장 가까운 빨간 공이 바뀌는 경우. 이처럼 목표 회피물이 달라지면 detour_red가 원하던 바를 수행할 수 없다.

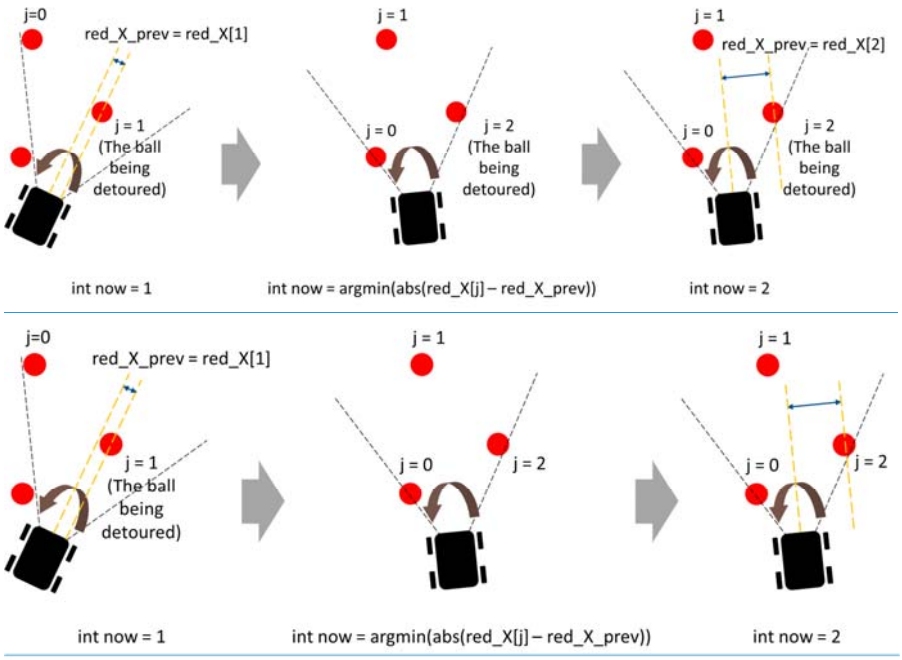


그림 4 원래 회피하던 공의 index를 계속 추적하는 알고리즘 ①

② 만일 추적 중인 공이 시야에서 벗어나면 now는 초기화 과정에서 선언되었던 값인 -1을 그대로 가지며, 이 경우 break문을 통해 루프 구문을 탈출함으로서 회피를 마친다.
③ 매 루프마다 현재 추적 중인 공의 index 정보(int now)와 로봇으로부터의 x축 방향 거리(red_X_prev)를 출력하며 turn_left() 함수를 통해 루프 종료 조건을 만족할 때까지 좌회전한다. 회전하는데 걸린 step의 수를 int형 변수 t에 저장한다.

```
else if(near_red_X <= 0){ ... }
```

만일 우회해야 할 빨간 공이 로봇을 기준으로 왼쪽에 있을 때, 공을 안전하게 회피할 수 있는 방향까지 우회전하는 코드이다. 앞서 설명한 것과 방향만 반대로, 정확히 같은 기능을 한다.

```
//Go straight a bit(time_based control)
for(int i = 0; i < 38 ; i++){ //For 38/40s
  go_straight(); // Go straight
}
cout<<"Avoidance finished!"<<endl;
```

빨간 공과의 충돌을 피하도록 방향을 돌린 후, 38/40초만큼 직진한다.

```
// Turn the opposite direction for 1.6t (time_based control)
if(near_red_X<0) // If the nearest red ball was on the left side of the robot, the robot must have detoured right, so...
  for(int i = 0;i < 1.6*t;i++)
    turn_left(); // Turn left so that the robot should aim where the robot pointed originally

else if(near_red_X>0) // Else if the nearest red ball was on the right, the robot must have detoured left, so...
  for(int i = 0 ; i < 1.6*t ; i++)
    turn_right(); // Turn right so that the robot should aim where the robot pointed originally

cout<<"re-align stage 1 finished"<<endl;
```

공을 회피하느라 방향을 틀었으니, 다시 원래 목표로 이동하기 위해 방향을 바꿔 준다. 경험적으로 1.6t만큼 반대 방향으로 회전을 수행하면 원래의 목표물을 향하도록 함을 알았기에, 이를 수행한다.

```
ros::spinOnce();
if (blue_number == 0) { // If no blue ball have been found when trying to re-align...
    cout << "No blue ball found!" << endl;
    while (blue_number == 0) {
        move_backward(); // Move backward until any blue ball is visible
    }
}
```

만일 현재 시야에서 파란 공이 전혀 보이지 않을 경우, 하나라도 보일 때까지 후진한다.

```
// Re-align with the blue ball which the robot is aiming now
int wanted = 0;
ros::spinOnce();
if(check == 1) // If the checker method is on...
   wanted = near_blue; // Target the nearest blue ball
else // Else... (when the checker method is not activated)
   wanted = rightmost_blue; // Target the rightmost blue ball
```

