FINAL REPORT

ME492 Mobile System Programming

C 조

20140899 황성원 | 20160530 이창준 | 20150117 김동하

KAIST Department of Mechanical Engineering

2019.06.16

4 ROS

4.1.1 Capston Design Project Mission

벽으로 둘러싸인 3m x 5m 직사각형 모양의 지역이 존재한다. 그 안에, 3개의 파란공과 3개의 빨간공이 다음과 같은 규칙을 만족하면서 무작위로 놓인다.

1) 모든 공 사이의 거리는 최소 50cm이상 떨어져 있을 것

2) 모든 공과 벽은 최소 50cm이상 떨어져 있을 것

그리고, 한쪽 3m길이의 선분 쪽에는 바구니가 놓여있다. 그 바구니의 양쪽 끄트머리에는 각각 1개씩 초록색공이 붙어있다.

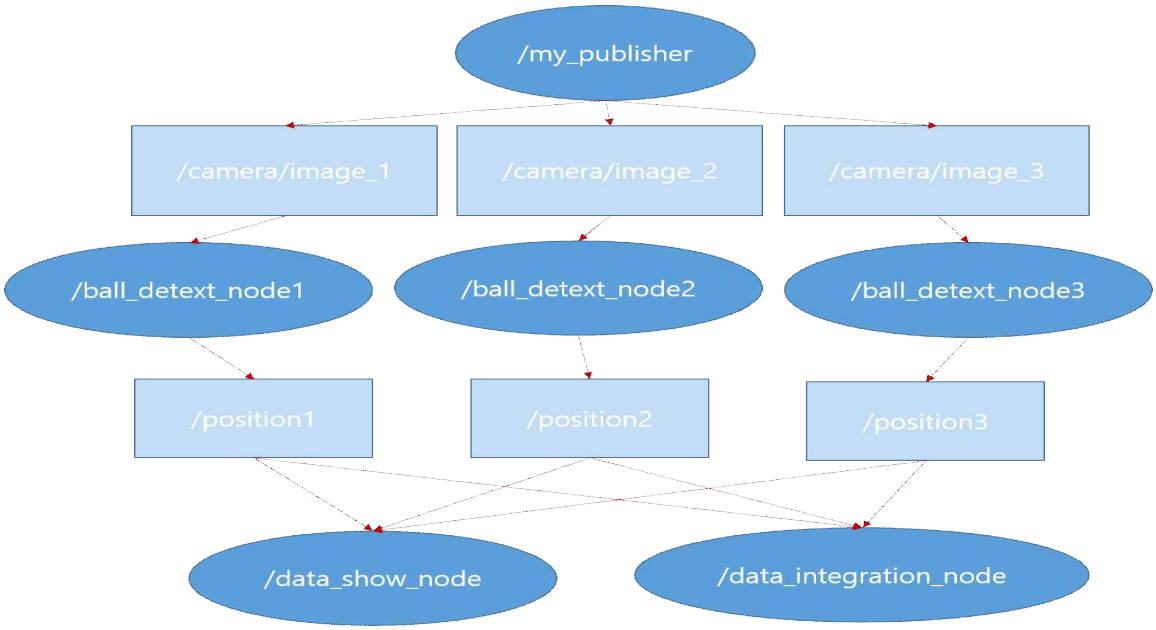
로봇을 바구니의 조금 앞에서 출발시켜, 작동하여 놓여있는 파란공과 빨간공들 중 파란공 3개만을 집어 바구니에 집어넣는 것이 Capstone Design의 궁극적인 목표이다.

4.1.2 Overall Os Structure

4.1.2.1 ROS(Robot Operating System)란?

로봇용 공개소스 메타 운영체제. 프로세스간 메시지 전달, 패키지 관리 기능들을 제공한다. 이외에도 여러 다양한 기능들이 있지만, 우리가 사용하는 주 기능은 node의 간의 상호작용을 통하여(publish, subscribe) 바퀴의 움직임에 명령을 주는 것이다. node는 어떠한 variable을 ros운영체제를 통하여 다른 node로 전달해 줄 수 있다. 예를 들어 node1에서 node2로 test라는 값을 옮기고 싶다고 하자. node1의 test값을 ros운영체제로 publish(전달)하고, ros운영체제에서 그 publish된 값을 node2에서 subscribe(수령)하는 것이 test값을 옮기는 방법 중 하나이다.

4.1.2.2 How our nodes connected (rqt-graph)



<Fig 4.1.1> 우리 로봇 운영 체제의 rqt graph

우리의 로봇은 3개의 카메라를 사용하기 때문에 /ball_detect_node가 3개 존재한다.

4.1.2.3 /my_publisher

/my_publisher는 webcam을 통하여 320*240 사이즈의 이미지를 캡쳐하고, 그 이미지를 /camera/image_1,2,3라는 이름의 메시지로 publish한다. 이때, Message Source는 Sensor_msgs::ImagePtr이다. 또한 보내는 메시지는 아래와 같이 구성되어 있다.

uint32 height	Image height(#rows)
uint32 width	Image width(#cols)
string encoding	Encoding of pixels
uint8 is_bigendian	Is data bigendian?
uint32 step	Full row length in bytes
uint8[] data	Actual image matrix data

<Fig 4.1.2> 카메라 데이터 메시지 정보

4.1.2.4 /ball_detect_node, relation with OpenCV

/my_publisher 노드에서 publish한 /camera/image 메시지 값을 subscribe한다. 이 메시지를 통해 OpenCV로 공들을 인지하고, 공들의 정보를 계산해 /position_1,2,3라는 이름의 메시지를 publish 한다. 이때 Message Source는 core_msgs::ball_position이고, 메시지는 아래와 같이 구성되어 있다.

int32 size	number of balls
float32[] img_x	x-direction distance to the ball
float32[] img_y	y-direction distance to the ball

<Fig 4.1.3> ball_detect 노드가 송출하는 메시지 정보

여기서 img_x, img_y 가 array형식으로 이루어져 있다는 점에서 어떤 공이 array의 0번째 자리로 올지 알 수 없었다. 그래서 아래와 같은 방식으로, 가장 가까운 공을 array의 0번째로 오게 설정해 두었다.

```
//Pick most large ball
if(contours_r.size() > 1){
    //Fill 0 index
    float max = 0;
    size_t max_i = 0;
    for(size_t i =0; i<contours_r.size();i++){
        if(radius_r[i] > max){
            max = radius_r[i];
            max_i = i;
        }
    }
    vector<Point2f>temp_r(1);
    temp_r[0] = center_r[max_i];

    radius_r[max_i] = radius_r[0];
    center_r[max_i] = center_r[0];

    radius_r[0] = max;
    center_r[0] = temp_r[0];
```

<Fig 4.1.4> ball_detect_node의 공 거리에 따른 데이터 순서

제일 큰 radius max와 그때의 array number max_i를 저장해두고, max_i번째 자리의 값과 0번째 자리의 값을 바꿔주었다.

4.1.2.5 /data_integration_node

/position_1,2,3를 subscribe하여, 그 데이터들을 토대로 data[24]의 값을 결정한다. data[24]은 후술할 labview에서 사용할 값으로, motor를 제어하는데 사용된다.

이 노드는 주로 학생들이 작성해야할 노드로서, 모터의 움직임을 어떻게 제어할 것인지 결정하는 주요 노드라 할 수 있다.

