로봇 센서 소프트웨어

Turtlebot3 Navigation Project

기계로봇에너지공학과

2016110576 강재원

목차

Ⅰ. 코드의 기본개념

모바일 로봇을 navigation 하는 데에는 수많은 방법들이 존재한다. 모바일 로봇의 navigation은 보통 크게 3가지의 단계로 나뉘는데, 센서로부터 받을 값들을 가지고 모바일 로봇 주변 환경에 대한 지도를 그리는 mapping, 지도 상에서 로봇의 현재 위치를 파악하는 localization, 그리고 로봇이 현재 위치에서 목표지점으로 어떻게 이동할 것인지 계획하는 path planning이 바로 그것이다. 이 중 path planning에는 지도가 주어진 상태에서 전체경로를 생성하는 global path planning과 센서값들로부터 환경을 인지하며 목표점에 도달하는 local pathplanning이 있다.

Global path planning은 가시도그래프(V-graph), 보르노이다이어그램(voronoi-diagram), 구배법, A*알고리즘 등 다양한 방법들이 연구되었으나 경로이탈 또는 새로운 장애물을 발견할 시에 다시 새로운 경로를 생성해야하므로 실시간으로 적용하기엔 문제점이 있다.

반면에 DWA(dynamic window approach), EB(elastic band), VFH(vector field histogram), Potential Field 등과 같은 local pathplanning 방법들의 경우, 센서를 기반으로 실시간으로 주변의 장애물을 인식하여 목표점까지 이동하는 방법이다. Local pathplanning 방법은 새로운 장애물 발견시 실시간 처리가 가능하고, 로봇의 동력학 특성을 고려가 가능한 장점이 있지만, 지역적인 제한으로 인해 지역최소문제(local minima)에 빠져 목표점에 도달하지 못하는 경우가 발생할 수 있다. 본 프로젝트에서는 전체적인 맵을 미리 알 수 없었기 때문에, global path planning은 불가능하였고, mapping, localization, local pathplanning까지 진행하였다.

다음의 단락들은 필자가 사용한 코드가 어떻게 작동하는지 설명한다.

1. Grid map

가장 먼저, 라이다로부터 주변의 장애물까지의 각도, 그리고 거리 값을 배열로 입력받은 후에는 그 값들을 이용해 격자 지도(grid map)를 만들어주었다. 이 격자지도는 상대좌표로, 로봇의 현재위치를 원점으로 놓았다.

그 이후에는, 장애물들 주위에 패딩(padding)을 만들어주었다. 즉, 장애물에게 두께를 부여하는 것으로, 장애물 주위로 몇 cm의 두께를 가진 가상의 장애물이 존재한다고 터틀봇에게 인식시켜주는 것이다. 패딩의 두께를 크게 주면 그만큼 장애물과 부딪힐 확률이 적어지지만, 로봇이 충분히 지나갈 수 있는 좁은 통로를 지나갈 수 없다고 판단하는 결과를 낳기도 했다. 따라서 패딩의 두께는 여러가지의 맵에서 실험을 통해 선정해야했다.

마지막으로, 터틀봇의 크기로는 통과할 수 없는 통로를 인지시키고, 국부최소문제를 조금이나마 해소하고자 가상의 장애물들을 만들어주었다.

2. DWA(Dynamic Window Approach)

격자지도를 만든 이후에는, 이를 이용하여 local path planning을 진행하였다. 필자는 여러가지의 local path planning 방법들 중, DWA를 선정하여 코딩하였다.