이제, 목표로 삼을 파란 공을 지정한다. 이는 check 변수의 값에 따른다.

```
  // Precisely align with the targeted blue ball (within 30mm from the robot's y-axis)
  if(blue_X[wanted] > 30){ // If the aimed blue ball is on the right side of the robot... (farther than 30mm in x-pos)
    while(blue_X[wanted]> 30){
      wanted = 0;
      if(check == 1){
         wanted = near_blue;
      }
      else{
         wanted = rightmost_blue;
      }
      cout<<"align2 - blue_X : "<<blue_X[wanted]<<endl;
      turn_right_slow(); // Turn right slowly so that the aimed blue ball should be kept tracked
    }
  }
  else if(blue_X[wanted]< -30){ ... }
  cout<<"re-align stage 2 finished"<<endl;
}
```

목표로 삼은 파란 공이 로봇의 축을 기준으로 x < 30[mm] 이내로 들어올 때까지 로봇을 천천히 회전해 방향을 나란히 한다. 결과적으로, 로봇의 방향은 파란 공을 향하게 된다.

2.3.4 pick_up()

```
void pick_up(){
  ROS_INFO("Start picking up \n");
  ros::Duration(0.1).sleep();
  ros::spinOnce();
  cout<<"Ball x coordinate(Before align) : "<<blue_X[near_blue]<<"mm"<<endl;
  if(blue_X[near_blue]< -18){ // If the nearest blue ball (the ball should be picked up) is on the left from the pickupable range...
    while(blue_X[near_blue]< -18){
      turn_left_slow(); // Turn left slowly so that the robot and the ball should be aligned
    }
  }
  else if(blue_X[near_blue]> 20){ // If the nearest blue ball (the ball should be picked up) is on the right from the pickupable range...
    while(blue_X[near_blue]> 20){
      turn_right_slow(); // Turn right slowly so that the robot and the ball should be aligned
    }
  }
  else{
    cout<<"No problem."<<endl;
    }

 cout<<"aligning in pick-up stage is finished."<<endl;
 cout<<"Ball x coordinate(After align) : "<<blue_X[near_blue]<<"mm"<<endl;
```

파란 공을 줍기 위해 실행하는 함수다. 목표로 삼은 파란 공이 로봇과 나란히 정렬되도록(즉, blue_X[near_blue]→0) 천천히 회전하며 방향을 맞춰 준다.

```
// Put upward the wing for 32/40s
 for(int i = 0;i<32;i++){
   dataInit();
   data[10] = 1; // Command to raise up the wing
   write(c_socket,data,sizeof(data)); // Send the command
   ros::Duration(0.025).sleep();
 }
```

로봇의 pickup module이 들어올려져 있어야 공을 주울 수 있다. 32/40초만큼 이를 들어올린다.

```
//Go straight for 30/40s
for(int i = 0;i<30;i++){
  dataInit();
  data[17] = 1; // Command to move forward
  data[10] = 1; // Command to raise up the wing
  write(c_socket,data,sizeof(data));
  ros::Duration(0.025).sleep();
}
```

30/40초만큼 직진함으로서, 공을 충분히 주울 수 있는 거리 범위 내에 로봇을 위치시킨다.

```
  cout<<"Going straight by time_based control is finished. Now pick up!"<<endl;
  dataInit();
  data[20] = 1; // Command to pick up
  write(c_socket, data, sizeof(data));
  ros::Duration(1).sleep(); // Wait for wing to go back to its original position.
}
```

이제 pick up 명령을 준다. 공이 들어올려진 뒤, pickup module은 원래 위치로 되돌아간다.

2.3.5 drop_balls()

```
void drop_balls(){
  ros::spinOnce();
  for(int i =0;i<26;i++)
    go_straight(); // Go straight for 26 steps
  }
```

26/40초만큼 직진해서, 공을 안전하게 바구니에 놓을 수 있는 거리 안에 로봇을 위치시킨다.

```
  for(int i =0;i<t*40;i++){ // Raise up the storage for 40/40s.
    dataInit();
    data[21] = 1; // Command to drop balls
    write(c_socket, data, sizeof(data));
    ros::Duration(0.025).sleep();
  }
}
```

로봇이 공을 놓도록 drop module을 들어 올리는 모터 제어 명령을 준다. 이 결과, 파란 공을 담은 storage는 바구니 쪽으로 기울어지고 공은 자연스레 흘러내려가 바구니에 담긴다.

2.4. 3~~차 함수~~단계 함수 - Toward blue ball & Return to base~~- Toward blue ball & Return to base -~~

2.4.1. ~~Overall Explanation~~설명

3단계~~차~~ 함수는 Toward blue ball과 Return to base 함수로 이루어져있다. Toward blue ball은 3 rule operation에 따라 파란공을 줍는 미션을 수행하는 함수이다. Toward blue ball이 세 번 이루어져 파란 공을 모두 pick up했다고 판단하면 Return to base 함수가 실행되는데, 이는 RPLIDAR와 RGBD 카메라로부터의 시각 정보를 모두 이용해 로봇이 최적의 경로를 거쳐 반환점으로 돌아갈 수 있도록 한다. Toward blue ball은 RPLIDAR를 사용하지 않고 vision feedback으로만 이루어지며, Return to base는 처음에 로봇의 위치를 추정하고, 방향을 골대 쪽으로 향하는 데 RPLIDAR를 사용하고 골대 앞의 초록 공이 보이는 시점부터는 vision feedback을 이용한다.

