2019 Spring ME492 Final Report

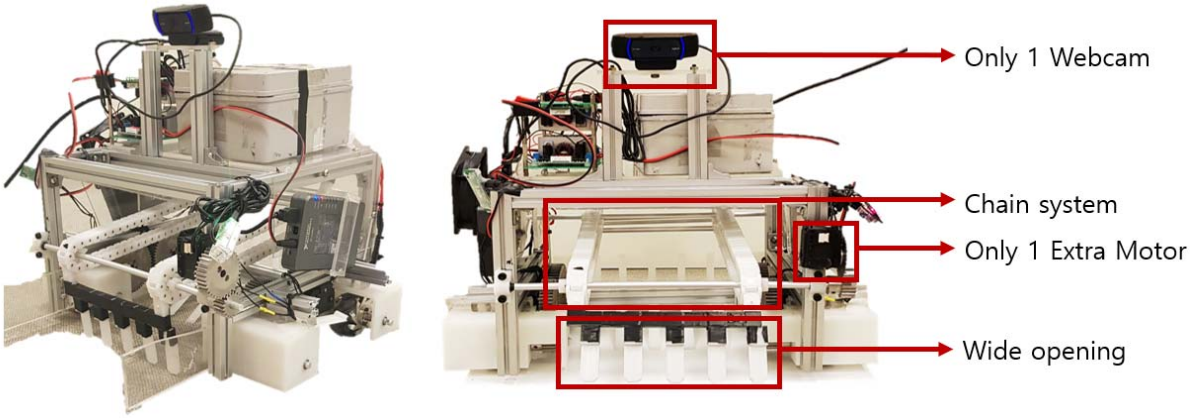
Team H
Junhwi Cho, 20150725
Sanguk Choi, 20150750

2019.06.16

목차

1. 로봇의 하드웨어 소개

본 조 로봇의 하드웨어는 아래 그림과 같이 구성되어 있다.



1.1 pick-up & drop off 방식

로봇에는 경사로 뒤에 공 저장공간이 있으며 체인에 부착된 pick-up용 blade로 공을 저장공간에 밀어 넣어 pickup을 한다. 파란 공 3개를 다 먹고 바구니 앞으로 이동하면 체인이 pick-up의 반대방향으로 돌면서 drop-off용 blade가 공을 앞으로 밀어내 바구니에 넣는다. 이때 체인은 모터에 의해 움직이며 모터의 회전방향에 따라 공을 pick-up, drop-off한다.

1.2 로봇의 하드웨어 특징 및 장단점

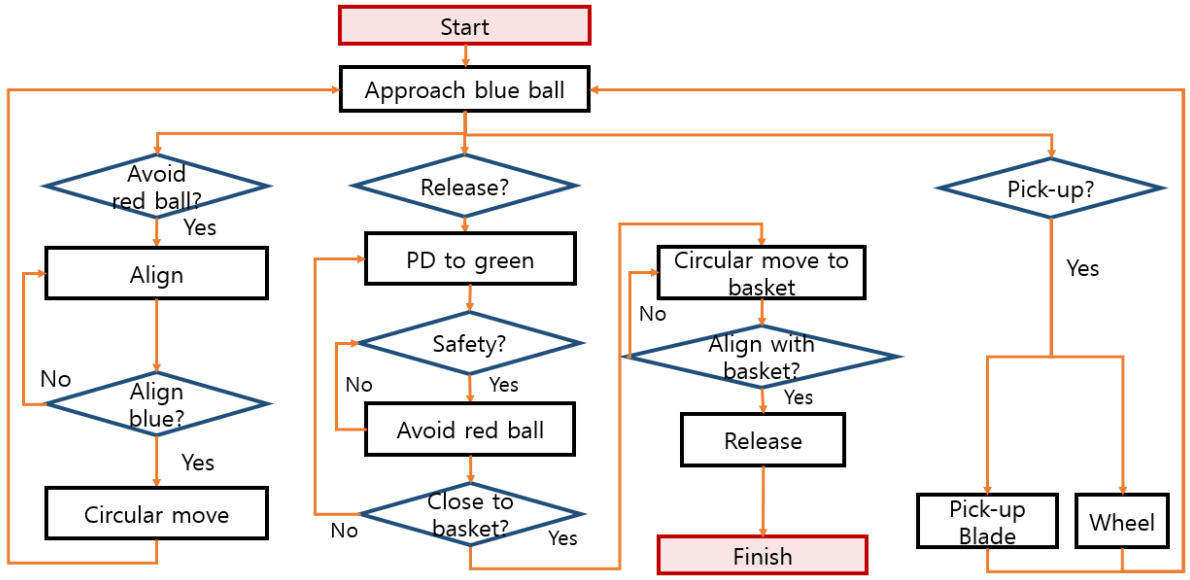
로봇의 출입구가 넓어서 파란 공을 주울 때 로봇의 위치 오차를 허용할 수 있다는 장점이 있다. 이로써 로봇의 위치를 조정하는 시간이 단축되어 미션 수행 시간을 단축할 수 있다. 그리고 pick-up & drop-off 방식으로 로봇에 사용되는 구동기의 개수를 최소화할 수 있었다. 하나의 체인 시스템으로 pick-up과 drop-off를 동시에 할 수 있으므로 로봇의 사용되는 구동기는 바퀴를 구동하기 위한 4개의 모터 이외에 단 하나의 모터만 추가적으로 사용하였다. 또한 웹 캠도 하나만 사용하였다. 로봇의 공 입구와 출구가 같아 바구니에 주차를 할 때 전면 주차하기 때문에 후면 주차를 해야 하는 조와 같이 후면에 웹 캠을 부착할 필요가 없었다. 또한 넓은 출입구로 인해 로봇의 바로 앞이 웹 캠의 시야에 있지 않아도 되기 때문에 웹 캠 하나로 경기장 전체를 보면서 pick-up과 drop-off까지 할 수 있었다.

로봇의 넓은 출입구로 인한 장점도 존재하지만 빨간 공을 피할 때 불리하다는 단점도 있다. 출입구가 넓으면 로봇의 너비도 불가피하게 넓어질 수밖에 없다. 따라서 빨간 공을 피할 때 많이 움직여야 한다. 이로 인해 빨간 공을 피하는 코드를 다른 조와 다르게 빨간 공을 피하는 동안 빨간 공의 위치 데이터를 계속 받아가며 피드백을 하여 피하도록 작성하였다.

2. 알고리즘 컨셉

2.1 전반적인 알고리즘

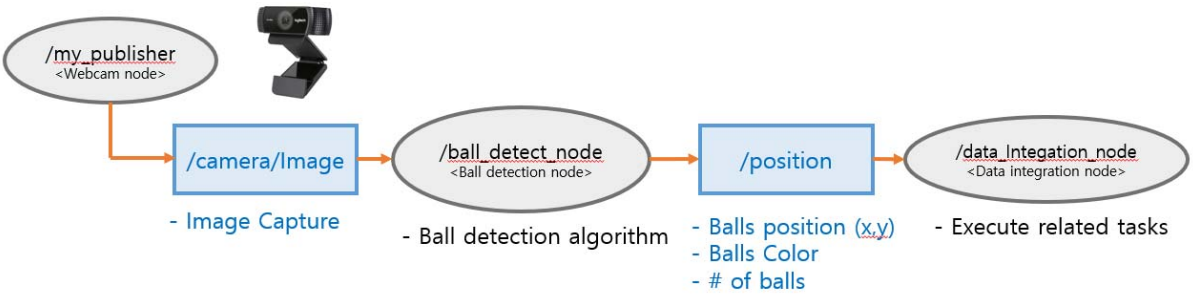
알고리즘의 Flow chart는 아래 그림과 같다.



가장 기본이 되는 state는 타겟 파란 공을 정하고 그 공을 향해 움직이는 Approach blue ball state이다. 여기서 타겟 파란 공은 웹 캠 상에서 가장 왼쪽에 있는 파란 공으로 정하며 따라서 로봇은 경기장을 시계방향으로 움직이게 된다. 이렇게 하면 파란 공을 찾기 위해 돌 때 무의미하게 180도 넘게 움직이는 것을 방지할 수 있다. 파란 공을 향해 움직이다가 빨간 공이 경로 상에 위치하면 빨간 공이 경로에서 벗어날 때까지 파란 공을 중심으로 원 운동을 하여 빨간 공을 피한다. 파란 공에 가까이 다가가면 pick-up을 하게 되는데 이때 pick-up blade와 바퀴는 state가 나눠져 있기 때문에 독립적으로 구동한다. 이 과정을 파란 공 3개를 먹을 때까지 반복하며 다 먹으면 Release state로 넘어가게 된다. 먼저 바구니를 향해 움직이고 바구니에 가까워지면 바구니의 중심을 기준으로 원 운동을 하여 로봇의 공 출입구와 바구니를 나란하게 조정한다. 조정이 되면 직진하여 바구니에 주차하고 마지막으로 drop-off blade로 파란 공들을 바구니에 넣는다.

2.2 rqt graph

rqt graph는 아래 그림과 같다. 본 조는 웹 캠 하나만을 사용하였기 때문에 rqt graph가 비교적 단순하다.



ROS part는 웹 캠의 이미지 데이터를 받아와서 상황 별로 필요한 모터의 각속도나 위치 값을 계산하여 myRIO에 전달해주는 역할을 한다. my_publisher node에서 ball_detct_node로 camera/Image message를 전달한다. Ball_detect_node는 받은 Image data를 가공하여 공 위치 정보와 색, 개수, 바구니 위치 정보를 position message로 publish하고 data_Integration_node에서 subscribe한다.