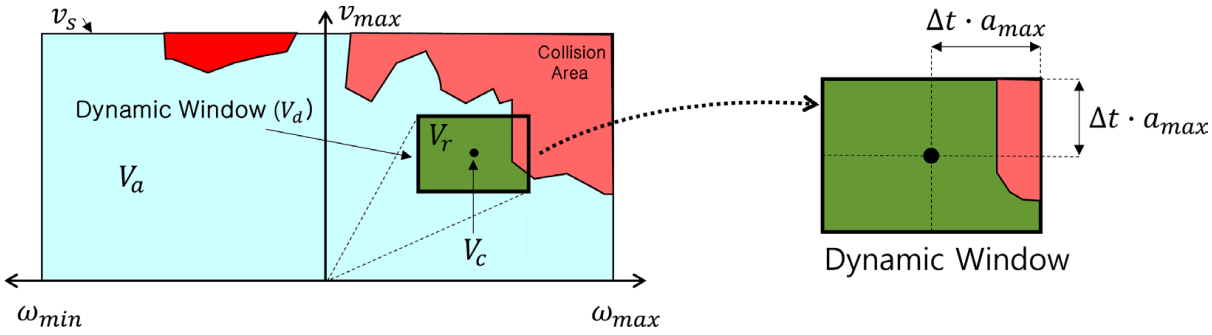

그림 1 로봇의 Dynamic Window

DWA는 현재 로봇의 속도에서, 로봇의 최대 가속도를 고려하여 Δt 시간 이후에 가질수 있는 로봇의 각속도/선속도의 범위를 설정한다. 이를 동적 창(Dynamic Window)라 하고, 이 동적 창 내에서 모든 각속도/선속도들에 대해 목적함수를 계산한다. 이후, 이 속도들 중 가장 목적함수 값이 높은 속도를 선택한다.

DWA의 목적함수는 다음과 같이 방향함수, 속도함수, 그리고 충돌함수, 이렇게 세 가지의 함수들과 각각의 가중치들을 통해 연산된다.

$$O = \gamma_{head} w_{head} + \gamma_{speed} w_{speed} + \gamma_{clear} w_{clear}$$

먼저 방향함수를 살펴보자. 방향함수는 현재 로봇의 위치에서 목표지점의 방향을 수치화한 함수이다. 아래의 그림에서 p_c는 현재 터틀봇의 위치이고, p_t는 목표지점, p_n은 특정 선속도와 각속도를 가지고 이동했을 때 Δt 이후의 예상지점이다. DWA의 방향함수는 p_t 와 p_n 사이의 각 θ를 계산한다. θ가 0 인경우에는 목표지점과 로봇의 방향이 일치하므로 최대값인 1을 부여하고, θ가 ±π인 경우, 즉 목표지점과 로봇의 방향이 반대일 경우 0을 부여한다.

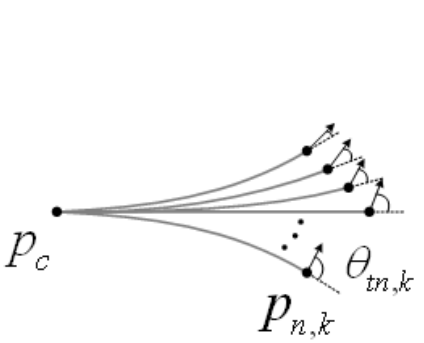

그림 2 DWA의 방향함수

다음으로 속도함수를 살펴보자. 모바일 로봇이 같은 경로를 움직일 때에는 그 속도가 빠를수록 목표지점까지 도달하는 시간이 빨라진다. DWA에서는 속도가 빠를수록 값을 속도함수에 부여한다. 필자는 로봇의 속도가 0일 때 0, 임의로 지정한 한계속도인 0.1 일때 1의 값을 갖도록 하였다.

마지막으로 충돌함수를 살펴보자. 충돌함수는 로봇의 Δt 이후의 예상지점에서 가장 가까운 장애물까지의 거리를 수치화한 함수이다. 장애물까지의 거리가 아닌 장애물까지의 충돌 시간을 수치화하는 경우로 있었으나, 필자는 단순하게 장애물까지의 거리를 사용하였다.