2.4.2. Toward_blueballs()

Toward_blueballs의 기본적인 알고리즘은 blue ball이 detect되면, 예외상황 3개를 제외하고, target blue ball이 일정거리 이내로 들어올 때까지 blue ball을 향해 직진하는 것이다. 각각의 예외 상황에서는 직진 이외의 움직임을 주어 예외 상황을 처리하고, 다시 공을 향해 직진한다. 코드를 자세히 분석하면 다음과 같다.

2.4.2.1 Initialization

가장 먼저, Initialization을 진행한다. Blue ball과의 거리를 now_distance, target으로 하는 blue ball의 번호를 wanted라는 variable로 선언한다. Checker가 발동된 경우 target ball을 가장 가까운 파란 공으로 변경하고, 아닐 경우 rightmost ball을 target으로 한다.

서식 있음: 줄 간격: 배수 1.15 줄

서식 있음: 줄 간격: 배수 1.15 줄

서식 있음: 가운데, 줄 간격: 배수 1.15 줄

서식 있음: 줄 간격: 배수 1.15 줄

2.4.2.2 First exception - No blue ball detected

```
if(blue_number == 0){ //When no blue ball is found. (at the moment)
  cout<<"No blue ball found!"<<endl;
  while(blue_number != 0){
    move_backward();
  }
  ros::Duration(0.2).sleep(); // move backward for 0.2 second until blue ball is detected
  ros::spinOnce();
  return 0;
}
```

서식 있음: 가운데, 줄 간격: 배수 1.15 줄

첫 번째 예외는 순간적으로 blue ball이 detecting 되지 않은 경우이다. 주로 카메라에서 받아오는 정보가 오류가 있었을 때 나타나는 현상이다. 이 경우에는 다시 파란 공이 보일 때까지 뒤로 움직인 후 toward_blueballs 함수를 빠져나온다. Toward_blueballs함수가 0을 반환하는 경우에는 원하는 공을 쥽지 못한 경우로 판단을 하고 다시 toward_blueball을 호출한다. 이는 2.5에서 다시 설명할 예정이다.

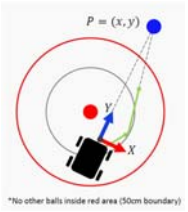
서식 있음: 줄 간격: 배수 1.15 줄

2.4.2.3 Second exception - Red ball blocking the path

```
if(check == 1) //update "wanted" variable
  wanted = near_blue;// if an exception happens,change the priority to near ball. exception happens when another near blue ball is in route
else
  wanted = rightmost_blue;// if no exception, target the rightmost ball

if(red_number!=0 && red_distance[near_red] <= blue_distance[wanted] &&
  red_Y[near_red] < 520 && abs(red_X[near_red]) < 185 && red_Y[near_red] > 0 ){ // if red ball is detected and blocks the path (threshold is 18.5cm in x-axis and 52cm in y-axis)
  cout<<"Avoid red ball!"<<endl;
  cout<<"red ball's distance is"<<red_distance[near_red]<<endl;
  cout<<"red ball X , Y coordinate is :"<<red_X[near_red]<<"and "<<red_Y[near_red]<<endl;
  detour_red();// during going to the target blue ball, detour detected red ball
}
else{
  go_straight();// no red ball blocks the path, go straight
}
```

```
if(now_distance != blue_distance[wanted]){
  now_distance = blue_distance[wanted]; // as vehicle moves, continuously define "now_distance" variable to distance of target blue ball
  cout<<"Targeting blue ball Y coordinate is : "<<blue_Y[wanted]<<"mm"<<endl;
}
ros::spinOnce();
}
```



두 번째 예외 상황은 target blue ball을 향해 가던 중 red ball이 경로에 있을 경우이다. Red ball이 경로를 막고 있을 경우, 2차 함수인 detour_red 함수를 실행하여 빨간 공을 피한다. 빨간 공을 피하는 예외가 없을 경우에는 직진하는 명령(go_straight)을 지속적으로 준다.

← 그림 5 두 번째 예외 상황. 빨간 공이 이동 경로를 막고 있다.

서식 있음: 줄 간격: 배수 1.15 줄

2.4.2.4 Third exception - Strange distance feedback occur

```
if(blue_Y[wanted]<0){//if blue ball distance is minus, which is strange case
  cout<<"Y coordinate is very strange"<<endl;
  while(blue_Y[wanted]<0){
    move_backward();
  }
  ros::Duration(0.2).sleep(); // go backward to get new information
  ros::spinOnce();
  return 0;//Y coordinate strange! Retry required
}
```

서식 있음: 가운데, 줄 간격: 배수 1.15 줄

Blue ball의 거리가 음의 값으로 튀는 경우가 매우 적은 빈도로 발생하는데, 이에 대한 예외 처리문이다. 첫 번째 예외상황과 동일한 방법으로 처리한다.