아래의 이미지는 각 data가 무엇을 의미하는지 나타낸다.

```
data[0] = 0; //lx*data[3];ey'
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);
data[9] = 0; //GamepadTriggerLength(_dev, TRIGGER_RIGHT);
data[10] = 0; //GamepadButtonDown(_dev, BUTTON_DPAD_UP);
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);
data[16] = 0; //GamepadButtonDown(_dev, BUTTON_X);
data[17] = 0; //GamepadButtonDown(_dev, BUTTON_Y);
data[18] = 0; //GamepadButtonDown(_dev, BUTTON_BACK);
data[19] = 0; //GamepadButtonDown(_dev, BUTTON_START);
data[20] = 0; //GamepadButtonDown(_dev, BUTTON_LEFT_SHOULDER);
data[21] = 0; //GamepadButtonDown(_dev, BUTTON_RIGHT_SHOULDER);
data[22] = 0; //GamepadButtonDown(_dev, BUTTON_LEFT_THUMB);
data[23] = 0; //GamepadButtonDown(_dev, BUTTON_RIGHT_THUMB);
```

<Fig 4.1.5> data_integrate 노드가 송출하는 data array의 element 정보

각각의 데이터 element에는 0~1 값을 대입하여 전송할 수 있으며, 그 값에 비례하여 모터 속도를 구현하게 된다.

이 구조를 기반으로, 실질적 행위 (직진, 좌직진, 시계방향 회전 등등...)를 함수로 정의해 그 함수를 이용해 data array에 필요한 값을 정의하도록 했다. 그 구조는 아래와 같다.

```
void robot_move_forward(float velocity)
{
  data[1] = velocity;
}
void robot_move_back(float velocity)
{
  data[1] = -velocity;
}
void robot_move_left(float velocity)
{
  data[4] = -velocity;
}
void robot_move_right(float velocity)
{
  data[4] = velocity;
}
```

```
void robot_rotate_CW(float velocity)
{
  data[0] = velocity;
}
void robot_rotate_CCW(float velocity)
{
  data[0] = -velocity;
}
void robot_pick_ball(float velocity)
{
  data[5] = velocity;
}
void robot_exit_ball()
{
  data[8] = 1;
}
```

<Fig 4.1.6> Operation의 직관적 사용을 위한 함수 구축

이렇게 매 loop마다 어떤 data array를 publish할 지를 결정한 후, 이 노드는 결정된 data 값을 myrio에서 읽을 수 있는 TCP-IP communication형식으로 바꾸어 송출하게 된다. 이때 반드시 같은 wifi connection, ip address, port number from myRio를 맞춰줘야 한다.

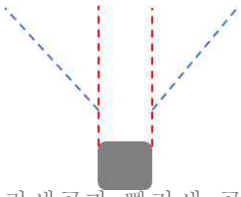
4.1.4 Overall Algorithm Philosophy

4.1.4.1 Path Planning

우리 로봇은 파란공 3개를 집기 전, 파란공 3개가 시야에 다 들어올 때에 path planning을 시작합니다. 먼저 1) 가장 가까운 공으로 가는 것을 결정하고 2) 두 번째 공은 첫 번째 공을 먹은 위치에서 가장 가까운 공을 먹는 것으로 설정하되 3) 두 번째 공을 먹기 위해서 첫 번째 공을 먹은 위치에서 CW으로 돌아야 할지 CCW으로 돌아야 할지를 기억합니다. 4) 세 번째 공도 마찬가지로 돌아야 할 방향을 기억합니다. 5) 세 번째 공을 먹은 후 초록색 공을 보기 위해 CW 으로 돌아야 더 빨리 볼지 CCW 로 돌아야 할지 계산한 후 기억합니다.

4.1.4.2 Universal blue-ball collecting algorithm

우리의 알고리즘은 아래의 파란색 점근선을 threshold로 하여 마치 우리 앞에 가상의 깔때기가 있는 것처럼 파란색 공을 먹습니다. Threshold를 벗어나려고 하면 바로 보정을 하여 우리가 먹으려는 파란색 공이 항상 threshold안에 있게 합니다. 이렇게 하는 경우에는, 파란색 공의 다양한 위치에 따른 케이스를 나누어 고려할 필요 없이, 하나의 알고리즘으로 다양한 위치의 파란색 공을 먹을 수 있습니다.



<Fig. 4.1.7> 파란색공과 빨간색 공을 위한 threshold

4.1.4.3 Universal red-ball detour algorithm

단순히 근거리 카메라에 빨간색 공이 보이고 그 빨간색 공이 위의 빨간색 점근선 안에 존재한다면, 파란색 공과 더 가까워 질 수 있는 방향으로 빨간색 공을 우회를 합니다. 이런 단순한 알고리즘이 universal한 형태를 띄울 수 있는 이유는, 위의 universal blue ball detour 알고리즘에서 이미 파란색 공을 threshold안에 항상 넣어놓고 있기 때문에,어느 위치의 빨간색 공을 피하는 과정에서도 우리가 먹을 파란색 공은 항상 먹을 수 있는 영역 안에 있기 때문입니다.

4.1.4.4 Ball releasing and adapting 3 cameras.

초록색 공에 가까워지기 위해서는 앞면의 근, 장거리 카메라를 사용하지만, 이후 정밀한 도킹과 공 releasing을 위해 후방에 카메라를 설치해 사용한다. 우리의 차체는 뒤쪽으로 공을 배출하기 때문에, 3번 카메라가 없다면 1,2번 카메라로 맞추고 180도 회전해야하는데, 180도를 정확하게 회전하는 방법이 전무하다고 판단하여 3번카메라를 사용하게 되었다. 3번 카메라를 이용한 알고리즘은 대강 다음과 같다. 먼저 초록 공을 인식하여 바구니의 앞으로 이동하게 만든 후, 어느 정도 거리가 되면 180도 회전한다. 이때 이 180도 회전은 정확하지 않을 것이다. 그 후, 3번 카메라를 통해 초록 공을 인식하면서 두 초록공을 이은 직선과 차체가 평행하게 조정한 뒤, 차체를 두 초록공 사이의 중점에 놓이도록 제어한 후, 그 후 직진하다가 어느정도 거리가 가까워지면 release하는 것이 우리 조의 전략이다.

4.1.5 Our coding schedule

아래는 우리의 대략적인 시스템 및 알고리즘 구현 스케줄로, 아래의 스케줄을 미리 정해 일을 진행했고, 결과적으로 시간에 이끌리지 않고 효율적으로 일을 할 수 있었다.

※실제로 코드를 디버깅 하는 과정이 제일 중요하기 때문에, 디버깅하는 과정은 한달 정도 잡아두는게 좋고, 디버깅을 하기 위해서는 하드웨어를 사용하기 때문에, 하드웨어 완성 기간과 ROS 디버깅 스케줄을 맞추는게 상당히 중요하다.

※또한, 디버깅 기간이 길면 길수록 좋은 이유는, 디버깅을 하면서도 하드웨어의 문제점과 개선점을 계속 찾기 마련인데, 하드웨어를 개선하는데는 시간이 필요하기 때문에 (ex. 3d 프린트하는데 걸리는 시간, 외주를 맡겼을 때 걸리는 시간 등등..) 최대한 일을 미리 해 놓고 디버깅 기간을 길게 하는 것이 좋다.