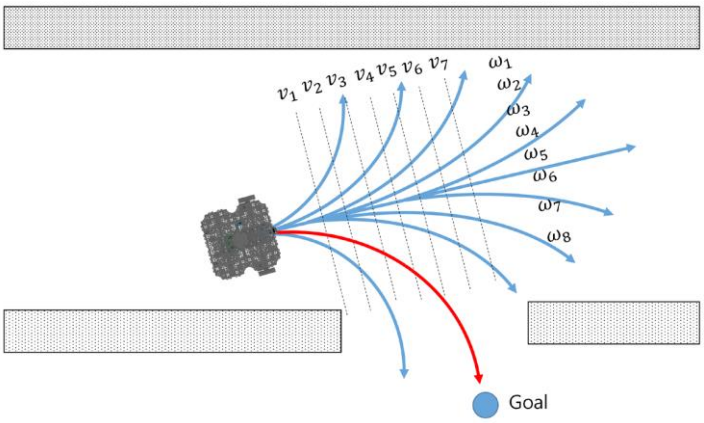

그림 3 DWA의 속도탐색

Ⅱ. 코드의 세부내용

1. 코드 전문

본 프로젝트에서 사용한 코드의 전문은 다음과 같다. package.xml 파일과 Cmakelist.txt 파일은 코드의 길이 상 생략하도록 하겠다. 또한, 다음 장의 노드 외에도 터틀봇을 주행 중에 임의로 정지시키기 위한 emergency break 노드가 존재하는데, 이 또한 생략하였다.

①

```
//ros basic header
#include "ros/ros.h"
//message header
#include "sensor_msgs/LaserScan.h"
#include "geometry_msgs/Twist.h"
#include "std_msgs/Bool.h"
#include "nav_msgs/Odometry.h"
#include "navigation/stopmsg.h"
//math header&constant
#include "math.h"
#define PI acos(-1)
//declare msg,pub as a static so that we can use it outside of the main function
static geometry_msgs::Twist motorvalue;
static std_msgs::Bool motorpower;
static ros::Publisher path_planning_motorpub;
static ros::Publisher path_planning_motorpowerpub;


static double turtlebot_width = 0.138, turtlebot_length = 0.178;//actually 0.178
static double lidarxy[2][360];//point cloud, NOT FILTERED
static double x = 0, y = 0, q = 0; //where am I now
static double destination_x = 5.0, destination_y = 0;//###Change your INPUT at here ###
ros::Duration duration(0.05);
double go = 1;


void msgCALLBACK(const sensor_msgs::LaserScan::ConstPtr& scan)
{
```

②

```
	motorpower.data = 1 * int(go);
	//######### making GRID MAP #########
	int map_size = 72;    //%4 should be 0
	double grid_size = 0.01;
	int turtlebot_width_grid = int(turtlebot_width / 2 / grid_size) * 2 + 2;
	int turtlebot_length_grid = int(turtlebot_length / 2 / grid_size) * 2 + 2;
	double map_length = double(map_size) * grid_size;
	char map[map_size][map_size];
	//initializing the map
	for (int i = 0; i < map_size; i++)
	{
		for (int k = 0; k < map_size; k++)
		{
			map[i][k] = ' ';
		}
	}
	for (int i = 0; i < 360; i++)//assumming the forward lidar angle is 0
	{
		lidarxy[0][i] = scan->ranges[i] * cos(double((i + 180) % 360) *PI / 180);
		lidarxy[1][i] = scan->ranges[i] * sin(double((i + 180) % 360)*PI / 180);
		if (fabs(lidarxy[1][i]) < map_length / 2 && fabs(lidarxy[0][i] - map_length / 4) < map_length / 2)
			//if the obstacle locates inside of the rectengular map
		{
			if (!(lidarxy[0][i] == 0 && lidarxy[1][i] == 0))//filtering 0 values
			{
				int grid_x = (map_size - 1) - int((lidarxy[0][i] + map_length / 4) / grid_size);
				int grid_y = int((map_length / 2 - lidarxy[1][i]) / grid_size);
				if(grid_x > 2 && grid_x < map_size - 2 && grid_y > 2 && grid_y < map_size - 2)
				{
					for(int n = 0; n < 25; n++)
					{
						map[grid_x + (n / 5) - 2][grid_y + (n % 5) - 2] = 'x';
					}
				}
			}
			//grid with 'x' is an obstacle grid
		}
	}
```