서식 있음: 줄 간격: 배수 1.15 줄

2.4.2.5 Blue ball is close, then Pick up

```
    ros::Duration(0.2).sleep();
    ros::spinOnce();
    cout<<"Targeting blue ball's X, Y coordinate is : ["<<blue_X[wanted]<<" , "<<blue_Y[wanted]<<"]"<<endl;
    cout<<"Targeting blue ball's distance is :"<<blue_distance[wanted]<<endl;

    int pickup_distance = static_cast<double>(blue_Y[near_blue]-30)/1000; // if target blue ball is near to pick up
    pick_up(pickup_distance); // try to pick up
    return 1;//successfully got a blue ball.
}
```

예외를 처리하면서 직진하다가, 원하는 blue ball과의 거리가 충분히 가까워지면 2차함수인 pick up() 함수를 실행하면서 공을 줍게 된다. 공을 성공적으로 주우면 toward_blueballs는 1을 반환한다. 1을 반환하는 경우에는 경기장에 남아있는 파란 공의 개수가 1개 줄어든 것으로 판단하고 다른 toward_blueballs 함수를 호출하여 다음 target blue ball을 줍는다.

2.4.3 Return and drop

2.4.3.1 ~~Overall explanation~~설명

돌아오는 과정에서 가장 큰 문제는 모든 파란 공을 주운 시점에서 로봇이 goal이 어디에 있는지 알지 못한다는 점이었다. 심지어 카메라의 감지 범위 바깥에 초록색 공이 위치하는 경우도 있었다. 따라서 우리는 RPLIDAR를 이용해 goal position을 찾기로 하였다.

모든 과정은 RPLIDAR로 네 코너를 감지하는 것으로 시작한다. 후술할 방법으로 코너를 감지한 이후에 각각의 range정보와 angle 정보를 저장한다. 그리고 그 중 가장 멀리 떨어진 코너를 중심으로 양 옆의 다른 코너 정보를 사용하여 양 변의 길이를 비교한다. 자명하게도, 짧은 길이의 변이 goal이 있는 위치다(그림 6 참조). 자세한 알고리즘은 아래에서 설명한다.

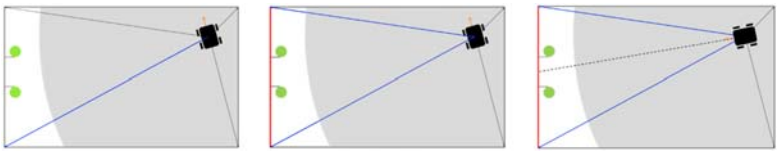


그림 ~~6~~ 6 로봇으로부터 각 꼭지점의 거리 정보를 이용해 goal이 있는 지점을 탐색하는 알고리즘.

2.4.3.2 Find corners

```
//This function find four corners and store degree and distance value from -180 degree to 180 degree
void find_corners(){
  ros::spinOnce();
  int corner_num = 0;

  for(int i=0 ; i<lidar_size ; i++){
    if(isinf(lidar_distance[i])) continue;//if corresponding distance value is null value, lidar returns infinite value, so we need to neglect such value.
    float differenceSum = 0;//to check the corresponding angle is corner angle. we need to sum up differences
    for(int j = i-25;j<i+25;j++){//check for +-25 degree to seek corners
      if(isinf(lidar_distance[j])){//if corresponding distance value is null value, lidar returns infinite value, so we need to neglect such value.
        continue;
      }
      int k = j;
      if(j<0) k = lidar_size+j;//if the value is smaller than 0, its real value is 180 - value.
      if(j>lidar_size) k = j-lidar_size;//vice versa
      differenceSum = differenceSum + 1-lidar_distance[k]/lidar_distance[i];//Sum up all differences
    }
    if(differenceSum>1.0){//if the difference sum is larger than certain value, it means there's corner like region.
      corners[corner_num] = i;
      //cout<<"corner"<<corner_num+1<<" angle is "<<lidar_degree[corners[corner_num]]<<endl;
      //cout<<"corner"<<corner_num+1<<" distance is "<<lidar_distance[corners[corner_num]]<<endl;
      corner_num++;
      i = i+20;//if corner is detected, it means there's no corner in +-20 degree.
    }
    if(corner_num == 4) {//there's only four corners, so we need to stop
      //cout<<"Now we have four corners"<<endl;
      break;
    }
  }
}
```

처음에 코너를 감지하기 위해서는 단순히 range정보 상에서 local maxima를 찾으면 된다고 생각했었다. 그러나 실제로는 벽에 약간의 요철이 있었고 그것들을 local maxima로 읽어 들여

서식 있음: 가운데, 줄 간격: 배수 1.15 줄

서식 있음: 줄 간격: 배수 1.15 줄

서식 있음: 가운데, 줄 간격: 배수 1.15 줄

서식 있음: 줄 간격: 배수 1.15 줄

서식 있음: 가운데, 줄 간격: 배수 1.15 줄

서식 있음: 줄 간격: 배수 1.15 줄

서식 있음: 글꼴: (영어) 바탕체, (한글) 바탕체

서식 있음: 글꼴: (영어) 바탕체, (한글) 바탕체

서식 있음: 글꼴: (영어) 바탕체, (한글) 바탕체

서식 있음: 글꼴: (영어) 바탕체, (한글) 바탕체

서식 있음: 글꼴: (영어) 바탕체, (한글) 바탕체

서식 있음: 글꼴: (영어) 바탕체, (한글) 바탕체

코너로 감지하는 문제가 존재했다. 새로운 알고리즘을 구성하기 위해 학술 논문 자료[1]를 참고하였다. i번째 각도에서의 range 값과 그 부근 각도(i-25번째 ~ i+25번째)에서의 range값의 차이들을 다 더한 후, 그 합이 일정한 값(1m)보다 크면 i번째 각도가 바로 corner라는 사실을 알게 되었다. 예를 들어, (40도에서의 range값 - 25도에서의 range값) +(40도에서의 range값 - 26도에서의 range값) + …. + (40도에서의 range값 - 65도에서의 range값) 이 1m보다 크다면 코너는 40도 지점에 있는 것이다.