My_publisher node는 skeleton code에서 그대로 사용하였으며 ball_detect_node는 OpenCV part라서 OpenCV 담당 조원이 코딩하였다. Data_integration_node를 코딩하는 것이 ROS 담당이다.

2.3 코드에 담긴 철학과 아이디어

우리 조 알고리즘의 키워드는 Robustness이다. 최대한 Open loop를 사용하지 않고 Feedback을 사용하여 안전하게 임무를 수행하는 것을 최우선 목표로 하여 코딩을 하였다. 그러면서 미션 완수 시간도 단축하기 위해 여러가지 방법을 사용하여 코딩을 하였다. 아래는 우리 조 알고리즘의 철학을 구현하기 위한 아이디어들이다.

1) 코드를 state로 나누어 작성하였다.
(1) 전체 코드는 While문으로 이루어져 있기 때문에 for문을 사용할 수 없었다. 따라서 for문을 대신하여 if문을 사용해야 하여 코드가 매우 복잡해질 수 있다. 이런 문제를 해결하기 위해 state라는 개념을 이용하여 코드를 state라는 큰 묶음으로 분리하여 코드를 훨씬 간략하게 작성하였다.
(2) pickup 파트와 moving 파트를 독립적으로 구동할 수 있었다. 이를 통해 공을 pickup 할 때 움직이면서 pickup을 수행할 수 있었고, pick-up 전체 과정(blade가 공을 저장공간에 넣었다가 다시 원상태로 돌아오는 과정)이 수행되지 않더라도 blade가 저장공간을 밀폐시켜 pickup한 공이 밖으로 나가지 못하게 될 때부터 다음 공을 찾을 수 있었다. 이러한 점은 시간 단축에 큰 도움이 되었다.
(3) 각 state별로 조원간 코딩 업무를 분할하여 주된 node로 data integration node 하나만 사용함에 불구하고 한 명만 코드를 짜는 불상사를 막을 수 있었다.
(4) 데모 준비할 때 원하는 state만 작동시킬 수 있어서 디버깅하기 쉽고 새로운 test state를 만들어서 시험하기도 수월하였다.

2) 바퀴와 연결된 각 모터의 각속도 값을 독립적으로 입력하여 다양한 움직임을 구현할 수 있도록 하였다. 이를 이용해 원운동(공전), PD control 등의 움직임을 부드럽게 할 수 있었다. 특히 직진과 회전 운동을 동시에 작동하게 함으로써 빠르게 파란 공에 접근할 수 있게 되었다.

3) 원하는 위치로 이동할 때는 PD control을 이용하여 정확하게 목적지로 이동하도록 코드를 작성하였다.

4) Closed loop을 사용하기 가장 힘들었던 부분은 빨간 공을 피하는 코드였다. 빨간 공을 피하기 위해 움직이면 타겟 파란 공이 웹 캠 시야에서 벗어날 수 있기 때문이다. 이를 해결하기 위해 타겟인 파란공을 중심으로 원 운동하며 빨간 공을 피하는 내내 파란 공을 시야에 두도록 하였다.

5) 시간을 단축하기 위해 pick-up과 moving을 동시에 하였는데 파란 공 두개가 가까이 있는 경우 하나의 공을 먹고 pick-up용 blade가 원래 상태로 위치하기 전에 두 번째 공을 pick-up하라는 명령이 들어갈 수도 있다. 이를 방지하기 위해서 blade가 원상태로 위치하기 전에 다른 공을 타겟으로 잡을 때는 원상태로 위치할 때까지 moving을 멈추는 코드를 추가하였다.

3. 코드 설명

본 H조의 ros code는 대부분 data integration node에서 진행되고 처리되었다. 따라서 코드에 대한 설명은 data integration node에 대한 설명이 될 것이다. 전체적인 코드 설명은 사용한 코드 중 특징적인 변수들, 함수들, 그리고 본 조의 ros code 알고리즘 구현 방식인 state들에 대한 설명으로 진행할 예정이다. (추가로, data integration node에 주석이 존재해 이와 같이 읽으면 이해에 더 도움이 될 수 있다.)

3.1 변수

변수 설명은 크게 가장 중요한 Input 변수인 공들의 좌표 변수와, Output 변수인 myRIO로 보내주는 data[] 변수들에 대한 내용을 갖는다. (본 Platform은 1개의 Webcam 센서를 사용하고 4개의 wheel motors와 1개의 pickup motor를 구동기로 사용한다. 따라서 ROS에서 담당하는 역할은 Input 정보로 웹캠 변수 정보들을 받아 처리한 후 Output 정보를 5개의 모터에 각각 명령을 보내주는 역할을 한다.)

3.1.1 공 좌표 변수 선언

Data integration node는 OpenCV를 통해 처리한 파란 공, 빨간 공, 초록 공, 바구니에 대한 중심 좌표의 x, y 좌표 정보를 message로 받는다. 이렇게 받아온 변수 값을 새로운 벡터 자료형 변수로 선언하여 새로운 변수 값을 만들 때 사용하거나 뒤에서 공의 정보를 labelling 하는 등에 사용할 수 있도록 하였다.

```
//Callback function으로, 웹캠 정보가 들어올때 작동한다. 본격적인 모든 platform의 알고리즘이 담겨있다.
void camera_Callback(const core_msgs::ball_position::ConstPtr& position)//27Hz, v*t=3375=1m
{
    int count=position->img_rx.size()+ position->img_bx.size(); //웹캠에 보이는 빨간공과 파란공의 개수
    ball_number=count;
    ball_rX.resize(position->img_rx.size());                    //메세지에서 받아오는 빨간공들의 x좌표를 저장하는 변수 사이즈 동기화.
    ball_rY.resize(position->img_ry.size());                    //메세지에서 받아오는 빨간공들의 y좌표를 저장하는 변수 사이즈 동기화.
    ball_bX.resize(position->img_bx.size());                    //메세지에서 받아오는 파란공들의 x좌표를 저장하는 변수 사이즈 동기화.
    ball_bY.resize(position->img_by.size());                    //메세지에서 받아오는 파란공들의 y좌표를 저장하는 변수 사이즈 동기화.
    ball_gX.resize(position->img_gx.size());                    //메세지에서 받아오는 초록공들의 x좌표를 저장하는 변수 사이즈 동기화.
    ball_gY.resize(position->img_gy.size());                    //메세지에서 받아오는 초록공들의 y좌표를 저장하는 변수 사이즈 동기화.
    basket_X.resize(position->img_basket_x.size());             //메세지에서 받아오는 바구니의 x좌표를 저장하는 변수 사이즈 동기화.
    basket_Y.resize(position->img_basket_y.size());             //메세지에서 받아오는 바구니의 y좌표를 저장하는 변수 사이즈 동기화.

    red_ball_distance.resize(position->img_rx.size());          //공, 바구니까지의 거리를 저장할 변수 사이즈 동기화.
    blue_ball_distance.resize(position->img_bx.size());
    green_ball_distance.resize(position->img_gx.size());
    basket_distance.resize(position->img_basket_x.size());

    //316줄~340줄 : 메세지에서 받아오는 공, 바구니들의 좌표 정보를 새로운 array 변수에 입력하는 과정. 이 과정을 하는 이유는 후에 target을 설정하는데에 있어서
    //공들을 labelling하여 더 쉽게 코딩하기 위함이다.
    for(size_t i = 0; i < position->img_rx.size(); i++)
    {
        ball_rX[i]=position->img_rx[i] ;
        ball_rY[i]=position->img_ry[i] ;
        red_ball_distance[i] = sqrt(position->img_rx[i]*position->img_rx[i]+position->img_ry[i]*position->img_ry[i]);
    }
    for(size_t i = 0; i < position->img_bx.size(); i++)
    {
        ball_bX[i]=position->img_bx[i] ;
        ball_bY[i]=position->img_by[i] ;
        blue_ball_distance[i] = sqrt(position->img_bx[i]*position->img_bx[i]+position->img_by[i]*position->img_by[i]);
    }
    for(size_t i = 0; i < position->img_gx.size(); i++)
    {
        ball_gX[i]=position->img_gx[i] ;
        ball_gY[i]=position->img_gy[i] ;
        green_ball_distance[i] = sqrt(position->img_gx[i]*position->img_gx[i]+position->img_gy[i]*position->img_gy[i]);
    }
    for(size_t i = 0; i < position->img_basket_x.size(); i++)
    {
        basket_X[i]=position->img_basket_x[i] ;
        basket_Y[i]=position->img_basket_y[i] ;
        basket_distance[i] = sqrt(position->img_basket_x[i]*position->img_basket_x[i]+position->img_basket_y[i] *position->img_basket_y[i] );
    }
```

위 코드와 같이 메세지를 통해 받은 빨간 공, 파란 공, 초록공과 바구니에 대한 x좌표들,

y좌표들의 변수들과 index 정보를 새로운 vector array들을 만들어 입력해 주었다. 이렇게 받은 메시지 정보를 새로운 변수로 만들어서 사용해 얻은 장점은 다음과 같다.

- Subscribe한 메시지 정보는 수정할 수 없는데, 이렇게 변수를 새로 만들어 추가하면 수정해서 사용이 가능하다.
- 변수 이름 길이를 많이 축소할 수 있다.

단, 새로운 변수를 만들었기 때문에 array 크기 등과 같은 점을 신경써주지 않을 경우 core dumped error가 나타나는 문제가 발생하니 유의해서 사용해야 한다.