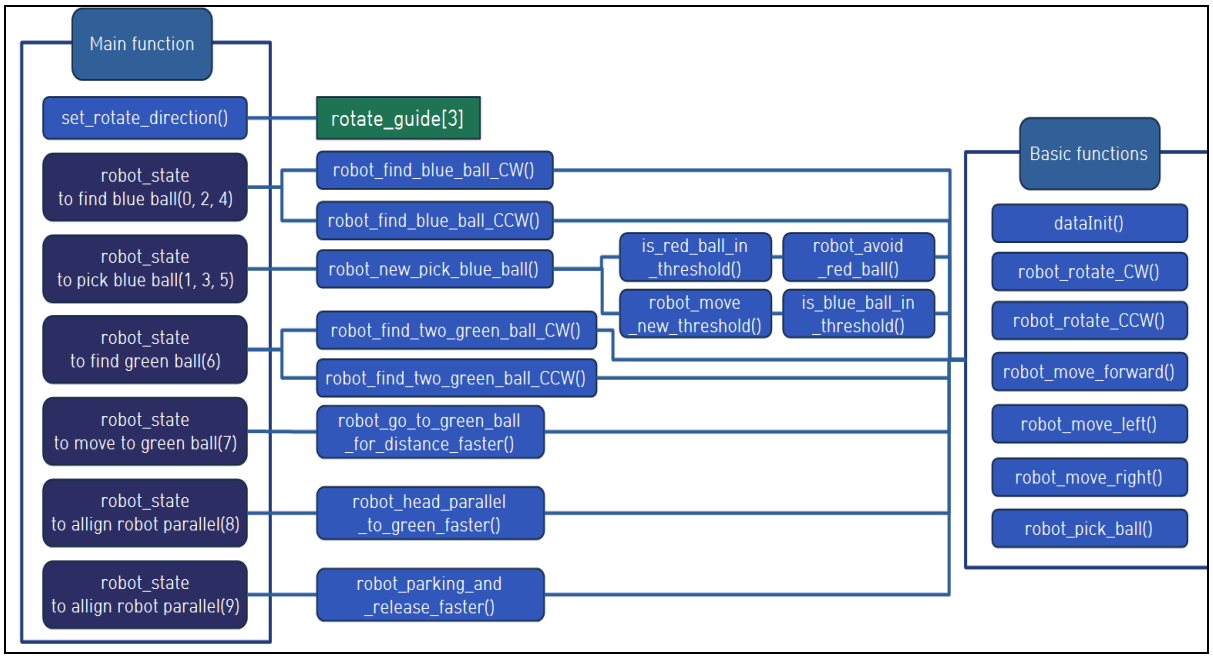
<table>
<tr><th colspan="2">3월</th><th colspan="2">4월</th></tr>
<tr><td rowspan="3">~3/31</td><td rowspan="3">학습 기간
- ROS 사용법 및 구조 공부
- 소스코드 이해
- roslaunch, rqt, gazebo, rosbag, publisher&subscriber 사용 이해
- c++언어 공부</td><td>~4/8</td><td>- 센서 종류 / 개수 최종 선정
- 이에 맞게 ROS 운영체제 구축</td></tr>
<tr><td>~4/19</td><td>- 전체 알고리즘 철학 & 구조 구축</td></tr>
<tr><td>~4/31</td><td>- 디버깅 위한 초안 알고리즘 구축
(하드웨어도 디버깅을 위해 이때를 목적으로 완성 계획)</td></tr>
</table>

<Fig. 4.1.8> 우리조의 ROS schedule

※5월은 시간날 때 마다 계속 디버깅 및 하드웨어 개선.

2.1.1 Blue ball picking

Blue ball pick up은 본 capstone design의 가장 핵심적인 부분이며 시간 단축 및 미션 성공을 위한 최적의 코드를 구현하기 위해 여러 차례 코드를 수정하였다. 전체적인 함수들 간의 대략적인 구조는 다음 diagram과 같다.



가장 왼쪽 Main function에는 main 함수 내의 구조이다. 공 탐색시 최적의 회전 방향을 결정하는 함수인 set_rotate_direction이 먼저 실행되고 난 후, robot_state라는 variable의 값에 따라 미션을 수행한다. robot_state는 0~10까지 있으며, 각각의 단계에서 수행하는 미션은 diagram에 설명되어 있다. robot_state가 10일 때는 미션 종료 후 잠시 정지하는 단계이다. 각 robot_state마다 사용되는 함수와 이들의 종속관계는 diagram과 같다. 대부분의 의사결정 함수는 마지막 Basic function에 의해 최종적으로 로봇의 움직임을 결정하게 된다. 최종적으로 구현된 코드의 main함수에서 미션 수행과 관련된 부분은 다음과 같은 구조이다.

```
// final main function
set_rotate_direction();

if (robot_state == 0)
{
else if (robot_state == 1)
{
else if (robot_state == 2)
{
else if (robot_state == 3)
{
else if (robot_state == 4)
{
else if (robot_state == 5)
{
else if (robot_state == 6)
{
else if (robot_state == 7)
{
else if (robot_state == 8)
{
else if (robot_state == 9)
{
else if (robot_state == 10)
{
else
{
    dataInit();
    data_print();
    std::cout << "final demo clear" << std::endl;
    return 0;
}
```

<Figure 2.1.1> Overall structure of main function

위 부분은 미션을 수행하는데 관련된 main함수의 부분으로 이전에는 ros publisher 및 subscriber 선언, TCP/IP 통신을 시작하는 코드가 포함되어 있다. 미션을 수행하는 단계는 robot_state라는 변수에 따라 나눠지며, 0~5 까지 파란공을 찾는 과정과 파란공을 줍는 과정을 3번 반복하고 6~9 까지 녹색공을 탐색하고 바구니에 접근하여 release하는 과정이다. robot_state 10은 미션 종료 후 잠시 대기하는 과정이다. Blue ball picking에 해당하는 0~5의 robot_state 중 0, 2, 4는 파란공을 탐색하는 과정이며 파란공을 인식하면 다음 단계인 1, 3, 5로 넘어가서 파란공으로 접근하고 pick up한다. 이 robot state는 다음과 같은 구조를 갖는다.

```
else if (robot_state == 2)
{
    std::cout << "robot_state " << robot_state << std::endl;
    std::cout << "rotate_guide = [" << rotate_guide[0] << rotate_guide[1] << rotate_guide[2] << "]" << std::endl;
    if (rotate_guide[0] == 1){
        if (robot_find_blue_ball_CW()){robot_state = 3;}
        data_print();
    }
    else{
        if (robot_find_blue_ball_CCW()){robot_state = 3;}
        data_print();
    }
}
else if (robot_state == 3)
{
    std::cout << "robot_state " << robot_state << std::endl;
    std::cout << "rotate_guide = [" << rotate_guide[0] << rotate_guide[1] << rotate_guide[2] << "]" << std::endl;
    if (robot_new_pick_blue_ball()){robot_state = 4;}
    data_print();
}
```

<Figure 2.1.2> robot_state to find blue ball and pick blue ball

파란공을 pick up하는 과정을 3번 반복하므로 위 구조는 robot_state 0, 1과 robot_state 4, 5에 대해서도 같이 적용된다. robot_state 0에서는 rotate_guide를 확인하는 과정없이 시계방향으로만 회전하여 공을 탐색한다. 하지만 로봇은 데모 시작 시점에서 파란공을 인식하고 있는 상태이므로, 이 부분은 최적화를 크게 감소시키지 않는다. 두 번째 파란공과 세 번째 파란공을 탐색하는 과정에서는 rotate_guide라는 더 유리한 회전방향을 알려주는 array의 값에 따라 회전방향을 결정한다.