따라서 i번째 각도의 range 정보와 근처 각도에서의 range 정보들 차이들을 모두 더해서 그 합이 일정 값(1m)보다 클 때, 그 지점을 코너로 판단하였다. 한번 특정 각도가 코너라고 판단했으면, 중복을 피하기 위해 그 다음 20개 각도에 대해서는 연산을 생략했다. 이러한 방식으로 실제 시험장에서 코너를 감지해 보았을 때, 모든 경우에서 네 코너의 위치를 정확하게 구할 수 있었다.

```
//calculate square of sides and substract each other to compare which one is smaller.
float compare = pow(partner1_distance,2.0)-pow(partner2_distance,2.0)-2*farthest_distance*(partner1_distance*cos(angle_partner1)-partner2_distance*cos(angle_partner2));
int real_partner;
if(compare<0){//if compare value is less than zero, the leftside partner is real partner
  real_partner = partner1;
}
else{
  real_partner = partner2;
}
float ans =  (lidar_degree[corners[farthest]]+lidar_degree[corners[real_partner]])/2;//value to return

//if answer value is extreme cases, we need to calibrate.
if(abs(lidar_degree[corners[farthest]])+abs(lidar_degree[corners[real_partner]]) > 180 && lidar_degree[corners[farthest]]*lidar_degree[corners[real_partner]]<0){
  if(ans<0) return 180+ans;
  else return -180+ans;
}
else{
  return ans;
}
}
```

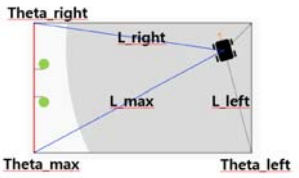


그림 7 각 코너로 얻는 변수 정보

4개의 코너가 있는 각도를 구했으므로 4개의 코너까지의 range정보를 구할 수 있으며, 가장 range가 큰 corner를 특정할 수 있다. 가장 멀리있는 corner의 각도를 theta_max, 그 각도에서의 range값을 L_max라고 하자. 그리고 theta_max의 좌우에 존재하는 corner의 각도를 각각 theta_left, theta_right라고 하고, 그 각도에서의 range 정보를 각각 L_left, L_right라고 하자. 로봇의 위치를 꼭짓점으로 생각하고 구한 3개의 corner를 점들로 생각하면, 점 4개에 대해서 기하학적으로 삼각형이 2개 그려진다. 2개의 삼각형에 대해서 두 변의 길이(예를 들어 L_max, L_right)와 끼인 각(예를 들어 |theta_max - theta_right|)을 알고 있으므로 나머지 한 변의 길이를 제2코사인법칙을 이용해 알 수 있다. 삼각형이 2개이므로 구하는 변이 2개일 것이다. 이 때 구한 나머지 한 변의 길이 중에서 길이가 짧은 변에 goal이 존재한다고 볼 수 있다.

[1] Liu, Shifei, et al. "Adaptive covariance estimation method for LiDAR-aided multi-sensor integrated navigation systems." Micromachines 6.2 (2015): 196-215.Ref) Liu, Shifei & Atia, Mohamed & Gao, Yanbin & Noureldin, Aboelmagd. (2015). Adaptive Covariance Estimation Method for LiDAR-Aided Multi-Sensor Integrated Navigation Systems. Micromachines. 6

서식 있음: 가운데, 줄 간격: 배수 1.15 줄

서식 있음: 줄 간격: 배수 1.15 줄

서식 있음: 글꼴: (영어) 바탕체, (한글) 바탕체, 굵게 없음

서식 있음: 캡션, 줄 간격: 1줄

서식 있음: 글꼴: (영어) 바탕체, (한글) 바탕체, 굵게 없음

```
//This function let robot turn toward the goal post and, search for green balls.
void to_find_green(){
  while(find_goal()> 10 || find_goal()< -10){//Until the angle between front and goal is less than 10 degree, turns.
    if(find_goal()> 0){
      turn_left();
    }
    else{
      turn_right();
    }
  }
  ros::spinOnce();

  int t =0;
  //Go straight to see two green ball.
  while(green_number != 2){
    go_straight();
    t++;
    if(t>100){//if camera can't find green balls for 2.5 sec, it means the robot miscalculated the goal position, so start it again.
      to_find_green();
      return;
    }
  }
  for(int i = 0;i<40;i++){//go straight for 1 sec (To assure seeing 2 green balls)
    go_straight();
  }

  dataInit();
  write(c_socket, data, sizeof(data));
  ros::Duration(0.5).sleep();//for drop out dragged red balls by pickup module.
  cout<<"Webcam sees "<<green_number<<" green balls now"<<endl;//To ensure in testing scenario.
}
```

서식 있음: 가운데, 줄 간격: 배수 1.15 줄

이렇게 goal이 존재하는 각도를 구해서 회전을 마치면, 이후에는 두 개의 초록색 공을 카메라로 감지할 때 까지 직진해 준다. 감지거리의 boundary에서는 카메라가 받아들이는 값들이 진동하기 때문에 1초 더 직진해서 초록 공이 확실하게 보이도록 했다.