3.1.2 Data[] 변수 선언

Data integration node에서 처리한 결과 data[]를 myRIO로 보내주어 motor에 그때그때 필요한 입력 값을 제공하는 게 가장 중요한 요소 중 하나인데, 본 조의 알고리즘은 data[1]~data[8] 변수들을 할당해 Output 정보로서 myRIO에 송신하였다.

```
85  //86줄 ~ 294줄 코드에서 사용하는 함수들을 선언.
86  void dataInit()     //myRIO에 보내주는 data 변수들 초기화 함수 선언.
87
88  {
89    data[0] = 0;
90    data[1] = 0; //wheel 1 각속도
91    data[2] = 0; // wheel 2 각속도
92    data[3] = 0; // wheel 3 각속도
93    data[4] = 0; //wheel 4 각속도
94    data[5] = 0; //pick-up motor 각속도
95    data[6] = 310; //pick-up motor 위치제어 각도
96    data[7] = 1;   //pick-up motor 속도제어, 각도제어 trigger (1:position 0:velocity)
97    data[8] = 30;  //모터 maximum acceleration값
```

각각 할당된 변수의 역할은 위 코드처럼 지정하였다.

- data[1]~data[4]는 각 바퀴 구동부의 모터 회전 각속도 값
- data[5]는 pickup 모터의 회전 각속도 값(속도제어 할 경우)
- data[6]는 pickup 모터의 angle position 값(위치제어 할 경우, 이때 absolute encoder를 사용하는 모터를 써서 0~360 사이에서만 사용 가능해 제한적인 조건에서만 사용 가능했다.)
- data[7]은 pickup 모터의 속도제어(0일때), 위치제어(1일때) 를 선택하는 trigger 역할을 했다.
- data[8]은 사용하는 전체 5개 모터들의 maximum acceleration 값을 제한하는 값으로, 모터에 걸리는 부하의 상한선에 제한을 걸어 부드러운 움직임을 하도록 했다.

(여기서 myRIO로 보낸 데이터들이 모터의 입력 값으로 들어가도록 하거나, maximum acceleration을 제한하도록 하는 과정 등은 Labview에서 구현하였다.)

이렇게 myRIO에 보낼 output 변수를 설정하는 것은 꽤나 당연하고 별거 없어 보일 수 있지만, 본 조의 알고리즘 구현에 매우 중요한 역할을 했다.

- 각 모터의 각속도 값을 따로 입력해줄 수 있어서, 단순히 몇 가지 운동 패턴들을 저장해두고 사용하는 것이 아닌 원하는 다양한 운동들을 구현할 수 있었다. 이를 통해 간단한 직선 운동이나 회전 운동이 아닌 각 바퀴의 속도 벡터 값을 더해 원운동, PD 제어 등을 구현할 수 있었다.
- 즉 부드러운 곡선 운동 등이 가능해 시간, 경로에서 이득을 얻을 수 있고 진정한 메카넘 휠의 장점들을 끌어낼 수 있었다.

3.2 함수

알고리즘을 구현하는 데에 있어서 자주 사용하는 몇 가지들을 함수로 만들어서 사용했는데, 이는 크게 두가지 종류로 나눠졌다. 1. 차량의 움직임을 구현하거나 이와 관련된 함수, 2. 타겟 설정 함수들로 pickup할 파란공을 선정하거나, 회피할 빨간 공을 선정하거나, release할 지점의 좌표를 얻기 위한 초록 공 위치 추출 함수. 이 함수들은 뒤에 나오는 전체 알고리즘의 뼈대를 구성하는 state들 속에서 남은 알고리즘의 살의 역할을 한다.

3.2.1 차량 움직임과 관련된 함수

차량의 움직임을 구현하는 함수는 3.1.2에서 정의한 변수 중 각 바퀴의 모터 각속도 값인 data[1]~data[4]에 적절한 값을 입력해주면 움직임이 구현이 되고, pickup은 data[5]나 data[6]에 원하는 회전 속도 값을 넣거나 position 값을 넣어주면 움직임을 구현할 수 있었다.

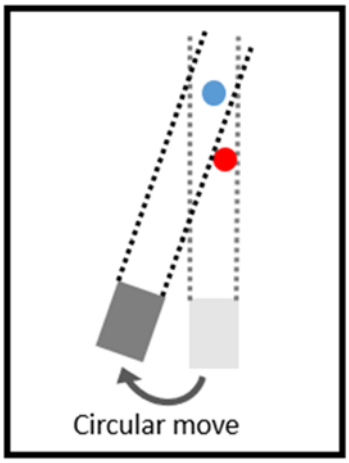
data_integration_edit1.cpp

```
114  //115줄 ~ 218줄 : 차량 움직임과 관련된 함수. 자주 사용하는 움직임으로 함수화 하였다.
115  //pickup motor 회전하는 함수. 속도제어
116  int pick_up(float vel_p) {
117    data[5] = vel_p;
118    return 0;
119  }
120
121  //직진운동 함수.
122  int movef(float ang_vel){
123    data[1] = ang_vel;
124    data[2] = ang_vel;
125    data[3] = ang_vel;
126    data[4] = ang_vel;
127    cout<<"moving"<<endl;
128    return 0;
129  }
130
131  //회전운동 함수.
132  int align(float ang_vel){
133    data[1] = ang_vel;
134    data[2] = -ang_vel;
135    data[3] = ang_vel;
136    data[4] = -ang_vel;
137    cout<<"aligning"<<endl;
138    return 0;
139  }
140
141  //좌회전 함수.
142  int move_left(float ang_vel){
143    data[1] = ang_vel;
144    data[2] = ang_vel;
145    data[3] = -ang_vel;
146    data[4] = -ang_vel;
147    cout<<"move left"<<endl;
148    return 0;
149  }
150
151  //우회전 함수.
152  int move_right(float ang_vel){
153    data[1] = -ang_vel;
154    data[2] = -ang_vel;
155    data[3] = ang_vel;
156    data[4] = ang_vel;
157    cout<<"move right"<<endl;
158    return 0;
159  }
```

~/Downloads/data_integration_edit1.cpp 166:26

위 그림에서 나타난 코드는 pickup, 전진, 후진 운동, 회전운동, 왼쪽으로 병진 운동, 오른쪽으로 병진 운동을 하는 함수를 만들어 놓은 코드이다. 단순히 data[1]~data[4]에 적절한 값이나 부호를

넣은 묶음 만을 함수로 만들었는데, 이에 원하는 속도만을 입력하면 바로 그 움직임을 사용해서 쓸 수 있다. 이렇게 쉬운 운동을 함수로 굳이 만들어서 써야하나 싶을 수 있지만 자주 쓰는 움직임이다 보니 함수로 지정해서 사용하면 코드를 짤 때 굉장히 편리하다. 또한 여기서 3.1.2에서 data[] 변수에 값을 입력하는 방식의 장점이 나타나는데, 이러한 함수 생성 과정에서 data[1], data[2], data[3], data[4] 값에 원하는 값을 넣어 함수를 만들면 그 각각의 속도의 벡터 합이 운동으로 구현이 되고 함수로 지정이 된다. 그 사례인 원운동과 PD control의 코드를 아래에 열거하겠다.



data_integration_edit1.cpp

```
161  //원운동 함수. 원의 중심을 항상 바라보는 공전운동이다.
162  //r은 원의 중심까지의 거리(주로 파란공이나 바구니까지 거리가 된
163  //v는 원운동시의 선속도이다.
164  //t는 원운동이 실제로 구현하는데에 생기는 오차를 줄이기 위해 만
165  void circular_move(float r,float v, float t){
166    data[1]= -v+v*r0/r+kc*t;
167    data[2]= v-v*r0/r-kc*t;
168    data[3]= v+v*r0/r+kc*t;
169    data[4]= -v-v*r0/r-kc*t;
170  }
171
```

위에 나와있는 주석처럼 위 운동은 움직이는 Platform이 원운동을 하는 중심을 항상 중앙에 바라보며 운동하는 함수로, 쉽게 말해 지구를 항상 같은 면으로 바라보는 달과 같은 공전 운동을 하는 운동이다.

먼저 위와 같은 운동을 구현하게 된 계기는 Platform이 빨간 공을 피하는 운동을 할 때, 최적의 짧은 루트만을 이동하며 Robust하게 빨간 공을 먹지 않는 운동을 고안하고 있었다. Robust함을 유지하기 위해서는 Closed loop control이 필요하고, 동시에 짧은 루트로 피해가려면 기존에 pickup하려던 파란공을 향해 가는 운동에서는 많이 벗어나지 않도록 해야 했다. 위 조건들을 고민하다가, 이들을 충족시키는 운동인 공전하는 원운동을 발견했다. 공전하는 원운동을 이용한 움직임의 장점은 다음과 같다.

- 항상 회피할 빨간 공을 시야에 두고 target 파란공을 중심에 두고 회전하기 때문에 빨간 공을 피하면서 계속 control이 가능하다. 즉 항상 Closed loop control이 가능하다.
- 빨간 공을 회피하는데 부수적으로 필요한 이동 경로가 효율적이다.
- 안전거리가 확보된 후 즉 빨간 공 피하기가 종료된 시점에서도 target 파란공을 중심에 두고 있어 pickup하기 위해서 직진운동만 하면 된다.