rotate_guide가 1인 경우 시계방향으로 회전하고 0인 경우 반시계방향으로 회전한다. rotate_guide를 결정하는 set_rotate_direction 함수에 대해서는 2.1.2 find blue ball에서 설명한다. 로봇이 회전하여 공을 인식했다면 robot_state는 다음 단계로 넘어가서 공에 접근하고 pick up하는 robot_new_pick_blue_ball 함수를 실행한다. 이 함수는 다음과 같은 구조로 구성되어 있다.

```
int robot_new_pick_blue_ball()
{
    float threshold = 0.13/2;
    float red_threshold = 0.16;
    int stop_number = 100;
    // collect ball
    if ((ball_number_2_blue == 0 && global_state == 1) || (global_state2 == 2) && ball_distance_2_blue[0] > 0.3){
    // avoid red ball
    if (is_red_ball_in_threshold(red_threshold)){
        std::cout << "Avoid red ball" << std::endl;
        robot_avoid_red_ball(red_threshold);
        return 0;
    }
    // move to blue ball
    std::cout << "Get blue ball" << std::endl;
    robot_move_new_threshold(threshold);
    return 0;
}
```

<Figure 2.1.3> Function to pick blue ball

파란공을 pick up하기 위해 접근하고 빨간공을 회피하고, pick up module의 프로펠러를 회전시키는 움직임은 모두 이 함수 안에서 진행된다. collect ball 주석에 해당하는 if문은 pick up module의 프로펠러를 회전시키는 과정이다. 이에 대해서는 2.1.4 Pick blue ball에서 설명한다. avoid red ball과 move to blue ball은 각각 빨간공을 회피하는 과정과 파란공으로 접근하는 과정으로 robot_avoid_red_ball과 robot_move_new_threshold 함수에 의해 진행된다. 이에 대한 설명은 2.2.1 Red ball detouring과 2.1.3 Move to blue ball에 포함되어 있다. 위와 같은 과정을 통해 미션을 수행하게 된다.

2.1.2 Find blue ball

파란공을 찾기 위해 로봇은 기본적으로 제자리에서 회전한다. 초기의 최적화되지 않은 코드에서는 파란공을 찾기 위해 무조건 시계 방향으로 회전하였다. 이후, 시간 단축을 위해 수정한 코드에서는 로봇이 공을 pick up한 후 더 적은 각도를 회전할 수 있도록 구현하였다.

`int rotate_guide[3] = {0, 0, 0};`
 	
`void set_rotate_direction()`
`{`
	`int robot_find_blue_ball_CW()`
	`{`
	
	`int robot_find_blue_ball_CCW()`
	`{`

Rotation direction decision	Rotate function

<Figure 2.1.4> path planning result function and rotation function

set_rotate_direction은 main함수에서 1번만 실행되며 데모를 시작할 때, 파란공 3개를 보고있으면 파란공의 위치에 따라 회전 방향을 결정한다. 공을 pick up하는 순서는 거리 순서대로 진행하며 가장 가까운 공을 먼저 pick up한다. rotate guide의 index 0, 1, 2는 각각 첫 번째 파란공을 pick up한 후, 두 번째 파란공을 pick up한 후, 세 번째 파란공을 pick up한 후에 회전해야할 방향을 의미한다. 1이면 시계 방향, 0이면 반시계 방향으로 회전한다. 가장 가까운 파란공이 3개의 공 중에 가장 왼쪽에 있는 경우, 가운데에 있는 경우, 가장 오른쪽에 있는 경우로 나눠서 각 공의 x좌표를 비교하며 더 좋은 회전 방향을 판단한다. 로봇은 공을 약 4.5m 거리까지 인식할 수 있고, 데모를 시작하는 상황에서 공 3개를 인식할 수 있음을 확인하였다. 따라서, 위와 같은 방법을 이용하는 것은 유효하며, 실제로 파란공을 탐색하는 과정을 최적화할 수 있었고, 미션 수행 시간을 줄일 수 있었다.

2.1.3 Move to blue ball (is_blue_ball_in_threshold() & robot_move_new_threshold())

로봇이 파란 공을 향해 갈 때 로봇은 파란 공을 계속 보고 있어야 한다. 이를 유지하기 위해 우리는 **깔때기 모양의 threshold**를 가상으로 만든 후 일정 영역 안으로 파란 공이 들어오도록

로봇을 움직이게 한다. 먼저 is_blue_ball_in_threshold() 라는 함수는 우리가 만든 threshold 영역 안에 파란 공이 있는지 없는지를 판단해주는 함수이다. 다음은 그 코드의 일부분이다.

```
if (ball_number_2_blue > 0){
    if (ball_Y_2_blue[0] < neck){
        if (std::abs(ball_X_2_blue[0]) < threshold){return 1;}
        else if (std::abs(ball_X_2_blue[0]) > threshold){return 0;}
        return 0;
    }
    else if (ball_Y_2_blue[0] > neck){
        if (ball_Y_2_blue[0] > -m * (ball_X_2_blue[0] + threshold) + neck && ball_Y_2_blue[0] > m * (ball_X_2_blue[0] - threshold) + neck){return 1;}
        else{return 0;}
    }
    return 0;
}
```

```
else if (ball_number_1_blue > 0){
    if (ball_Y_1_blue[0] < neck){
        if (std::abs(ball_X_1_blue[0]) < threshold){
            return 1;
        }
        else if(std::abs(ball_X_1_blue[0]) > threshold){
            return 0;
        }
        return 0;
    }
    else if (ball_Y_1_blue[0] > neck){
        if (ball_Y_1_blue[0] > -m*(ball_X_1_blue[0] + threshold) + neck && ball_Y_1_blue[0] > m*(ball_X_1_blue[0] - threshold) + neck){
            return 1;
        }
        else{return 0;}
    }
    return 0;
}
return 0;
```

<Figure 2.1.5> is_blue_ball_in_threshold() function

먼저 파란 공이 일정 거리보다 가까이 있으면 공의 x좌표만을 비교해 threshold를 나타낸다. 하지만 일정 거리보다 멀리 있을 때는 threshold에 기울기를 주었다. 이 함수에서 threshold의 기울기는 10으로 설정했다. 이렇게 설정하면 로봇이 공과 멀리 떨어져 있을지라도 방향을 잡고 움직이지 않고 로봇이 직진을 하면서 방향을 맞추게 된다. 이는 로봇이 움직이는데 시간을 효율적으로 사용할 수 있다.

다음은 로봇을 움직이는 부분인 robot_move_new_threshold() 함수의 일부이다. 아래의 빨간 박스와 같이, 위에서 만든 is_blue_ball_in_threshold() 함수를 이용한 조건문을 사용함으로써, 파란색 공이 깔때기 모양의 threshold안에 있을 때 어떻게 행동할지 설정한다.

```
// blue ball is in threshold
if (is_blue_ball_in_threshold(threshold)){
    if (ball_Y_2_blue[0] > 1){
        dataInit();
        robot_move_forward(1);
        if(ball_X_2_blue[0] > 0){robot_rotate_CW(0.21);}
        else if(ball_X_2_blue[0] < 0){robot_rotate_CCW(0.21);}
        return;
    }
    else if (ball_Y_2_blue[0] > 0.5){
        dataInit();
        robot_move_forward(0.6*(ball_Y_2_blue[0] - 1) + 1);
        if(ball_X_2_blue[0] > 0){robot_rotate_CW(0.21);}
        else if (ball_X_2_blue[0] < 0){robot_rotate_CCW(0.21);}
        return;
    }
    else if (ball_Y_2_blue[0] > 0){
        dataInit();
        robot_move_forward(0.7);
        if(ball_X_2_blue[0] > 0){robot_rotate_CW(0.21);}
        else if (ball_X_2_blue[0] < 0){robot_rotate_CCW(0.21);}
        return;
    }
    return;
}
```