서식 있음: 줄 간격: 배수 1.15 줄

2.4.3.3 Toward the goal and drop balls.

```
//Function for final drop out.
void toward_goal(){
  //align to rightmost green ball since our drop module is on leftside. It is ok to align toward right ball.
  //At first, we don't need to align too precise.
  if(green_X[rightmost_green] > 70){
    ros::spinOnce();
    while(green_X[rightmost_green]> 70 ){
      turn_right_slow();
    }
  }
  else if(green_X[rightmost_green]< -100 ){
    ros::spinOnce();
    while(green_X[rightmost_green]< -100 ){
      turn_left_slow();
    }
  }

  //Go toward rightmost green until distance is 540mm
  ros::spinOnce();
  while( green_Y[rightmost_green] > 540){
    dataInit();
    ros::spinOnce();
    if(green_number == 0){ //When no green ball is found. (at the moment)
       ros::Duration(0.025).sleep();
            ros::spinOnce();
    }
    go_straight();
  }
```

서식 있음: 가운데, 줄 간격: 배수 1.15 줄

```
  //align strictly and go straight robot need to see slightly left of green ball.
  if(green_X[rightmost_green]< 30){
    while(green_X[rightmost_green]< 30){
      turn_left_slow();
    }
  }
  else if(green_X[rightmost_green] > 60){
    while(green_X[rightmost_green] > 60){
      turn_right_slow();
    }
  }
  else{
     cout<<"No problem."<<endl;
  }
  ros::spinOnce();

  //Go straight for 10/40 sec and drop off for 3 sec to ensure all the blue balls dropped out.
  drop_balls(3);
}
```

마지막으로 남은 과정은 goal 지점에 도착하고 파란 공을 내려 놓는 것이다. 우리의 drop module이 차체의 왼쪽에 붙어 있기 때문에 두 초록 공 중 오른쪽에 있는 공을 향해서 정렬한다면 drop module이 정확히 바구니를 향했다. 따라서 모든 직진 및 정렬 과정을 오른쪽 초록색 공을 기준으로 진행하였다.
초록 공이 멀리 있는 경우에는 먼저 threshold를 크게 잡아서 rough하게 정렬했고, 직진하면서 초록 공과 가까워지면(공의 y좌표가 54cm 이하) 다시 느린 회전으로 정밀하게 정렬해 주었다.
가장 마지막으로 시간 기반 모터제어를 이용해서 일정한 거리만큼 직진한 다음에 drop 신호를 보내주었다. 이 때 직진하는 시간은 경험적으로 정했다.

서식 있음: 줄 간격: 배수 1.15 줄

2.5 Start Path finding

이 함수는 앞서 말한 모든 과정을 통합하고, blue ball을 pick up하는 순서를 정해준다. Pick up이 끝난 이후에는 차례대로 to_find_green과 toward_goal을 호출해 준다.

```
void start_path_finding(){ //Final function, integrate everything and select order of blue balls.
  go_straight_lidar(1);//Need to go 1m before starting. To ensure every blue balls are detected.
  int i = 0;
  while(i<3){//Collect three balls.
    ros::spinOnce();

    //When we can't find blue balls. turn CCW to find blue ball.
    while(blue_number==0){
      turn_left();
    }

    //align to rightmost blue ball within 10cm boundaries.
    if(blue_X[rightmost_blue] > 100){
      ros::spinOnce();
      cout<<blue_X[rightmost_blue]<<endl;
      while(blue_X[rightmost_blue]> 100 )
        turn_right();
    }
    else if(blue_X[rightmost_blue]< -100 ){
        ros::spinOnce();
        cout<<blue_X[rightmost_blue]<<endl;
        while(blue_X[rightmost_blue]< -100 ){
          turn_left();
      }
    }
```

서식 있음: 가운데, 줄 간격: 배수 1.15 줄

가장 먼저, 시작하는 지점에서는 모든 파랑 공이 보이지 않을 수도 있기 때문에 RPLIDAR로 피드백 제어하며 1m 전진한다. 그 이후로 파란 공을 차례대로 pick up하는 과정으로 돌입한다. 기본적으로 가장 오른쪽에 있는 공으로 접근하면서 시계 반대방향으로 진행하기 때문에 가장

서식 있음: 줄 간격: 배수 1.15 줄

오른쪽의 공으로 정렬해 준다.

```
//Consider checker-trigger situation && toward_blueball starts!.
check = checker();
```

```
int checker(){//Use if near ball is close enough and return 1 if it's close enough
  ros::spinOnce();
  int check1 = 0;
  cout<<"rightmost = "<<rightmost_blue<<", near blue = "<<near_blue<<", near blue_x = "<<blue_X[near_blue]<<endl;
  if(near_blue != rightmost_blue && blue_X[near_blue]-500 > 0 ){//if near blue ball is in 500mm distance in x coordinate, check triggered!
    check1 = 1;
    cout<<"change priority"<<endl;
  }
  else{//Or checker is not triggered and keep going forward.
    cout<<"Keep forward"<<endl;
  }
  return check1;//return 0 or 1
}
```