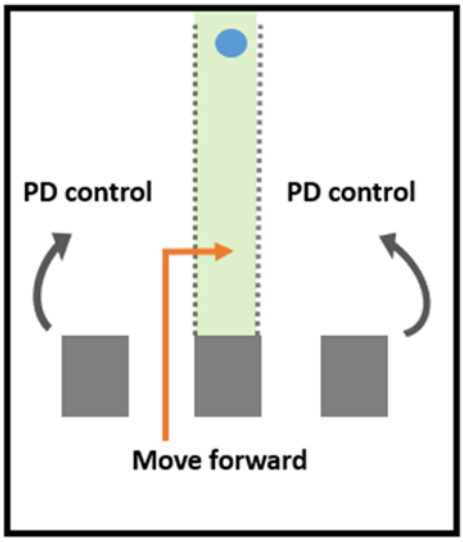
원운동의 구현 방법은 다음과 같다. 먼저 차는 공전하려는 방향으로 일정한 선속도를 가진다. 이와 동시에, 공전 주기와 맞물리도록 자전을 하기 위해 공전 각속도와 자전 각속도를 일치시키도록 한다. 따라서 옆으로의 병진운동과 원하는 공전 각속도만큼의 자전 각속도를 갖는 회전운동의 벡터 합을 위 코드에서처럼 각 바퀴에 입력하면 구현할 수 있다.

이에 주의할 점이 있는데, 실제 Platform은 완전한 원 운동을 하지 않을 수 있다. 본 조의 Platform의 경우 특정 지역에서는 원운동 구현이 잘 일어났는데, 미끄럼이 심한 구역에서는 원운동이 잘 일어나지 않을 때가 존재했다. 이를 위해 원운동에서의 회전운동 값을 파란공이 중심에서 멀어진 만큼 보상하는 코드를 작성했다. 위 코드에서 t가 theta error의 값이고, kc가 p gain의 역할을 하도록 하여 더 안정적인 제어를 구현할 수 있었다.

```cpp
//PD control하는 함수 선언이다.
//x, y값은 타겟 공과의 x거리 오차, y 거리 오차로 공의 x좌표와 y 좌표가 된다.
//m, n은 주행의 효율을 위해 둔 PD control의 작동 오차범위 threshold로 오차가 이
int PD_control(float x, float y, float m, float n){
  double theta_pd=atan(x/y);     //거리 좌표를 이용해 angle 변수 생성
  if(sqrt(x*x+y*y)>65){          //거리에 따라서 제어의 threshold를 다르게
    if(x>m){
      data[1]=50+theta_pd*k_p-(theta_pd0-theta_pd)*k_d;
      data[2]=50-theta_pd*k_p+(theta_pd0-theta_pd)*k_d;
      data[3]=50+theta_pd*k_p-(theta_pd0-theta_pd)*k_d;
      data[4]=50-theta_pd*k_p+(theta_pd0-theta_pd)*k_d;
    }
    else if(x<-1*m){
      data[1]=50+theta_pd*k_p-(theta_pd0-theta_pd)*k_d;
      data[2]=50-theta_pd*k_p+(theta_pd0-theta_pd)*k_d;
      data[3]=50+theta_pd*k_p-(theta_pd0-theta_pd)*k_d;
      data[4]=50-theta_pd*k_p+(theta_pd0-theta_pd)*k_d;
    }
    else{
      data[1]=67;
      data[2]=67;
      data[3]=67;
      data[4]=67;
    }
  }else{
    if(x>n){
      data[1]=20+theta_pd*k_p2-(theta_pd0-theta_pd)*k_d;
      data[2]=20-theta_pd*k_p2+(theta_pd0-theta_pd)*k_d;
      data[3]=20+theta_pd*k_p2-(theta_pd0-theta_pd)*k_d;
      data[4]=20-theta_pd*k_p2+(theta_pd0-theta_pd)*k_d;
    }
    else if(x<-1*n){
      data[1]=20+theta_pd*k_p2-(theta_pd0-theta_pd)*k_d;
      data[2]=20-theta_pd*k_p2+(theta_pd0-theta_pd)*k_d;
      data[3]=20+theta_pd*k_p2-(theta_pd0-theta_pd)*k_d;
      data[4]=20-theta_pd*k_p2+(theta_pd0-theta_pd)*k_d;
    }
    else{
      data[1]=67;
      data[2]=67;
      data[3]=67;
      data[4]=67;
    }
  }
  double theta_pd0=theta_pd;  //D control를 위한 이전 루프 오차변수 저장.
  cout<<theta_pd<<endl;
```

data_integration_edit1.cpp

~/Downloads/data_integration_edit1.cpp 224:55



다음은 PD control인데, target 파란공과 Platform의 중심선이 이루는 각도 차이만큼 error 값으로 두고(코드에서는 theta_pd) 그 reference angle을 0이 되도록 회전을 제어하는 PD control을 하였다. 이때, 기본적으로 움직임을 구현하는 방법에 있어서 직진 속도 벡터에 제어가 들어가는 회전하는 속도 벡터를 합한 값을 data[1]~data[4]에 위 코드처럼 더해 넣어주었다. 이때 theta_pd라는 변수에 저장되는 각도 error만큼의 값을 k_p라는 전역변수 P gain 값에 곱해서 더해지고, 이전 루프의 각도 error인 theta_pd0 라는 변수를 사용해 error의 변화 값인 theta_pd0-theta_pd 값에 k_d라는 전역변수 D gain 값을 곱해서 더해주었다. 여기서 사용한 gain 변수들은 일반적으로 PID gain을 튜닝하는 방법 중 하나인 P gain을 높여 충분히 제어가 되는 범위에서 진동이 일어나는 구간을 찾고, D gain을 높이며 진동이 어느정도 잡히는 구간을 잡는 방법을 사용했다.

PD control을 사용하면서 얻을 수 있었던 장점들은 다음과 같다.

- Target 파란공을 향해 platform이 갈 때, align을 맞추는 과정을 따로 멈춰서 하지 않아도 되기 때문에 시간을 절약할 수 있다.
- 파란공에 다가가면서 계속 제어가 들어가기 때문에 정확한 approach가 가능하다.
- 부드러운 주행이 가능하다.

한편 몇번의 시행착오를 거치며 새로운 조건을 추가하게 되었는데, 바로 target과의 angle error가 일정 크기 이내일 때는 PD control을 하지 않고 최대 속도로 직진하며 다가가는 것이다. 이 조건을

추가하게 된 이유는 두가지가 존재했는데,

1. 제어가 충분히 잘 된 angle error가 발생할 때에도 즉 충분히 파란공을 중앙에 두고 다가가는 과정에서도 계속 제어가 들어가서 조금씩 진동하는 경우가 발생함을 막기 위해.
2. 최대 속도에서는 PD control이 구현되지 않는다. 모터 바퀴에는 최대 속도 상한 제한이 걸려있어, 일정 값 이상의 명령이 들어오면 상한선의 속도만 입력이 된다. 예를 들어, 모터의 최대 상한속도가 67이고, 기본으로 더하는 직진 속도 벡터 값이 60인 경우 회전하는 제어 부분에서의 결과가 data[1], data[3]에는 +10, data[2], data[4]에는 -10만큼이 들어가면 총합은 각각 data[1]=70, data[2]=50, data[3]=70, data[4]=50이 된다. 이때 70은 상한선에 걸려 67로 입력이 되어 실제로 바퀴는 data[1]=67, data[2]=50, data[3]=67, data[4]=50 만큼 각속도가 입력이 들어가, 의도치 않은 제어가 일어날 수 있다. 이러한 점을 방지하기 위해 PD control을 하는 과정에서는 기본적으로 더하는 직진 속도 벡터 값을 적당히 낮게 두어야 하는데 이 값이 실질적으로 곧 target 공에 다가가는 데에 걸리는 시간과 직결된다.

따라서 PD control을 시작과 동시에 작동하고, 차가 어느정도 제어가 되어서 threshold 내로 error가 작아지면 최대속도로 직진하는 구간을 넣어 시간 단축을 꾀하였다.

3.2.2 타겟 선정 함수

이번 장에서는 앞서 언급된 함수들과는 다르게 차량의 운동에 직접적으로 관련된 함수는 아니지만, 그 운동들을 하는 대상 즉 target을 지정하는 함수에 대해서 소개하겠다. 위에서 나온 circular move 운동이나, PD control은 둘 다 pickup할 대상을 기준으로 control을 하게 되는데 이러한 pickup할 파란 공 target을 선정하거나, 원운동을 하며 피할 빨간 공을 선정하고, 마지막에 release를 하기 위해 관측해야 하는 초록공들과 그 공들의 중심좌표를 추출하는 함수에 대해서 다룰 것이다.