<Figure 2.1.6> robot_move_new_threshold() function

파란색 공이 threshold 안에 있을 때 로봇이 어떻게 움직여야 할지를 나타낸다. 파란 공이 로봇과

얼마나 멀리 떨어져 있냐에 따라 로봇의 직진 속도를 다르게 설정했다. 먼저 공이 로봇과 1m 이상 떨어져있는 경우에는 로봇이 최대 속도로 직진하도록 했고, 공이 로봇을 기준으로 0.5m 이내에 있을 경우에는 최대 속도의 0.7 배 속도로 움직이게 했다. 움직임의 자연스러움을 추가하기 위해 0.5m 와 1m 사이에는 속도가 linear하게 감소하도록 설정했다.
다음은 파란 공이 threshold 밖에 있는 경우이다.

```
// blue ball is on your left
else if(ball_X_2_blue[0] < 0){
    if (ball_Y_2_blue[0] > 1){
        dataInit();
        robot_move_forward(1);
        robot_rotate_CCW(0.74);
        return;
    }
    else if (ball_Y_2_blue[0] > 0.5){
        dataInit();
        robot_move_forward(0.6+(ball_Y_2_blue[0] - 1) + 1);
        robot_rotate_CCW(0.41);
        return;
    }
    else if (ball_Y_2_blue[0] > 0){
        dataInit();
        robot_move_forward(0.7);
        robot_rotate_CCW(0.34);
        return;
    }
    return;
}
return;
```

<Figure 2.1.7> Operation when the blue ball gets out of the threshold

이 경우에도 마찬가지로 로봇과 공이 얼마나 떨어져 있는지를 이용해 회전 속도를 다르게 설정했다.

2.1.4 Pick blue ball

우리 로봇은 프로펠러를 회전시켜 공을 들어올리는 구조이다. 프로펠러를 돌리는 모터의 속도를 제어할 수 있기 때문에 일정시간 동안 모터를 회전하는 방법을 선택했다. 또한 로봇이 공을 줍기 위해서는 공이 로봇의 바로 앞에 놓아져야 한다. 하지만 로봇의 구조상 프로펠러가 카메라 시야를 가려 일정거리 이내에 들어온 공은 위치를 확인하지 못한다. 따라서 우리는 하드웨어의 문제점을 개선하기 위해 공을 줍는 로봇의 움직임에 다양성을 부여했다. 카메라로 최대한 가까이 갔을 때에는 파란 공을 로봇의 중앙에 위치하도록 움직인 다음 공을 줍는 알고리즘을 실행했다. 로봇이 공을 줍는 과정은 총 3단계로 나눈다. 첫번째는 단순한 직진 운동, 두번째는 직진 운동과 프로펠러 회전, 마지막으로는 프로펠러만 회전하는 운동으로 나누었으며, 그 코드는 아래와 같다.

```
// collect ball
if ((ball_number_2_blue == 0 && global_state == 1) || (global_state2 == 2) && ball_distance_2_blue[0] > 0.3){
    dataInit();
    robot_move_forward(0.7);
    // robot_pick_ball(0.8);
    global_count++;
    // end collecting ball
    if (global_count < 10){
        dataInit();
        robot_move_forward(0.7);
        global_count++;
        return 0;
    }
    else if (global_count < 55){
        dataInit();
        robot_move_forward(0.7);
        robot_pick_ball(0.5);
        global_count++;
        return 0;
    }
    else if (global_count <= stop_number){
        dataInit();
        robot_pick_ball(0.5);
        global_count++;
        return 0;
    }
    else if (global_count > stop_number){
        global_count = 0;
        global_state = 0;
        global_state2 = 0;
        return 1;
    }
    return 0;
}
```

<Figure 2.1.8> robot_pick_new_blue_ball()

Global variable인 global_count의 값을 이용해 로봇이 동작하는 상태를 기억했고, global_count의 값에 따라 로봇은 다른 행동을 한다. 먼저 global_count가 10보다 작을 때에는 로봇이 직진 운동만 한다.

```
    if (global_count < 10){
        dataInit();
        robot_move_forward(0.7);
        global_count++;
        return 0;
    }
```

다음으로 global_count가 55 이하일 때는 직진 운동과 프로펠러 회전 운동을 함께 실행했다.

```
    else if (global_count < 55){
        dataInit();
        robot_move_forward(0.7);
        robot_pick_ball(0.5);
        global_count++;
        return 0;
    }
```

마지막으로 global_count가 stop_number보다 작은 경우에는 프로펠러 회전 운동만 실행했다. 이 함수에서 stop_number는 100으로 설정했다.

```
    else if (global_count <= stop_number){
        dataInit();
        robot_pick_ball(0.5);
        global_count++;
        return 0;
    }
```

2.2.1 Red ball detouring

파란 공을 따라가는 과정에서 가는 길에 빨간 공이 존재하는 경우 그 빨간 공을 피해야 한다.

따라서 우리는 파란 공을 쫓아가는 함수 안에 빨간 공을 피하는 함수를 작성했다. 먼저 기본적인 파란 공을 찾아서 가는 함수는 다음과 같다.

```
int robot_new_pick_blue_ball()
{
    float threshold = 0.13/2;
    float red_threshold = 0.16;
    int stop_number = 100;
    // collect ball
    if ((ball_number_2_blue == 0 && global_state == 1) || (global_state2 == 2) && ball_distance_2_blue[0] > 0.3){ ... }
    // avoid red ball
    if (is_red_ball_in_threshold(red_threshold)){
        std::cout << "Avoid red ball" << std::endl;
        robot_avoid_red_ball(red_threshold);
        return 0;
    }
    // move to blue ball
    std::cout << "Get blue ball" << std::endl;
    robot_move_new_threshold(threshold);
    return 0;
}
```

<Figure 2.2.1> How red ball detour algorithm is implemented in blue ball algorithm

코드의 기본 흐름은 공을 주워야 하는 상황이면 공을 줍고, 그렇지 않다면 파란 공을 주으러 가는데, 중간에 빨간 공이 있으면 피하는 과정을 넣었고, 그 내용을 is_red_ball_in_threshold 라는 함수와 robot_avoid_red_ball 함수에 작성했다. 먼저 is_red_ball_in_threshold 함수는 로봇이 파란 공을 찾아 가는 도중에 빨간 공을 발견했을 때 피할지 말지를 판단해주는 함수이다. 코드는 아래와 같다.

```
bool is_red_ball_in_threshold(float red_threshold)
{
    // blue ball in camera 2
    if (ball_number_2_blue > 0){
        // red ball in camera 2
        if (ball_number_2_red > 0){
            // blue ball is close
            if (ball_Y_2_red[0] > ball_Y_2_blue[0]){return 0;}
            // red ball is close
            else if (ball_Y_2_red[0] < ball_Y_2_blue[0]){
                if (std::abs(ball_X_2_red[0]) < red_threshold && ball_Y_2_red[0] < 0.6){return 1;}
                return 0;
            }
            return 0;
        }
        // red ball in camera 1
        else if (ball_number_1_red > 0){return 0;}
        return 0;
    }
    // blue ball in camera 1
    else if (ball_number_1_blue > 0){
        // red ball in camera 2
        if (ball_number_2_red > 0){
            if (std::abs(ball_X_2_red[0]) < red_threshold){return 1;}
            return 0;
        }
        // red ball in camera 1
        else if (ball_number_1_red > 0){
            // blue ball is close
            if (ball_Y_1_blue[0] < ball_Y_1_red[0]){
                return 0;
            }
            // red ball is close
            else if (ball_Y_1_blue[0] > ball_Y_1_red[0]){
                if (std::abs(ball_X_1_red[0]) < red_threshold && ball_Y_1_red[0] < 0.6){return 1;}
                return 0;
            }
        }
        return 0;
    }
    return 0;
}
```