③

```
	//################  making virtual obstacles  ############
	for (int i = 0; i < map_size; i++)
	{
		for (int k = 0; k < map_size; k++)
		{
			if (map[i][k] == 'x')
			{
				for (int xy = 0; xy < 2; xy++)
				{
					int blank = 0, nearest_d = 0;
					for (int n = 1; n < map_size - (k * (1 - xy) + i * xy + 1); n++)
					{
						if (map[i + n * xy][k + n * (1 - xy)] == ' ')
							blank++;
```

```
                                                else
                                                {
                                                        nearest_d = n;
                                                        break;
                                                }
                                        }
                                        if (blank >= 5 && blank <= 15)
                                        {
                                                for (int n = 1; n < nearest_d; n++)
                                                {
                                                        map[i + n * xy][k + n * (1 - xy)] = '-';
                                                }
                                        }
                                }
                        }
                }
        }

        //######### Dynamic Window Approach #########
④       double v_max = 0.1, w_max = 2.84, a_max = 0.02;//w : counterclockwise is +
        double w_head, w_speed, w_clear;
        double caution_distance = 0.7, danger_distance = 0.07;
        double g_head = 0.1, g_speed = 0.9, g_clear = 0.4;
        double pn_x, pn_y, dt = 2;
        double pn_x_r, pn_y_r;
        int dwa_size = 21;//0 value is important, so let's pick odd num
        double O_max = 0, TEMP, v_dwa, w_dwa;
        double head_check, speed_check, clear_check;

⑤      for (int a = 0; a < dwa_size; a++)
        {
                for (int b = 0; b < dwa_size + 1; b++)
                {
                        double v = double(b) * (v_max / double(dwa_size));
                        double w = double(a - (dwa_size - 1) / 2) * (2 * w_max / double(dwa_size));
                        pn_x = x + v * cos(q + w * dt / 2) * dt;
                        pn_y = y + v * sin(q + w * dt / 2) * dt;
                        pn_x_r = pn_x * cos(q) + pn_y * sin(q) - x * cos(q) - y * sin(q);
                        pn_y_r = -pn_x * sin(q) + pn_y * cos(q) + x * sin(q) - y * cos(q);
                        w_head = 1 - fabs(atan2(destination_y - pn_y, destination_x - pn_x) - (q + w * dt)) / PI;
⑥                      //function of the direction
                        w_speed = v / v_max;
                        //function of the speed
                        double d = 100, TEMP_d = 100, d_x = 0, d_y = 0;
                        //calculate d with grid map
                        for (int i = 0; i < map_size; i++)
                        {
                                for (int k = 0; k < map_size; k++)
                                {
                                        if (map[i][k] == 'x' || map[i][k] == '-')
                                        {
                                                d_x = double((map_size*3/4 - (i + 1))*2 + 1) * grid_size/2;
                                                d_y = double((map_size*1/2 - (k + 1))*2 + 1) * grid_size/2;
                                                TEMP_d = sqrt(pow(d_x - pn_x_r, 2) + pow(d_y - pn_y_r, 2));
                                                if (TEMP_d < d)
                                                        d = TEMP_d;
                                        }
                                }
                        }
                        if (d < turtlebot_length * 1 / 3) //preventing the selection of the v,w crossing over the obstacles
                                break;
                        if (d < danger_distance)
                                w_clear = 0;
                        else if (d < caution_distance)