data_integration_edit1.cpp

```
//220줄 ~ 294줄 : 타겟 설정 함수들로, pickup할 파란공 타겟을 선정하거나, 피할 빨간공 타겟을 선정하거나, release할 초록공 타겟 지점을 구하는 함수.
//픽업할 파란공을 타겟으로 선정하는 함수.
//a는 시야에 보이는 파란공들의 x좌표들, b는 y좌표들이다.
//일반적으로는 가장 왼쪽에 보이는 파란공을 타겟으로 선정하는데, 더 가까이 있는 파란공과 부딪힐 것 같을때만 그 공을 타겟으로 선정하는 예외를 두었다.
int setting_a_target(vector<float> a,vector<float> b){
  float temp = a[0];
  int t=0;
  for(int i=0; i<a.size(); i++){
    if( temp > a[i]) {
      temp = a[i];                //가장 왼쪽에 있는 공을 타겟으로 선정하는 과정. t는 타겟의 공 번호로 최소 0에서 최대 2까지 있다.
      t=i;
      }
    }if(t==1){                    //가장 왼쪽에 있는 파란공 앞에 파란공이 하나 존재할 경우
      float d = std::abs(a[0]*b[1] - b[0]*a[1]) / sqrtf(a[0]*a[0]+b[1]*b[1]);
      if(d<margin_b){             //타겟이 아닌 파란공과 부딪힐 것 같은 경우
        t=0;                      //타겟을 그 공으로 변경한다.
        cout<<"blue ball 1,2 are too close!"<<endl;
      }else{
        t=1;
      }
    }if(t==2){                    //가장 왼쪽에 있는 파란공 앞에 파란공이 두개 존재할 경우 마찬가지로 두 공들에 대해 위와같은 처리를 해준다.
      float d02=std::abs(a[0]*b[2] - b[0]*a[2]) / sqrtf(a[0]*a[0]+b[2]*b[2]);
      float d12=std::abs(a[1]*b[2] - b[1]*a[2]) / sqrtf(a[1]*a[1]+b[2]*b[2]);
      if(d02<margin_b){
        t=0;
        cout<<"blue ball 1,3 are too close!"<<endl;

      }else if(d12<margin_b){
        t=1;
        cout<<"blue ball 2,3 are too close!"<<endl;

      }else{
        t=2;
      }
    }return t;
  }
```

이 코드는 pickup할 파란 공 target을 선정하는 setting a target이라는 함수에 대한 코드이다. 작동 원리는 웹캠 시야에 들어오는 모든 파란 공들의 x, y 좌표들을 받아오고, 그 좌표들 중 가장 x

좌표 값이 작은 공 즉 가장 왼쪽에 위치한 파란공을 타겟으로 선정한다.

또한 이때 생길 수 있는 잠재적 문제인 가장 왼쪽에 멀리 있는 공을 pickup 하기 위해 가까운 약간 오른쪽에 있는 파란공을 무시하고 부딪힐 수 있는 경우가 발생할 수 있다. 이 경우 차가 target을 pickup 하러 가는 경로 내에 부딪힐 가능성이 있는 파란공이 존재한다면 그 파란공을 target으로 변경해주는 코드를 232줄부터 252줄까지 추가로 작성해 주었다.

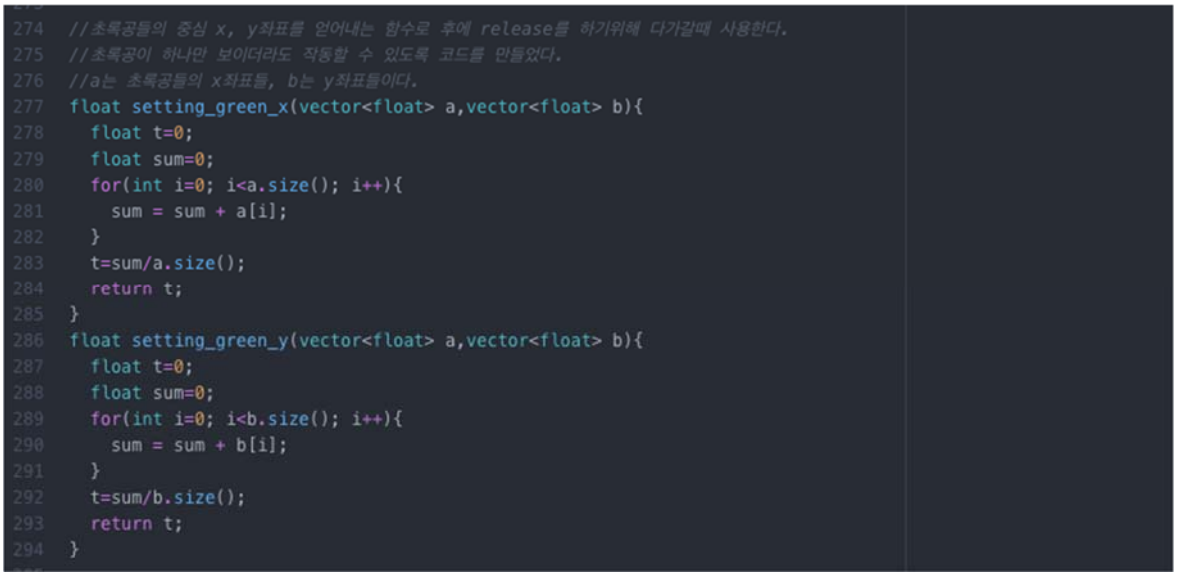
이 함수는 결국 받아들이는 input 값이 시야에 보이는 모든 파란 공들의 x, y좌표이고 output 값은 target이 되는 파란공의 index를 출력해준다. 이 index는 뒤에서 유용하게 사용할 수 있는데, 3.1.1에서 파란공들의 좌표 정보를 담는 vector array인 ball_bX[], ball_bY[]함수를 만들었는데 이 괄호 안에 target의 index를 넣어주면, 원하는 target의 x, y좌표만을 추출해 낼 수 있다. 또한 이 indexing 방식은 팁이 존재하는데, 바로 OpenCV에서 공의 정보를 받아올 때 한번에 여러 공의 정보가 들어오면 y값이 작은 원소부터 y값이 큰 원소 순서대로 index를 매긴다. 즉 파란공이 5개가 화면에 잡힐 경우 가장 아래에 있는 공이 [0]번째 원소가 되고, 가장 위에 있는 공이 [4]번째 원소가 된다. 이러한 팁을 인지하고 있다면 index를 이용한 코딩을 할 때 효율적인 코딩을 할 수 있을 것이다.

data_integration_edit1.cpp

```
256
257 //회피할 빨간공을 타겟으로 선정하는 함수로, 타겟 파란공과의 수직거리가 가장 가까운 빨간공을 선정하는 함수이다.
258 float setting_a_redball_target(float bx, float by, vector<float> rx, vector<float>ry){//rx, ry : red ball
259   int d = 10000; // to avoid returning trash value
260   int l = 0;
261   for(int i = 0; i < rx.size(); i++){ //웹캠 시야의 모든 빨간공에 대해
262     l = abs(bx*ry[i] - by*rx[i]) / sqrtf(bx*bx+by*by); //타겟 파란공과 빨간공의 수직거리를 계산.
263     if(d > l){
264       d = l; // find smallest distance
265       if(sqrt(rx[i]*rx[i]+ry[i]*ry[i])>50){   //빨간공과 파란공의 수직거리가 50보다 크면, 충돌 위험 없다고 판단하고 무시.
266         d=10000;
267       }
268     }
269   }
270   cout<<"d: "<<d<<endl;
271   return d;
272 }
```

위에 나타난 코드는 앞서 설명한 파란공의 target을 선정하는 setting a target 함수를 약간 변형시켜 만든 피해야할 빨간 공의 정보를 가져오는 setting a redball target 함수와, release할 지점의 좌표를 시야에서 보이는 초록공의 좌표들 정보로 구하는 setting green x, setting green y 함수이다. 먼저 setting a redball target 함수는 파란 공 target을 정하는 함수와는 약간 다르게, input에 시야에 보이는 모든 빨간 공 정보와 함께 현재 target 파란 공 정보 또한 입력을 받는다. 그리하여 시야에 보이는 모든 빨간 공과 target 파란 공 사이의 수평 거리(platform이 파란공을 중심으로 바라볼 때, 빨간 공의 x축 좌표 값과 같다)를 계산하고 이 수평 거리가 가장 짧은 빨간 공이 피해야 할 공 1순위가 되고, 이 공의 수평 거리를 출력하도록 한다.

뒤에서 이 함수가 어떠한 방식으로 쓰이냐면 웹캠 시야의 빨간 공들 중 target 공과의 가장 작은 수평거리가 platform의 pickup region의 절반 길이보다 작은 경우, 빨간 공을 피해야 한다고 인식해 원운동을 시작하도록 하는 avoid red ball decision의 factor가 된다.



```
//초록공들의 중심 x, y좌표를 얻어내는 함수로 후에 release를 하기위해 다가갈때 사용한다.
//초록공이 하나만 보이더라도 작동할 수 있도록 코드를 만들었다.
//a는 초록공들의 x좌표들, b는 y좌표들이다.
float setting_green_x(vector<float> a,vector<float> b){
  float t=0;
  float sum=0;
  for(int i=0; i<a.size(); i++){
    sum = sum + a[i];
  }
  t=sum/a.size();
  return t;
}
float setting_green_y(vector<float> a,vector<float> b){
  float t=0;
  float sum=0;
  for(int i=0; i<b.size(); i++){
    sum = sum + b[i];
  }
  t=sum/b.size();
  return t;
}
```

277줄 ~ 294줄에 나오는 setting green x, setting green y 함수는 입력 값으로 각각 웹캠 시야에 보이는 모든 초록공들의 x좌표들과 y좌표들의 벡터 array를 입력 받아서, 좌표들의 평균값을 구해서 출력하는 함수를 만들었다. 일반적으로 한번에 초록공은 두 개씩 인식할 수 있을 것 같았지만 혹시 초록공을 한 개만 인식하여 생길 수 있는 문제를 미리 방지하고자 공의 개수를 처음부터 2개라고 단정짓고 만들지 않았다. 따라서 초록공이 1개가 인식되면 그 공의 중심좌표가, 공이 2개가 인식되면 공들의 중점의 중심좌표가 출력된다.