<Figure 2.2.2> is_red_ball_in_threshold()

전방을 보는 카메라를 2개를 사용했기 때문에 각 카메라에서 빨간 공이 보일 때를 따로 코드를 작성했다. 이 함수는 카메라에 보이는 빨간색 공을 피해야 할 상황이라고 판단될 때 1을 return해준다. 빨간 공을 피하는 기준은 로봇을 기준으로 정면을 봤을 때 x좌표를 비교해서 로봇의 폭 크기보다 작은 값을 가지면 로봇이 직진할 때 빨간 공을 건드릴 수 있다고 판단해 1을 반환한다. 우리 로봇의 가로 폭은 46cm 이고 공을 줍는 프로펠러 부분은 13cm 이기 때문에 일정거리를 프로펠러 폭의 2배인 26cm로 설정했다. 단, 프로펠러의 폭 안에 빨간 공이 들어와 있지만 공이 멀리 있는 경우에는 공을 피하지 않기를 원했기 때문에 로봇의 정면으로 60cm 안에 있는 경우에만 공을 피하게 했다.

다음은 빨간 공을 피하는 함수인 robot_avoid_red_ball 이다. 이 함수는 빨간 공이 로봇 앞에 있다는 가정 하에 실행하는 함수이다. 아래 코드는 robot_avoid_red_ball 의 일부이다.

```
if (is_cam2_detect_red_ball_in_threshold(red_threshold) > 0){
    std::cout << "if (is_cam2_detect_red_ball_in_threshold)" << std::endl;
    if (ball_X_2_red[is_cam2_detect_red_ball_in_threshold(red_threshold)-1] > 0){
        std::cout << "if (ball_X_2_red[~~~~~~ -1] > 0)" << std::endl;
        dataInit();
        if (ball_Y_2_red[is_cam2_detect_red_ball_in_threshold(red_threshold)-1] > 0.5){robot_move_forward(0.5);}
        robot_move_left(0.5);
        if (ball_number_2_blue != 0){
            if (ball_X_2_blue[0] > 0){robot_rotate_CW(0.111);}
            else if (ball_X_2_blue[0] < 0){robot_rotate_CCW(0.111);}
        }
        else if (ball_number_1_blue != 0){
            if (ball_X_1_blue[0] > 0){robot_rotate_CW(0.111);}
            else if (ball_X_1_blue[0] < 0){robot_rotate_CCW(0.111);}
        }
        return;
    }
    else if (ball_X_2_red[is_cam2_detect_red_ball_in_threshold(red_threshold)-1] < 0){
        std::cout << "else if (ball_X_2_red[~~~~~~ -1] < 0)" << std::endl;
        dataInit();
        if (ball_Y_2_red[is_cam2_detect_red_ball_in_threshold(red_threshold)-1] > 0.5){robot_move_forward(0.5);}
        robot_move_right(0.5);
        if (ball_number_2_blue != 0){
            if (ball_X_2_blue[0] > 0){robot_rotate_CW(0.111);}
            else if (ball_X_2_blue[0] < 0){robot_rotate_CCW(0.111);}
        }
        else if (ball_number_1_blue != 0){
            if (ball_X_1_blue[0] > 0){robot_rotate_CW(0.111);}
            else if (ball_X_1_blue[0] < 0){robot_rotate_CCW(0.111);}
        }
        return;
    }
    else{
        std::cout << "Error: enter else 1" << std::endl;
        return;
    }
    return;
}
```

<Figure 2.2.3> robot_avoid_red_ball()

코드는 주으러 가는 파란 공의 위치와 피해야 하는 빨간 공의 위치 값에 따라 경우를 나눈다. 빨간 공이 로봇을 기준으로 왼쪽에 있을 때와 오른쪽에 있을 때 피해야 하는 방향을 다르게 설정했다. 직진 움직임과 좌 직진, 우 직진 움직임을 섞어 대각선으로 움직이는 코드를 짜 빨간 공을 피하도록 했다. 또한 피하는 도중에도 카메라가 파란 공을 놓치지 않도록 좌 직진, 우 직진 움직임에 대한 보상으로 회전운동을 함께 추가했다.

3.3. Ball Drop Algorithm.

3.3.1 Drop-module hardware and following considerations when structuring algorithm

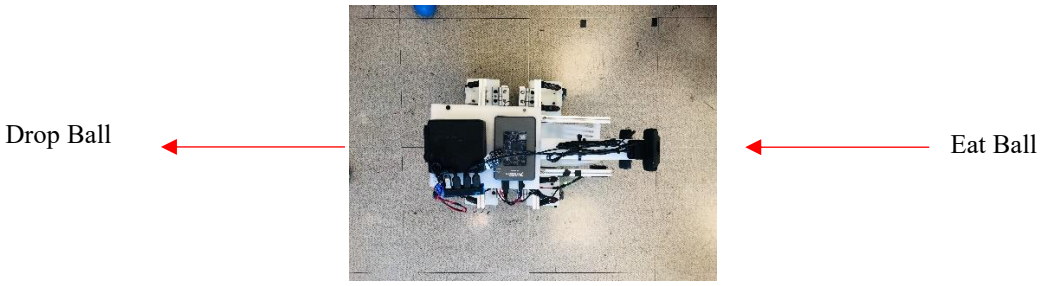


Fig 3.3.1 Separate pick-up module and drop-off module

Before understanding the ball drop algorithm, one needs to understand how our hardware's designed to pick up and to drop off. What's essential is that the robot is **eating the ball at the front**, and **exit the ball behind**. This restriction must be reflected to the algorithm.

3.3.2 main() function

```
else if (robot_state == 6)
{
  std::cout << "robot_state " << robot_state << std::endl;
  std::cout << "rotate_guide = [" << rotate_guide[0] << rotate_guide[1]
  << rotate_guide[2] << "]" << std::endl;
  if (rotate_guide[2] == 1){
    if (robot_find_two_green_ball_CW()){robot_state = 7;}
    data_print();
  }
  else{
    if (robot_find_two_green_ball_CCW()){robot_state = 7;}
    data_print();
  }
}
else if (robot_state == 7)
{
  std::cout << "robot_state " << robot_state << std::endl;
  if (robot_go_to_green_for_distance_faster()){robot_state = 8;}
  data_print();
}
else if (robot_state == 8)
{
  std::cout << "robot_state " << robot_state << std::endl;
  if (robot_head_parallel_to_green_faster()){robot_state = 9;}
  data_print();
}
```

```
else if (robot_state == 9)
{
  std::cout << "robot_state " << robot_state << std::endl;
  if (robot_parking_and_release_faster()){robot_state = 10;}
  data_print();
}
else if (robot_state == 10)
{
  std::cout << "robot_state " << robot_state << std::endl;
  if (robot_wait_second(1)){robot_state = 11;}
  data_print();
}
else
{
  dataInit();
  data_print();
  std::cout << "final demo clear" << std::endl;
  return 0;
}
```

Robot state 6 is when the robot finished eating the ball and looking for the green balls that flag the exit. However, note that we use the path planning data stored before. The sign of third index of rotation guide("rotation_guide[2]") will tell us which direction to rotate in order to rotate smaller angle to see the green ball. Then, the robot will rotate in either clockwise or counter-clockwise direction using the "**robot_find_two_green_ball_CW** / CCW()" functions. After those functions determine it found two green balls, the functions return 1, which updates the robot_state to be 7.