그 다음으로 checker 함수를 호출 해 준다. Checker 함수는 가장 가까운 파란 공이 가장 오른쪽에 있는 공과 다른 경우를 처리하는 함수이며, 앞서 말한 3 rule operation에서 유일하게 처리하는 예외 케이스이다. 가장 가까운 공의 x좌표가 50cm 이내인 경우에만 1을 반환하고 아닌 경우에 0을 반환한다. 1을 반환하는 경우에 가까운 공을 target으로 설정하고, 0인 경우에는 원래대로 가장 오른쪽에 있는 공을 target으로 설정한다.

```
if(check == 1){//if checker triggered
  int t = 0;
  while(blue_X[near_blue]<-60){//align toward the nearest ball. which means left ball.
    turn_left_slow();
    t++;
  }
  if(i>0) move_backward(0); //Move backward for a short time for no red ball detection. for safety!
  if(toward_blueballs() == 1){ //For extreme case such as blue ball no found!, toward_blueballs start!
    i++;
  }
  else{
    cout<<"Toward blue failed with unknown error! Retry"<<endl;//Retry i-th trial, not happening many times
    continue;
  }
  ros::spinOnce();
  for(int i=0;i<t;i++){//After picking up nearest ball, try to re-align. before that, camera need to detect blue ball.
    turn_right_slow();
  }
  while(blue_X[rightmost_blue]>60){//Align to the rightmost ball again.
    turn_right_slow();
  }
  check = 0;
}
```

checker 함수에서 1이 반환된 경우에 왼쪽에 있는 가장 가까운 공으로 정렬하기 위해 좌측으로 회전한다. 그리고 toward_blueballs 함수를 호출하여 i번째 공을 줍는 시도를 한다. 성공적으로 공을 주웠다고 판단하면 i값을 증가하면서 다음 공을 줍는다. 예외 상황이 발행하는 경우에는 toward_blueballs 함수가 0을 반환하며 빠져나오기 때문에 i값이 증가하지 않는다. 따라서 다시 start_pathfinding함수의 맨 위로 올라가서 i번째 공을 pickup 하는 과정을 실행한다. Toward_blueballs가 성공적으로 1을 반환하면 다시 가장 오른쪽에 있는 공을 향해 회전을 해야 한다. 이 때 확실하게 가장 오른쪽에 있는 공을 보기 위해서 처음에 왼쪽으로 회전한 시간만큼(t) 다시 우측으로 회전한다. 이후에는 가장 오른쪽에 있는 공의 x좌표가 60cm 보다 작아질 때까지 정렬한다. 이 방법을 이용하면서 가까이 있는 공을 먼저 주울 수 있었기에 경제적인 움직임이 가능했다.

서식 있음: 가운데, 줄 간격: 배수 1.15 줄

서식 있음: 줄 간격: 배수 1.15 줄

서식 있음: 가운데, 줄 간격: 배수 1.15 줄

서식 있음: 줄 간격: 배수 1.15 줄

```
else{//If checker is not triggered, just start toward blueballs.
  cout<<"Toward blue ball starts!"<<endl;
  if(i>0) move_backward(0); //Move backward for a short time for no red ball detection.
  if(toward_blueballs() == 1){ //For extreme case such as blue ball no found!
    i++;
  }
  else{
    cout<<"Toward blue failed with unknown error! Retry"<<endl;//Retry i-th trial, not happening many times
    continue;
  }
}
```

서식 있음: 가운데, 줄 간격: 배수 1.15 줄

서식 있음: 줄 간격: 배수 1.15 줄

가장 가까운 공과 가장 오른쪽에 있는 공이 같은 경우에는(check==0인 경우) 단순히 toward_blueballs함수를 호출한다. 정상적으로 파란 공을 주워서 toward_blueballs가 1을 반환하면 i값이 증가하면서 다음 파란 공을 줍도록 한다. 예외상황으로 인해 toward_blueballs이 0을 반환하는 경우가 생기면 i값을 증가시키지 않고 다시 toward_blueballs함수를 실행한다. 예외 상황은 2.4에서 설명했다. 파란색 공 세 개를 성공적으로 주웠다면 약간의 후진을 해서 goal이 위치하는 벽으로부터 좀 더 멀어지도록 한다. goal이 위치한 벽을 찾는 것은 가장 먼 corner를 찾는 것부터 시작하기 때문에, goal이 있는 벽으로부터 멀어지는 것은 되돌아오는 과정을 더 확실하게 해주는 작업이다.

```
dataInit();
write(c_socket,data,sizeof(data)); //Data Initialize for stop moving
move_backward(0); //Move backward for a short time. In case of too close to the middle position.
to_find_green();
toward_goal();
```

서식 있음: 가운데, 줄 간격: 배수 1.15 줄

서식 있음: 줄 간격: 배수 1.15 줄

이후에는 goal의 위치를 찾고 두 개의 초록 공을 관측할 때까지 접근하는 to_find_green함수를 호출한다. 초록 공이 잘 보이는 상태가 되면 최종적으로 goal position에 접근하고 모든 파란 공을 바구니 안에 떨어뜨리는 toward_goal함수를 호출하면서 미션을 완수한다.