3.3 State

State는 Platform이 수행해야할 미션을 작은 task들로 분할해 놓은 것으로, 중추 state가 되는 state0, state3등에서 State decision에서 결정된 행동에 맞는 state로 이동해 현재 platform이 필요로 하는 행동을 수행하게 된다. State들에 대한 설명은 Data integration node에 주석을 꽤 detail하게 적었기 때문에 주석과 함께 읽으면 더욱 이해에 도움이 될 것이다.

State는 크게 2종류로 구성이 되어있는데, Pickup blade의 state를 구분하는 pickup state와 wheel motor의 state를 구분하는 state0,1,2,3,4,5,6,7,8,9,10이 존재한다.

기능별로는 크게 4종류로 구분할 수 있다. 뒤에서는 기능별 State 들에 대한 코드 설명을 나열하였다.

- State Decision & PD control(State0)
- Pick up(Pickup state & State2)
- Avoid red ball(State1 & State6 / State 7,8,9,10 : avoid red ball at Releasing)
- Release(State 3 & State 4 & State 5)

3.3.1 State Decision & PD control (State0)

State0은 웹캠을 통해 얻은 정보로부터 State Decision을 하고, target 파란공을 setting a target 함수를 통해 선정한 뒤 이를 향해 PD control을 하는 State이다.

기본적으로는 다음 타겟 파란공을 찾고, 그 공을 향해 PD control을 하며 다가가는 도중 빨간공을 피해야 한다고 판단되면 avoid red ball을 하는 state인 State=1로 변경한다.

혹은 target 파란공까지 가까이 도달할 경우, PD control을 멈추고 Pickup state로 변경하기 위해 Pickup state=1, State=2로 변경한다.

또한 target 파란공을 찾기 못하거나 픽업을 3번 이상 했다고 판단되면 release state인 state3으로 넘어간다.

data_integration_edit1.cpp

```
//state0은 state decision과 PD control을 하는 구간이다.
//기본적으로는 다음 타겟 파란공을 찾고, 그 공에 PD control을 하며 다가간다.
//이 과정에서 빨간공을 피해야 한다고 판단되면 state=1로 변경해 avoid redball을 하는 state로 변경하고,
//타겟 파란공을 못찾거나 픽업을 몇번 이상 했다고 판단되면 release state로 넘어간다.
  if (state==0){
    data[8]=30;
    cout<<"state = 0(state decision & moving around)"<<endl;
    cout<<"num_pickup : "<<num_pickup<<endl;
    cout<<"pickup_time :"<<pickup_time<<endl;

    if(num_pickup>2){ //현재는 pickup을 세번 이상 했다고 판단될 때 바로 release하는 state로 보내주는 역할을 한다.
      state=3;
    }
    if(position->img_bx.size()==0){   /////blueballs on the cam
      align(40);
    }
    else{
      theta_release=0;
      int t = setting_a_target(position->img_bx, position->img_by); //픽업할 target 파란공을 정하는 함수.
      cout<<"target ball x position : "<<ball_bX[t]<<endl;
      cout<<"target ball distance : "<<blue_ball_distance[t]<<endl;
      cout<<t+1<<"  is target blue ball"<<endl;

      if(blue_ball_distance[t]>40){ //타겟 파란공과의 거리가 40cm보다 멀때,
        cout<<"here!"<<endl;
        int d=setting_a_redball_target(ball_bX[t], ball_bY[t], position->img_rx, position->img_ry); //위험한 빨간공이 있는지 확인한다.
        if(d>margin){                                  //타겟 파란공과 가장 가까운 빨간공과의 x축 거리가 margin보다 크면 부딪힐 일이 없으므로
          PD_control(ball_bX[t],ball_bY[t],7,5);  //PD control로 타겟 파란공에 다가간다.
          cout<<"PD controlling"<<endl;
        }
        else{       //타겟 파란공과 가장 가까운 빨간공의 x축 거리가 margin보다 작아 부딪힐 위험이 있으므로
          state=1;  //avoid red ball 동작을 하는 state 1로 변경해준다.
        }
      }
      else{         //타겟 파란공과의 거리가 40cm보다 가까울 때,
        pickup_state=1;   //블레이드를 pickup state=1 로 하여 pickup mode로 진입하게 하고
        state=2;          //이때의 wheel mode 또한 state=2로 하여 pickup을 시작한다.
      }
    }

  }
```

이 State에서 본 조의 Path concept가 나타나는데, 435, 436번째 줄을 보면 웹캠에 파란공이 없을 경우 시계방향으로 파란공이 나타날 때까지 회전하도록 하였다. 또한 440번째 줄에서 파란공이 웹캠에 인식되면 3.2.2 장에서 언급한 setting a target 함수를 이용해 가장 왼쪽에 있는 파란공을 target으로 선정하였다. 이 결과로, 가장 왼쪽에 있는 파란공을 target으로 선정하고 그 공을 pickup 한 후 다시 다음공을 찾기 위해 시계방향으로 회전하는 패턴을 보인다. 즉 전체 경기장 Field를 큰 시계방향으로 회전하며 경로의 중복을 막는 알고리즘을 사용한 것과 같은 결과를 낸다.

3.3.2 Pick up (Pickup state1 & State2)

Pick up state는 pickup motor와 wheel motor들을 구동 시킴으로써 blade와 wheel을 회전시켜 State 0에서 가까이 다가간 target 파란공을 향해 더 움직이면서 pickup을 하는 task를 수행하는 state이다.

가장 큰 특징은 pick up시 소요되는 시간을 최소화하기 위해 차는 멈추지 않고 pickup을 진행하며 이를 위해서는 동시에 독립적으로 pickup motor와 wheel motor의 제어가 필요하다. 이를 충족시키기 위해 Pick up state에서는 Blade를 제어하는 Pickup state1과 그때의 wheel을 제어하는 State2를 작동시킨다.

먼저 Pickup state1은 pickup시 chain과 blade 즉 pickup module을 제어하는 구간이다. Pickup state는 0과 1로 이뤄져 있으며 1일때 blade를 움직이게 해 pickup을 하도록 한다. 이 과정은 state decision 파트인 state가 0인 상태에서 PD control로 충분히 target 파란공과 가까워져 pickup을 해야겠다고 판단될 경우 작동하게 된다. 가장 큰 특징은 pickup motor는 이 과정에서 position control이 가능해, 정확한 제어가 가능하다는 점이다.

data_integration_edit1.cpp

```
343
344   //pickup state는 pickup시 chain과 blade 즉 pickup module을 제어하는 구간이다.
345   //pickup state는 0과 1일때로 이루어져있으며, 1일때 blade를 움직이게 해 pickup을 한다.
346   //이 과정은 state decision 파트인 state가 0일때 pickup을 해야겠다고 판단될경우 작동한다.
347   //position control을 기반으로 해 정확한 제어가 가능하다.
348   if (pickup_state==1){
349     cout<<"picking up"<<endl;
350     data[7]=1;
351     if (pickup_time<t_p1){
352       data[6]=350;
353       cout<<"blade-up"<<endl;
354     }
355     else if (pickup_time<t_p1+t_d1){
356       data[6]=0;
357     }else if (pickup_time<t_p1+t_d1+1){
358       data[6]=0;
359       state=0;        //중요한 trigger인데, 블레이드가 닫히는 과정에서 공이 먼빠져나갈 수 있는 시점에서
360                       //wheel mode와 decision mode를 담당하는 state를 다시 state=0으로 변경하여 블레이드는 공을 마저 storage에 넣고 바퀴는 다음공을 찾으러 간다.
361       num_pickup=num_pickup+1;  //이 시점에서 총 pickup한 횟수를 누적해준다.
362     }else if (pickup_time<t_p1+t_d1+t_d2){
363       data[6]=0;
364     }
365     else if (pickup_time<t_p1+t_d1+t_d2+t_d3){
366       data[6]=310;
367     }
368     else{
369       pickup_time=0;
370       pickup_state=0;
371     }
372     pickup_time++;
373
374   }
```

350번째 줄에서 data[7]=1로 정해준 이유는 pickup motor의 제어 방법을 position control을 택하기 위하기 때문이다. 하지만 pickup_time이라는 시간변수를 만들어서 마치 time control을 하는 것처럼 보일 수 있는데 이는 몇 루프동안 제어를 하는 것이 아니고, position control이지만 모터의 회전 각속도의 상한선이 정해져 있어 한 loop만에 원하는 position 까지의 이동이 구현되지 않을 수 있다. 이러한 잠재적 문제를 해결하기 위해 넉넉하게 5~10개정도의 loop동안 position control이 일어나도록 하였고, 실제로 잘 작동함을 확인했다.

359번째 줄에 있는 state=0으로 해주는 코드는 중요한 trigger중 하나로 pickup에 소요되는 시간을 최소화하기 위해 블레이드가 닫혀 공을 platform의 외부와 차단시키는 이후부터는 pickup motor는 마저 공을 집어넣는 일을 하되, wheel을 제어하는 state는 다음 공을 찾거나 다음 decision을 하는 state0으로 바로 보내주어 시간 최적화를 구현하였다. 이해하기 쉽게 예를 들어 설명하자면, platform이 공을 pickup하는 일을 사람이 떡을 먹는 과정에 대응시켜보자. 사람이 여러 개의 떡을 하나씩 빨리 먹기 위해서 첫번째 떡을 입에 가둔 후, 떡을 삼킨 뒤에 다음 떡을 찾는 방법보다 첫번째 떡을 입에 가둔 후, 떡을 삼키기 시작하는 동시에 다음 떡을 찾는 방법이 떡을 삼키는 시간만큼 단축시킬 수 있을 것이다. 이는 대부분의 경우에서는 안전하게 시간을 절약할 수 있었고, 가끔 발생하는 특이케이스에 대해서는 state2에서 해결책을 제시하였다. 이는 뒤에서 설명한다.