At **robot_state 7**, it operates **robot_go_to_green_for_distance_faster()**. This function makes the robot to get close to the two green ball until the robot reaches certain distance from the green balls.

At **robot_state 8**, the **robot_head_parallel_to_green_faster()** | a. makes the head of the robot parallel to the exit | b. move parallel to the center point of two green ball. Completion of this function updates the robot state to 9.

At **robot_state 9**, **robot_parking_and_release**() operates. Here, the robot rotates so that the rear part of the robot can face the green balls. Then, it docks to the exit and releases the ball. Completion of this function updates the robot state to 10.

At **robot_state 10**, the robot wait for a second, after which all the operations are completed.

We will further explain how each of the function is structured.

3.3.3 robot_find_two_green_ball_CW()

```
int robot_find_two_green_ball_CW()
{
  if (ball_number_1_green == 0){
    dataInit();
    robot_rotate_CW(1);
    return 0;
  }
  else if (ball_number_1_green == 1){
    dataInit();
    robot_rotate_CW(0.2);
    return 0;
  }
  else if (ball_number_1_green == 2){return 1;}
  return 0;
}
```

The robot starts looking for green ball using number of green ball detected at camera 1, or the far-view camera (ball_number_1_green). If there is no green ball in sight, the robot rotates clockwise with its maximum speed. Then, as soon as one ball is detected, it slows its speed. This is because if the rotation is too fast, fps may not fast enough to make sure that the next frame captured has the two green ball. After it detects all the two green ball, the function returns 1 in order to terminate this function.

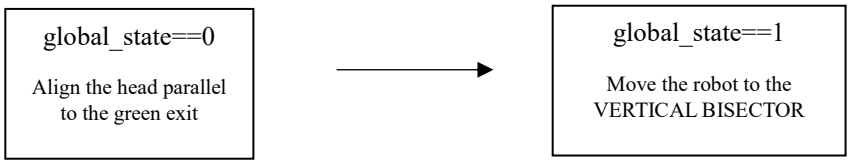
3.3.4 robot_go_to_green_for_distance_faster()

```
int robot_go_to_green_for_distance_faster()
{
  float x_threshold = 0.2;
  float x_sum = ball_X_1_green[0] + ball_X_1_green[1];
  float ball_car_distance = 0.75;
  if (ball_Y_1_green[0] > ball_car_distance){
①  if(x_sum > x_threshold){
      dataInit();
      robot_rotate_CW(0.25);
      robot_move_forward(0.6);
      return 0;
    }
    else if (x_sum < -x_threshold){
      dataInit();
      robot_rotate_CCW(0.25);
      robot_move_forward(0.6);
      return 0;
    }
②  else if (std::abs(x_sum) < x_threshold){
      dataInit();
      robot_move_forward(1);
      return 0;
    }
    return 0;
  }
  else if (ball_Y_1_green[0] < ball_car_distance){
    dataInit();
    return 1;
  }
  return 0;
}
```

After we locate two green balls, we then go to green ball until it reaches a certain y-factor distance to the green ball, which is 0.75m. ("ball car_distance"). Here, we use the closer green ball to measure the distance (ball_Y_1_green[0]). We're using the closer ball, or the zero index ball position just in case when the second ball gets out of the camera sight, which is impossible to happen because at the below part we do the feedback control so that the robot stays in threshold, but we did it just in case.

Since we realize that command does not appear accurate in physical world, move forward command cannot always be straight. In order to keep the robot from driving astray, I added the x-distance value of the two green balls detected, added them (x_sum), and gave certain lower & higher threshold boundary to keep the x_sum value inside the threshold. If the robot recognizes that the robot went astray enough, it tries to get back into the threshold, as can be seen in ①. Of course, if the robot thinks it's in the threshold, it goes straight as fast as possible, as can be seen in ②. Then, after it reaches the other else if statement, when the robot decides that it has come close enough, the function is terminated by returning 1, which will lead us to the next robot_state.

3.3.5 robot_head_parallel_to_green_faster()



<Fig 3.3.2> Brief structure of "robot_head_parallel_to_green_faster()" function

As can be seen from the diagram above, the function comprises two state. The first step align the head of the robot parallel to the green ball exit. After it has made sure that it's parallel, the robot moves the robot to the vertical bisector of the line made by two green ball.

Global state 0 part of the function looks like this;

④

```
int robot_head_parallel_to_green()
{
  // Try to go parallel and go center of two green ball
  if (ball_number_1_green > 1){
    float y_diff_threshold = 0.013;
    float x_diff_threshold = 0.1;
    float escape_threshold = 0.2;

    //When the ass is not parallel yet
    if (((std::abs(ball_Y_1_green[1] - ball_Y_1_green[0]) > y_diff_threshold))
    && global_state == 0){
      float x_sum = ball_X_1_green[0] + ball_X_1_green[1];
      if (ball_X_1_green[0] > ball_X_1_green[1]){
        if (ball_X_1_green[1] < -escape_threshold){
          dataInit();
          robot_move_left(0.1);
          robot_rotate_CW(0.2);
          return 0;
        }
        else if(0 > ball_X_1_green[1] > -escape_threshold){
          dataInit();
          robot_rotate_CW(0.2);
          return 0;
        }
        return 0;
      }
```

```
      else if (ball_X_1_green[1] > ball_X_1_green[0]){
        if (ball_X_1_green[1] > escape_threshold){
          dataInit();
          robot_move_right(0.1);
          robot_rotate_CCW(0.2);
          return 0;
        }
        else if (ball_X_1_green[1] < escape_threshold){
          dataInit();
          robot_rotate_CCW(0.2);
          return 0;
        }
        return 0;
      }
      return 0;
    }
    else if(global_state == 0 && (std::abs(ball_Y_1_green[1] - ball_Y_1_green[0])
    < y_diff_threshold)){global_state = 1;}
```

③

The criteria for determining whether the robot is parallel to the line formed by the two green ball is the y-factor distance to both green balls. We say that if the y-factor distances to each of the two green balls are close enough (ball_Y_1_green[0] and ball_Y_green[1]), we say that the robot is parallel enough. I have the maximum difference value allowed between those two y-factor distance as 0.013m (y_diff_threshold).

"If (ball_X_1_green[0]>ball_X_1_green[1])" can be translated as; "If the x-distance of the closer green ball is larger than the x-distance of the farther green ball". If you think about it, the x-distance of closer ball can be greater than the x-distance of farther ball only when the robot is located on the right side of the exit. Then, we have to rotate clockwise in order to get closer to being parallel.

However, if the robot rotates too much in order to be parallel, there is a possibility that the green ball farther away may get out of the camera sight. This is a big problem because we need to see both green balls in order to operate parallel. Thus, we set up a escape_threshold, as presented in ③. So, if the green ball that's farther away tries to get outside the escape_threshold, it moves toward the center of the two green ball so that the ball cannot get out of the sight. For example, in the scenario where the robot determines that it's located on the right side of the exit, if the ball that's farther away gets over the escape threshold, it adds the move left operation so that the robot would never loose the farther ball outside the vision.