3. 결론

3.1 전체적인 의견

우리 조의 코드는 전체적으로 기본에 충실하게 안정적으로 운영한다. 어떠한 공 배치에 대해서도 미션을 성공할 수 있도록 하겠다는 우리 조의 철학처럼, 실제로 테스트를 했을 때 난이도가 매우 높은 배치에 대해서도 안정적으로 미션을 완수할 수 있었다. 특히 만족도가 높았던 코드는 toward_blueballs()함수와 to_find_green()함수이다. 왜냐하면 파란 공을 주울 때 발생할 수 있는 예외 상황은 매우 많은데, 그 모든 상황들을 각각의 예외문으로 처리하지 않고 좀 더 일반적이면서 포괄적인 해결책을 사용하면서 코드를 간략하게 짰기 때문이다. 또한 공 3개를 다 줍고 다시 바구니로 돌아오는 과정에서 RPLIDAR를 이용한 알고리즘을 이용하면서 불필요한 움직임을 최소화할 수 있었기 때문이다.

다만 아쉬운 점도 분명히 존재하는데, 가장 아쉬운 점은 센서의 개수와 관련한 부분이다. 처음부터 최소한의 센서 개수를 사용하자고 합의를 하였기 때문에 카메라를 1개, RPLIDAR를 1개씩만 사용했다. 하나의 카메라로 매우 가까운 곳과 매우 먼 곳을 동시에 보는 것은 힘든 일이었기 때문에 다소 불리한 점이 있었다. 물론 카메라 1개로도 좋은 퍼포먼스를 보여줄 수 있었지만, 좀 더 강건하면서도 안정적인 결과를 위해서 보조 카메라 1개를 더 사용하는 것이 더 좋지 않았을까 하는 아쉬움이 있다.

3.2 Desired Coding schedule

이상적인 스케줄은 최소 10주차까지는 코드를 완성하는 것이다. 10주차는 2차 발표를 끝냈거나 준비하는 시기이다. 이 시기에 모든 코드를 완성하면 최대 4주 정도 테스트를 해볼 수 있다. 테스트를 하는 기간이 길어야 하는 이유는 크게 3가지이다.

첫째, 디버깅에 여유가 생긴다. 아무리 코드를 열심히 짜도 실제 기계 없이 코딩을 하다 보면 예상치 못한 오류가 발생한다. 예를 들어, 로봇이 이상적으로 직진하고 회전한다고 가정하고 코드를 짰을 경우, 실제 기계에서 생기는 관성 및 마찰력으로 인해 이상적인 퍼포먼스가

불가능한 경우에 대해서 대처할 수 없다. 관성은 기계들마다 다르기 때문에 다른 조 및 선배들의 결과로부터 참고를 할 수 없다. 또한 모터의 각속도에 따라서 관성이 달라지기 때문에 값을 통일할 수가 없다. 하드웨어 및 랩뷰 파트가 완성되는 대로 바로 테스트할 수 있도록 ROS파트는 최대한 빨리 초안을 완성해서 대기하고 있어야 한다.

둘째, 상대적으로 데모 경기장의 경쟁이 적다. 최종 데모 2주 전부터는 테스트를 위해서 경기장을 사용하려고 해도 다른 조들이 쓰고 있기 때문에 사용하지 못하거나 늦게 사용하는 경우가 허다하다. 따라서 다른 조들보다 미리 테스트를 해볼 수 있는 것은 큰 도움이 된다.

셋째, 도전적인 코딩을 할 수 있는 여유가 생긴다. 테스트할 수 있는 시간이 많아서 관성 및 마찰력, 또는 실제 상황에서 생기는 문제들을 해결했다면 좀 더 도전적인 코딩을 진행할 수 있다. 단순히 회전을 하지 않고 PD제어를 해보면서 시간을 단축할 수도 있고, 불안정하지만 시간이 적게 걸리는 알고리즘도 테스트해볼 수 있다. 창의적 시스템 구현 과목은 데모에서의 성공이 최우선시 되기 때문에 시간적인 여유가 없다면 도전적인 코딩이 제한되는 경향이 있다. 따라서 도전적이고 창의적인 알고리즘 구현을 위해 테스트 기간을 최대한 길게 확보해야 한다.

G조의 경우에는 대략 4주차부터 기본적인 뼈대를 만들기 시작했고, 7주차까지 파란 공을 1개 줍는 함수를 완성했다. 9주차까지는 현재 코드의 초안이 되는 1차 완성 코드를 완성했고, 10주차부터는 1차 완성 코드의 parameter값들을 손보는 작업을 진행했다. 덕분에 조금은 여유를 갖고 테스트를 해볼 수 있었고, 9개 조 가운데 유일하게 추가적인 공동강의실 새벽 예약 없이 프로젝트를 완수할 수 있었다. 남은 3주동안 예상치 못한 문제들을 발견하여 디버깅하는데 시간이 많이 쓰였지만, 앞서 말한 우리 조의 함수 modularization 철학 덕분에 비교적 빠른 디버깅이 가능했다. 따라서 최소 10주차 까지는 모든 코드를 완성하도록 스케줄을 잡는 것이 좋다고 판단한다.

서식 있음: 단락의 첫 줄이나 마지막 줄 분리 허용, 한글과 영어 간격을 자동으로 조절하지 않음, 한글과 숫자 간격을 자동으로 조절하지 않음