다음으로는 pickup시 Pickup stata1과 동시에 일어나는 State2에 대한 설명이다. State2는 pickup시 wheel을 제어하는 구간으로, 이 과정도 마찬가지로 대부분의 경우 State0에서 pickup의 필요를 느낄 때 Pickup state1과 동시에 시작한다. 이 짧은 pickup하는 구간 동안, target 파란공이 너무 가까워 웹캠으로 볼 수 없어 바퀴는 멈추지 않고 feedforward로 직진하게 하였다. 하지만 이러한 openloop는 본 platform의 pickup module이 꽤 robust하여서 항상 안전하게 잘 작동하는 모습을 보였다.

data_integration_edit1.cpp

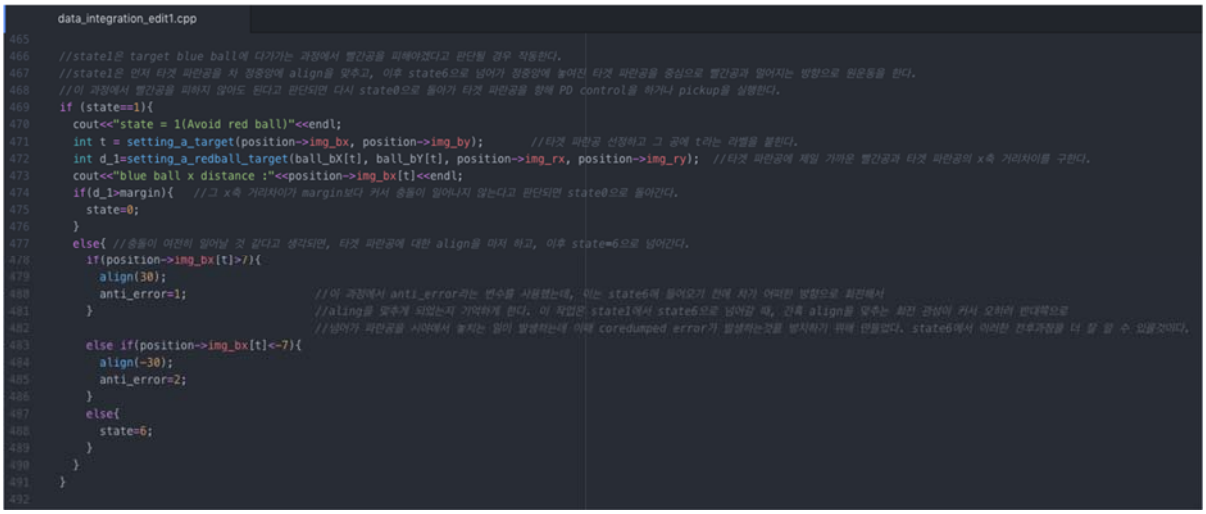
```
378  //state2는 pickup시 wheel을 제어하는 구간이다.
379  //이 과정은 state decision 파트인 state가 0일때 pickup을 해야겠다고 판단될경우 작동한다.
380  //이 짧은 pickup하는 구간동안, 타겟 공이 너무 가까워 웹캠으로 볼 수 없어 바퀴는 멈추지 않고 feedforward로 직진한다.
381  //하지만 본 platform의 pickup module은 꽤 robust해, 이정도의 openloop는 항상 잘 작동하였다.
382  if(state==2){
383
384      cout<<"state2_time :"<<state2_time<<endl;
385      cout<<"pickup_time :"<<pickup_time<<endl;
386
387      cout<<"state = 2(pick up wheel mode)"<<endl;
388      state2_time++;
389
390      if(pickup_time - state2_time >7){   //파란공을 언덕이 바로 pickup해야할 때 생길 수 있는 특이 케이스 문제를 해결하기 위한 코드이다.
391                                          //블레이드가 다음 파란공을 pickup을 할 준비를 끝마칠 때 까지는 느리게 뒤로 가거나 align을 맞추는 등 바퀴도 준비를 한다.
392        if(pickup_time<104){
393          movef(-20);
394          if(ball_bX[0]>9){
395            align(20);
396          }else if(ball_bX[0]<-9){
397            align(-20);
398          }
399
400          cout<<"ready for next pick-up(too close blue balls)"<<endl;
401        }else{
402          state2_time=0;
403          pickup_time=0;
404          state=0;
405        }
406      }else{
407        movef(60);                    //특이케이스가 아닌 경우 pickup시 바퀴는 pickup state에서 state를 0으로 바꾸기 전까지 직진운동 한다.
408
409      }
410      if(state2_time>30){
411        state=0;
412        state2_time=0;
413
414      }
415      cout<<"pickup_time :"<<pickup_time<<endl;
416      cout<<"state2_time :"<<state2_time<<endl;
417
418  }
419
```

이때 390 번째 줄부터 398 번째 줄 까지는 특이 케이스에 대한 plan 으로, 앞에서 설명한 Pickup state1 에서 설정한 시간 최적화 방법에 의해 생기는 특이 케이스의 경우이다. 만약 첫번째 공을 pickup하자마자 blade는 공을 storage로 집어넣기 시작하고(떡을 삼키기 시작하고) 다음 target 공을 찾으러 state0 으로 보냈는데(다음 떡을 찾으러 돌아봤는데) 바로 또 pickup 을 시작해야 하는 경우 즉 공이 연달아 있는 경우가 발생할 수 있다. 이러한 경우에는 pickup 의 openloop 가 실행되어야

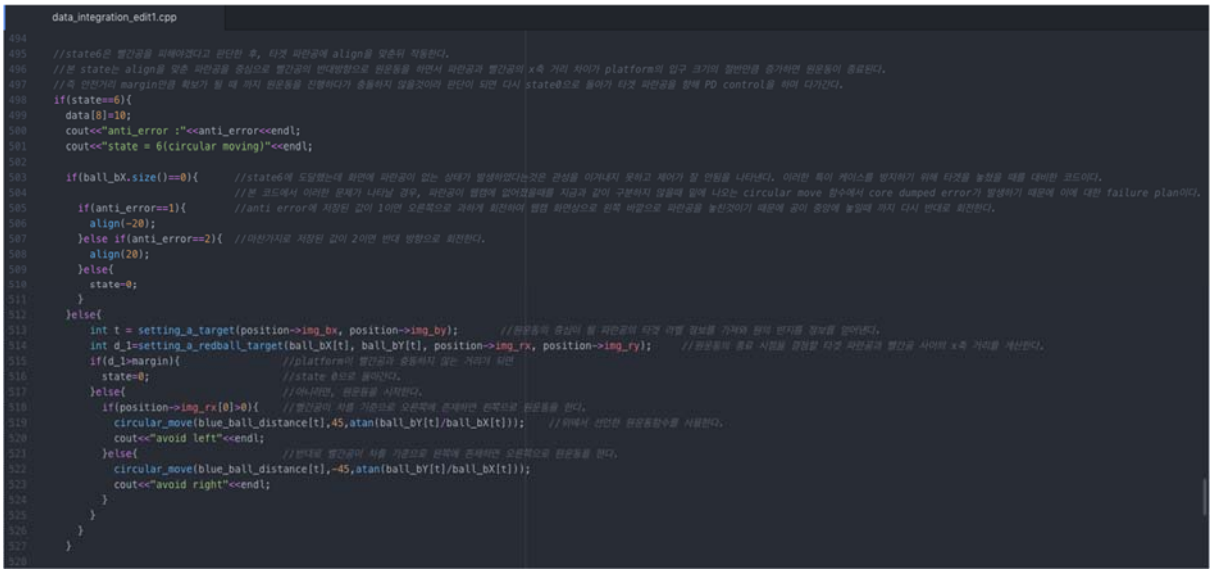
하는데 아직 blade 는 platform 내부의 공을 집어넣는 중이라 pickup 할 준비가 되어있지 않은 상황이다. (떡을 삼키는 중이라 바로 다음 떡을 먹는 명령을 수행하지 못한다) 이러한 경우를 대비해 390 번째 줄은 Pickup state1 의 동기화를 나타내는 변수 pickup time 과 state2 의 동기화를 나타내는 변수 state2 time 의 값이 같지 않다면 지금 상태는 pickup 을 하자 마자 다음 pickup 명령이 들어왔다는 것을 인지할 수 있다. 따라서 이와 같은 경우 다음 pickup 을 시작할 수 있는 때까지 wheel 은 단순 open loop 로 들어가지 않고 blade 를 기다렸다가 다음 pickup 을 할 수 있을 때 다시 open loop 을 실행하도록 하여 문제를 해결할 수 있었다. (떡을 다 삼킬 때까지 기다리는 장치를 만듦)

3.3.3 Avoid red ball (State1 & State6 / State 7,8,9,10 : avoid red ball at Releasing)

Avoid red ball state는 타겟(파란 공 또는 바구니)을 중심으로 원운동을 하여 빨간 공을 피하는 State이다. 타겟이 파란 공일 때는 State1과 State6을 사용하는데 State1은 State0에서 파란 공을 향해 움직일 때 경로 상에 빨간 공이 있으면 원 운동하기 전 준비 단계로 로봇의 중심을 파란 공과 일직선이 되도록 만드는 State이다. 중심에 맞게 되면 State6으로 넘어가서 빨간 공이 경로 상에서 없어질 때까지 원운동을 하게 된다. 만약 빨간 공이 타겟 파란 공에 다가가기 위한 경로 상에서 없어지면 State0으로 돌아가 PD control로 타겟에 다가가게 된다.