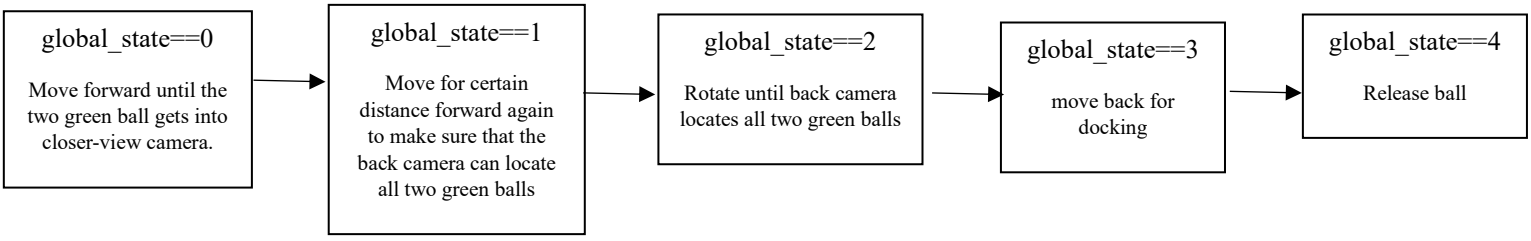
Vice versa happens on the other side of the two green balls, and once the robot determines that the robot head is parallel, as can be seen in ④, the condition statement changes the global state to be 1, which leads the function to the next global state.

Global state 1 part of the function looks like this;

```
  else if(global_state == 0 && (std::abs(ball_Y_1_green[1] - ball_Y_1_green[0])
  < y_diff_threshold)){global_state = 1;}
  //When the ass is parallel: We have to go to midpoint of two green balls
  else if (global_state == 1){
    ROS_INFO("Parallel has been done. Now starting to get to midpoint");
⑥   if (ball_X_1_green[0] + ball_X_1_green[1] > x_diff_threshold){
      robot_move_right(0.2);
      return 0;
    }
    else if(ball_X_1_green[0] + ball_X_1_green[1] < -x_diff_threshold){
      robot_move_left(0.2);
      return 0;
    }
⑤   else if (std::abs(ball_X_1_green[0] + ball_X_1_green[1]) < x_diff_threshold){
      std::cout << "y difference" << ball_Y_1_green[1] - ball_Y_1_green[0] << std::endl
      global_state = 0;
      return 1;
    }
    return 0;
  }
  return 0;
}
```

"If (ball_X_1_green[0] + ball_X_1_green[1] >x_diff_threshold)" means that the two green balls are located to the right side of the robot. Noting that our objective in this step is to go to the vertical bisector of the two green ball, we have to move right in order to achieve our goal. Such is realized in ⑤. After covering the vice-versa scenario, once the robot determines that it has reached to the vertical bisector close enough as can be seen in ⑥, it initialize the global state because it's a global variable that's used for dividing steps, and then returns non-zero value, which is 1, to go to the next robot_state.

3.3.6 robot_parking_and_release_faster()



<Figure 3.3.3> Brief structure of "robot_parking_and_release_faster()"

As can be seen from the above diagram, this function approaches to the green ball, rotate backward so that camera behind can locate all the two green balls, and move back for docking and releasing the collected balls.
Below is the code when global state is 0;

```
int robot_parking_and_release_faster()
{
  float diff_x_threshold = 0.3;
  float diff_green_threshold = 0.05;
  float x_sum = ball_X_1_green[0] + ball_X_1_green[1];
  int move_forward_limit = 40;
  if (global_state == 0){
    if (ball_number_2_green != 2){
      if (x_sum > diff_x_threshold){
        dataInit();
        robot_move_right(0.12);
        robot_move_forward(0.3);
        return 0;
      }
```

⑦

```
      else if (x_sum < -diff_x_threshold){
        dataInit();
        robot_move_left(0.12);
        robot_move_forward(0.3);
        return 0;
      }
      else if (std::abs(x_sum) < diff_x_threshold){
        dataInit();
        robot_move_forward(0.4);
        return 0;
      }
      return 0;
    }
    if (global_state == 0 && ball_number_2_green == 2){
      dataInit();
      global_state = 1;
    }
  }
```

As can be seen from the code, the robot moves forward untill the two green balls are seen in the second camera AND there is no green ball in vision in the far-view camera. (Keeping the robot in threshold while going forward mechanism has been repetitious, so I'll skip explaining ths part.) After the robot reaches this state, it converts the global_state to be 1, as can be seen above in ⑦.

Next is the code when the global state is 1 and 2;

```
else if (global_state == 1){
  if (global_count < 50){
    dataInit();
    robot_move_forward(0.6);
    global_count++;
    return 0;
  }
  else if (global_count == 50){
    dataInit();
    global_state = 2;
    global_count = 0;
    return 0;
  }
  return 0;
}
```

```
else if (global_state == 2){
  if (ball_number_3_green == 0){
    robot_rotate_CW(1);
    return 0;
  }
  else if (ball_number_3_green == 1){
    robot_rotate_CW(0.7);
    return 0;
  }
  else if (ball_number_3_green > 1){
    float x_sum_3 = ball_X_3_green[0] + ball_X_3_green[1];
    if (x_sum_3 > diff_green_threshold){
      robot_rotate_CW(0.3);
      return 0;
    }
```

```
    }
    else if (x_sum_3 < -diff_green_threshold){
      robot_rotate_CCW(0.3);
      return 0;
    }
    else if (std::abs(x_sum_3) < diff_green_threshold){
      global_state = 3;
      return 0;
    }
  }
  return 0;
}
```

In global state 1, we move forward for 50 loop counts. This is because if we do not go close enough, the back camera cannot locate the green balls because it's view range is designed to cover very small but accurate area. Thus, 50 is an experiental value that guarantees such operation of the next step. After it loops 50 move forward operation, it makes the global state to be 2.

In global state 2, we rotate untill the back part of the robot goes close enough to the center point of the two green balls. Note that initially the robot rotates fast when looking for the green balls, and gets slower as it finds more green balls, eventually two. Then, it rotates untill the x_sum_3 value, which is the added x-direction value of green balls, gets under the threshold value set (diff_green_threshold). Then, it the function switches the global state to 3.

Next is the code when the global state is 3 and 4;

```
else if (global_state == 3){
  std::cout << "Y = " << ball_Y_3_green[0] << std::endl;

  dataInit();
  robot_move_back(0.6);

  if (ball_number_3_green < 2 || ball_Y_3_green[0] < 0.32){
    dataInit();
    global_state = 4;
    return 0;
  }
```

```
else if (global_state == 4){
  if (global_count < 30){
    dataInit();
    robot_exit_ball();
    global_count++;
    return 0;
  }
  else if (global_count == 30){
    dataInit();
    global_count = 0;
    global_state = 0;
    return 1;
```

In global state 3, it is obvious that we move back untill the y-distance to the closer green ball is 0.32m. This distance is an experimental value that makes the robot stop right in front of the basket. Once it reaches 0.32m, it goes to global state 4

In global state 4, the robot releases the balls collected. Unfortunately, labView could not operate the position control, so we rotated the exit motor until it rotates half-way. As a result, if you see the global state 4 code, the code operates exit_ball command for 30 loops. After it operates 30 loops, the balls would have left the robot physically, and the function finally returns 1 to finish this robot_state. Then, the MISSION IS COMPLETE!