                                w_clear = (d - danger_distance) / (caution_distance - danger_distance);
                        else
                                w_clear = 1;
                        //function of the distance to the obstacle
                        TEMP = g_head * w_head + g_speed * w_speed + g_clear * w_clear;
                        map[(map_size - 1) - int((pn_x_r + map_length / 4) / grid_size)][int((map_length / 2 - pn_y_r) / grid_size)] =
int(TEMP / (g_head + g_speed + g_clear) * 9 + 0.5) + 48;
                        //ASCII code of integer
                        if (TEMP > O_max)
                        {
                                O_max = TEMP;
                                v_dwa = v; w_dwa = w;
                                head_check = w_head;
                                speed_check = w_speed;
                                clear_check = w_clear;
                        }
```

```
                }
        }
        motorvalue.linear.x = v_dwa * go;   //my tutlebot is reversed...
        motorvalue.angular.z = w_dwa * go;   //counterclockwise is +
        //stopping code : preventing turtlebot's oscillation at the last point
⑦      if (fabs(destination_x - x) < 0.05 && fabs(destination_x - x) < 0.05)
        {
                go = 0;
                printf("stop");
        }
        for (int i = 0; i < map_size; i++)
        {
                for (int k = 0; k < map_size; k++)
                {
                        printf(" %c", map[i][k]);
                }
                printf("₩n");
        }
        printf("DWA value : %f, %f, %f₩n", head_check, speed_check, clear_check);
        printf("DWA : %fm/s, %frad/s₩n", v_dwa, w_dwa);
        printf("location : %f, %f theta : %fdeg₩n", x, y, q / PI * 180);
        printf("=============================================================================================================₩n");
        path_planning_motorpowerpub.publish(motorpower);
        path_planning_motorpub.publish(motorvalue);
        duration.sleep();
}


void odomCALLBACK(const nav_msgs::Odometry::ConstPtr& odom)
⑧ {
        float sqw = odom->pose.pose.orientation.w * odom->pose.pose.orientation.w;
        float sqx = odom->pose.pose.orientation.x * odom->pose.pose.orientation.x;
        float sqy = odom->pose.pose.orientation.y * odom->pose.pose.orientation.y;
        float sqz = odom->pose.pose.orientation.z * odom->pose.pose.orientation.z;
        float pitch = asinf(2.0f * (odom->pose.pose.orientation.y*odom->pose.pose.orientation.z + odom->pose.pose.orientation.w*odom-
>pose.pose.orientation.x)); // rotation about x-axis
        float roll = atan2f(2.0f * (odom->pose.pose.orientation.w*odom->pose.pose.orientation.y - odom->pose.pose.orientation.x*odom-
>pose.pose.orientation.z),(-sqx - sqy + sqz + sqw)); // rotation about y-axis
        //q = yaw
        q = atan2f(2.0f * (odom->pose.pose.orientation.w*odom->pose.pose.orientation.z - odom->pose.pose.orientation.x*odom-
>pose.pose.orientation.y),(-sqx + sqy - sqz + sqw)); // rotation about z-axis


        //converting from quaternian to theta
        x = odom->pose.pose.position.x;
        y = odom->pose.pose.position.y;
}


void stoping_func(const navigation::stopmsg::ConstPtr& stop)
{
        go = 0;
}


int main(int argc, char **argv)
{
        //initialize the node, and resister it to ROSCORE
        ros::init(argc, argv, "path_planning");
        //declare NodeHandle
        ros::NodeHandle nh;
        //declare pub,sub
        ros::Subscriber path_planning_odom = nh.subscribe("/odom", 1, odomCALLBACK);
        ros::Subscriber path_planning_break = nh.subscribe("/stop", 1, stoping_func);
        path_planning_motorpub = nh.advertise<geometry_msgs::Twist>("/cmd_vel", 1);
        path_planning_motorpowerpub = nh.advertise<std_msgs::Bool>("/motor_power", 1);
        ros::Subscriber path_planning_sub = nh.subscribe("/scan", 1, msgCALLBACK);
        ros::spin();
        return 0;
}
```