State1에서 setting_a_target 함수를 이용하여 타겟 파란 공을 정하고 setting_a_redball_target으로 피해야 하는 빨간 공과 타겟 파란 공 사이 거리를 d_1로 정한다. 만약 d_1이 차폭/2(margin)보다 크다면 빨간 공을 피할 필요가 없다는 뜻이므로 State0으로 이동하여 파란 공을 향해 다가가게 된다. 반대로 d_1이 차폭/2보다 작다면 원운동을 하여 빨간 공을 피하기 전에 타겟 파란 공과 로봇의 중심을 일직선으로 맞춘다. 중심을 맞출 때 회전 운동을 통해 중심을 맞추는데 회전을 너무 많이 하여 타겟 파란 공이 웹캠 시야에서 벗어나면 core_dumped라는 에러가 떳다. 이를 방지하게 위해 anti_error라는 변수를 이용하여 시야에서 벗어나면 회전했던 반대 방향으로 회전하여 타겟이 다시 시야에 들어오게 만들었다. 이는 State6까지 연결되어 사용된다.

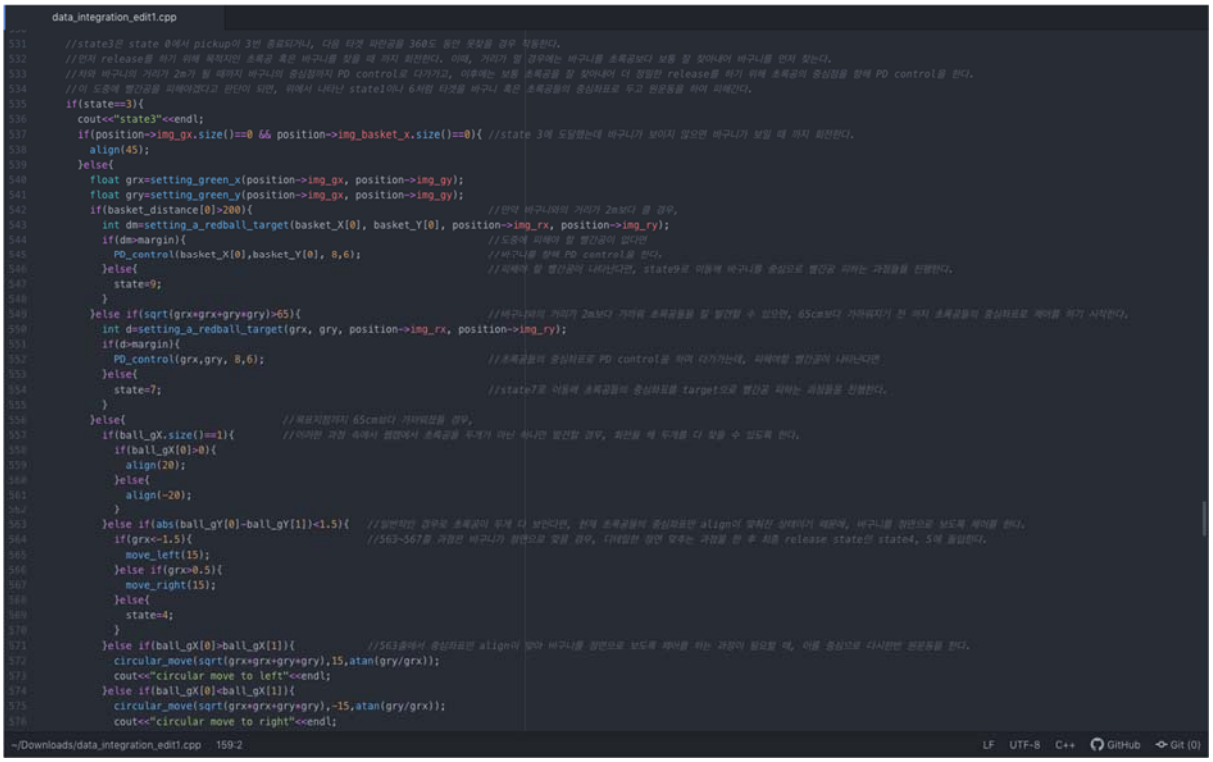


로봇의 중심과 파란 공의 중심이 맞으면 State6으로 넘어가 원운동을 하게 된다. 이때 원운동 경로를 최소화하기 위해 빨간 공이 왼쪽에 있으면 오른쪽으로, 오른쪽에 위치하면 왼쪽으로 원운동을 하도록 작성하였다. 원운동을 하다가 빨간 공에서 벗어나면 State0으로 넘어가 파란 공을 향해 다가가 pickup을 하게 된다.

타겟이 파란 공이 아니라 바구니일 경우에는 setting_a_target 함수 대신 setting_green_x와 setting_green_y 함수, 또는 basket_X[0](바구니의 x좌표)와 basket_Y[0](바구니의 y좌표)를 이용하여 타겟을 잡는 것 이외에 차이점은 없다.

3.3.4 Release (State3 & State4 & State5)

Release는 파란 공 3개를 다 먹었을 때 바구니를 찾아가 공을 바구니에 넣는 전체 State로 State 3, 4, 5가 순차적으로 연결되어 있다.



State3에 들어오면 우선 바구니나 초록 공이 보일 때까지 시계방향으로 돌게 된다. 바구니나 초록 공이 보이면 바구니와의 거리가 65cm가 될 때까지 PD_control 함수를 이용하여 다가가게 된다. 만약 경로상에 빨간 공이 있다면 앞에서 설명한 avoid red ball sate로 빨간 공을 피하게 된다. 초록 공이 멀리 있을 때 웹캠 상에 잘 잡히지 않는 문제 때문에 바구니의 x, y좌표를 메시지로 받았다. 따라서 거리가 멀 때(>200cm)는 바구니의 좌표를 타겟으로 하여 다가가며 가까워지면 초록 공이 잘 잡히기 때문에 초록 공의 중점을 타겟으로 한다. 65cm에 다가왔을 때 초록 공이 하나만 보인다면 중점을 잡기 위해 두개가 보이도록 회전을 한다. 두 개의 초록 공이 모두 보이면 바구니의 정면과 로봇이 평행하도록 맞춰야 한다. 이때 빨간 공을 피할 때 사용했던 circular_move 함수를 이용하여 바구니의 중앙을 중심으로 원운동을 하여 평행을 맞춘다. 이후에 정확히 바구니의 정면을 보기 위해 좌, 우 운동을 통해 한번 더 로봇의 위치를 정확히 조정한다. 원 운동과 좌, 우 운동을 통해 로봇의 위치가 정확히 바구니 정면에 있으면 State4로 넘어가게 된다.

State4로 넘어오면 느리게 회전을 하며 정밀하게 위치를 한번 더 조정한다. 정확히 조정이 되면 바구니와의 거리가 40cm가 될 때까지 앞으로 움직이며 최종 openloop release 단계인 State5로 넘어간다.

마지막으로 State5에서는 바구니 바로 앞까지 openloop으로 직진하며 주차를 한다. 이때 로봇 앞에 아크릴 판으로 만든 가이드가 있기 때문에 앞으로 움직여야 하는 실제 거리보다 마진을 추가하여 openloop이지만 최대한 Robust하게 만들었다. 로봇이 주차가 되면 pick up 모터를 반대로 회전시켜 파란 공을 바구니 속으로 넣게 된다. Release할 때는 체인이 약 2바퀴 반정도 돌아야 하기 때문에 1바퀴의 운동 제약이 있는 position control을 사용할 수 없다. 따라서 velocity control을 이용하여 openloop으로 release를 하게 된다.

4. coding schedules

코딩은 아래 일정에 맞게 진행되었다.

1~6 주차	ROS 기초(node, publish & subscribe 등) 및 C++ 공부	
7~8 주차	전략 결정, 알고리즘 작성	
9~10 주차	데모 2 코드 작성 및 ppt 작성	
11 주차	알고리즘에 맞춰 각 state 코딩	파란 공 타겟 정하기, pick-up state
12 주차		빨간 공 피하기, release state
13 주차	각 state 마다 부족한 부분 개선하기	
14 주차	시간 단축을 위한 코드 최적화	

코딩 스케줄에서 가장 중요한 부분은 알고리즘 작성이었다. 본 조는 알고리즘을 작성할 때 거의 모든 경우의 수를 생각하며 최선의 전략을 도출하기 위해 많은 시간과 노력을 투자하였다. 그런 결과 7~10 주차까지는 다른 조에 비해 코드 작성 속도가 느린 편이었지만 12 주차에는 모든 코드가 완성되어 진행 속도가 빠른 편에 속하였다. 이는 알고리즘 작성 단계에서 최악의 경우까지 모두 고려하였기 때문이다. 또한 state 사용 역시 빠른 코드 완성에 도움을 주었다. 하지만 12 주차에 빠르게 코드를 완성한 것에 비해 하드웨어가 최종 발표 주차인 14 주차까지 서스펜션을 다는 등 변화를 주어 연습이 많이 필요한 13 주차나 14 주차에 ROS 코드를 고치는데 많은 시간을 투자하지 못해 아쉬웠다. 내년에는 초반에 하드웨어를 완벽히 만드는 것을 목표로 하고 알고리즘을 최대한 정밀하게 작성을 하기를 추천한다.