2. 코드 상세설명

위의 코드 전문에서, 설명이 용이하도록 구간을 나누어 놓았다. 다음은 코드의 각 구간에 대한 상세설명이다.

① *헤더 파일 및 전역 변수와 목표지점의 선정*

이 노드가 섭스크라이브하는 토픽의 메시지 헤더파일들과 기타 전역변수들을 선언해주었다. 특히 모터의 속도값과 전원을 퍼블리시하는퍼블리셔와 그 메시지는 메인 함수 밖, 즉 라이다의 /scan 토픽을 퍼블리시 받았을 때 호출되는 msaCALLBACK 함수 내에서 사용하기 위해 전역변수로 선언해주었다.

② *패딩을 포함한 격자지도 생성*

/scan 메시지에는 크기가 360인 1차 배열 안에 0° 부터 359° 까지 360개의 거리 값이 저장되어 있다. 이 값들을 이용하면 터틀봇의 위치를 원점으로 하는 X, Y 좌표를 만들 수 있다. char 자료형인 격자지도의 크기는 map_size변수로 설정해 주었고, 장애물이 있는 곳은 ' x '로, 아무것도 없는 곳은 빈 공간으로 설정하였다.

또한, 장애물이 있는 격자를 중심으로 하는 5 * 5 격자들에 모두 장애물이 존재한다고 인지하게 하여, 2cm의 패딩을 준 효과를 내도록 하였다.

③ *격자지도 상에서 가상의 장애물 생성*

앞서 설명하였듯이, 본 코드는 협로구간, 또는 오목한 장애물이 있는 구간을 매끄럽게 통과하기 위하여 가상의 장애물들을 설정해 주었다. 격자지도 상에서 두 장애물 격자 사이의 빈 칸이 가로, 또는 세로로 5칸이상, 15칸 이하일 경우 그 사이의 빈 칸들은 장애물은 존재하지 않지만, 터틀봇이 지나갈 수 없는 공간이거나 오목한 장애물로 판단하여 가상의 장애물 ' - '로 설정해주었다.

④ *DWA 가중치 및 기타 상수값 선정*

DWA의 각 목적함수의 가중치, Δt 등의 상수 값들을 이곳에서 부여해주었다. 이러한 상수값들은 여러가지의 맵들에서 수차례 실험해보며 가장 안정적인 가중치를 선정하였다.

⑤ *탐색속도의 예상지점 연산 및 상대좌표로의 변환*

본 코드에서는 터틀봇의 최대 가속도를 무한대로 가정하고, for문을 사용해 앞서 선정한 DWA_size의 값 만큼 터틀봇의 가능한 모든 각속도/선속도를 나누어 각각의 목적함수를 연산하였다. 또한, 각 속도에 따른 Δt 이 후의 예상지점을 터틀봇의 현재위치를 기준으로 하는 상대좌표로 나타내었다.

DWA의 목적함수를 연산하기 위해선 예상지점의 절대좌표와 상대좌표가 모두 필요했기 때문에, 터틀봇의 시작지점을 원점으로하는 절대좌표와 터틀봇의 현재좌표를 기준으로 하는 두 좌표사이의 Transfer Function 행렬을 계산하여 예상지점의 절대좌표를 상대좌표로 변환해 주었다.

⑥ *DWA의 목적함수 연산*

DWA의 세 가지 목적함수를 연산하는 코드이다. 먼저 방향함수인 w_head를 구할 때는 예상지점의 절대좌표와, 목표지점의 절대좌표를 연산하여 방향의 차이 θ를 구하였고, 그 크기가 0에 가까울수록 큰 값을 할당하였다.

속도함수인 w_speed를 구할 때는, 로봇의 속도가 0일 때를 0, 지정한 최대속도(0.1)일 때를 1로 할당하였고, 그 사이 값들에 대해선 선형 보간을 이용하여 값을 할당하였다.

충돌함수인 w_clear를 구할 때는 예상지점의 상대좌표와, 앞서 만든 격자지도 상의 실제/가상 장애물 까지의 거리를 구하였고, 그 거리가 특정 값 보다 크면 더 이상 장애물 까지의 거리를 신경 쓸 필요가 없으므로 1을, 특정 값 보다 작으면 0을, 그 사이의 값이면 선형 보간을 이용하여 값을 할당하였다. **또한, 예상지점과 장애물 사이의 거리가 작은 특정 값보다 작으면, break 함수를 사용하여 더 이상 그 각속도 방향의 선속도들을 탐색하지 않고 다음 각속도를 탐색하도록 하였다. 이를 통해 예상지점이 장애물을 넘어서 생기지 않도록 하였다.**

이 후에, 각각의 각속도/선속도 마다 방향함수, 속도함수, 충돌함수에 가중치를 곱하고 모두 더한 목적함수를 부여하고, 그 중 가장 큰 목적함수 값을 갖는 각속도/선속도를 선택하였다. 또한, 코드의 수정을 용이하게 하기 위하여, 탐색한 모든 예상지점마다 목적함수를 0점부터 10점 사이로 정규화시켜 앞서 생성한 격자지도 위에 출력하였다.

⑦ *목적지에 도달시 호출되는 정지함수*

터틀봇이 목표지점 부근에서 속도가 감소하며 진동하는 것을 막기 위하여, 터틀봇이 목표지점에 특정 거리 내로 접근했을 시, 도착했음을 인지하고 정지시키는 함수이다.

⑧ *터틀봇 localization*

터틀봇은 자신의 절대좌표를 매 순간 /odom 토픽으로 퍼블리시한다. 포지션은 x, y 절대좌표로 퍼블리시 해주지만, 오리엔테이션은 쿼터니언으로 퍼블리시하기에, 우리가 필요한 q(z축 방향 회전, yaw)를 얻기위해서는 변환이 필요하였다.

Ⅲ. 코드의 한계점 및 해결방안

1. 코드의 한계점

최종적인 코드를 여러 종류의 맵에서 실험해보고, 여러 번의 수정도 거쳐보았지만, 필자의 코드가 갖는 명확한 두 가지의 한계점을 찾을 수 있었다.

a. 국부최소문제(Local Minima)

본 코드는 가상의 장애물을 생성했음에도, 오목한 정도가 큰 장애물에 대해서 국부최소문제를 자주 일으켰다. 이는 global path plannig이 없이 local path planning만 존재하는 본 코드가 근본적으로 가지고있는 문제라고 생각된다.

b. 속도 플래닝의 부재

앞서 언급했듯이, 기본적으로 DWA는 로봇의 현재속도에서, 로봇의 최대 선가속도와 최대 각가속도의 범위 내에 있는 선속도와 각속도를 탐색한다. 그러나 본 코드에서는 터틀봇의 최대 가속도가 무한대라고 가정하고 모든 선속도와 각속도를 탐색하였다. 따라서 속도를 선택하는데에 필요한 연산량이 더 많아질뿐더러, 터틀봇이 자신의 최대 가속도 이상의 속도변화를 선택했을 때 로봇이 부드럽게 움직이지 않는 현상이 발생하였다.

2. 해결방안

본 프로젝트에서는 미리 맵이 주어지지 않았기 때문에, global path planning 을 활용할 수 없었다. 그럼에도, 국부최소문제에 빠지지않고 장애물을 안정적으로 피할 수 있는 본 코드의 수정 방향 또는 새로운 코드를 다음과 같이 제안한다.

본 코드의 수정 방향 :

- 국부최소문제에 빠졌음을 인지하고 그 전의 위치로 되돌아오고, 국부최소문제를 일으켰던 위치를 장애물로 인식하는 코드
- 격자지도를 탐색하여 국부최소문제를 유발하는 공간을 인지하고, 그 공간을 장애물로 인식하는 코드

새로운 코드 :

- 한쪽 벽만 안정적으로 따라가는 코드
- 지역격자지도 내에서 특정 격자를 중간목표지점으로 선정하고, 그 목표지점까지의 global path planning을 이용해 이동하는 것을 반복하는 코드

위의 코드들을 잘 활용한다면 주행이 안정적이고, 어떠한 국부최소문제에도 빠지지않는 네비게이션 코드를 작성할 수 있을 것이라 생각한다.

Ⅳ. 참고문헌

[1] 윤회상, 박태형, "수정된 전역 DWA에 의한 자율이동로봇의 경로계획", 전기학회논문지, 60권, 2호, 389-397, 2011.

[2] 표윤석, 조한철, 정려운, 임태훈, "ROS 로봇 프로그래밍", 2017.