In [ ]:
```
def pathsearch(s_pt,g_pt):
    #######real map obstacle
    r_list = []
    for i in range (6):
        for j in range (4):
            r_pt = [9+j,i]
            r_list.append(r_pt)
    for i in range (5):
        for j in range (3):
            r_pt = [13+j,9+i]
            r_list.append(r_pt)
    for i in range (10):
        for j in range (5):
            r_pt = [3+j,6+i]
            r_list.append(r_pt)

    #way_pt = [[17,15],[6,14],[3,4],[17,3],[9,9]]
    route = []

    #s_pt = [1,2]
    s_gt_pt = mapper.from_map(s_pt[0],s_pt[1],0)
    #loc.plotter.plot_point(s_gt_pt[0], s_gt_pt[1],ODOM)
    #g_pt = [17,17]
    g_gt_pt = mapper.from_map(g_pt[0],g_pt[1],0)
    #loc.plotter.plot_point(g_gt_pt[0], g_gt_pt[1],GT)
    r_flag = 0
    if((20- s_pt[0]) > (20 - g_pt[0]) or s_pt[1] > g_pt[1]):
        buff = g_pt
        g_pt = s_pt
        s_pt = buff
        r_flag = 1



    #way_pt = [[17,8]] #sim_way_pt
    way_pt = [[10,10]]
    way_pt.append(g_pt)
    route.append(s_pt)
    s_gt_pt = mapper.from_map(s_pt[0],s_pt[1],0)
    loc.plotter.plot_point(s_gt_pt[0], s_gt_pt[1],ODOM)
    for i in range (len(way_pt)):
        g_pt = way_pt[i]
        dis_x_f = g_pt[0]-s_pt[0]
        dis_y_f = g_pt[1]-s_pt[1]

        con = 0
        flag_x = 0
        flag_y = 0
        tmp_x = s_pt[0]
        tmp_y = s_pt[1]
        y_en = 0
        y_en2 = 0

        while(con == 0):
            while(flag_x < abs(dis_x_f)):
                dis_x = g_pt[0] - tmp_x
```

```
            dis_y = g_pt[1] - tmp_y
            if(y_en == 0):
                if(dis_x <0):
                    tmp_x = tmp_x - 1
                    tmp_pt = [tmp_x, tmp_y]
                else:
                    tmp_x = tmp_x + 1
                    tmp_pt = [tmp_x, tmp_y]
            else:
                tmp_pt = [tmp_x, tmp_y]
            flag = 0
            for i in range (len(r_list)):
                if(np.any(tmp_pt == r_list[i])):
                    flag = flag + 1
                elif(tmp_pt[0] >= 21 or tmp_pt[0] <= -1):
                    flag = flag + 1
                elif(tmp_pt[1] >= 21 or tmp_pt[1] <= -1):
                    flag = flag + 1
                else:
                    flag = flag + 0
            if(flag > 0 and y_en == 0): #go up
                if(dis_x <0):
                    tmp_x = tmp_x + 1
                else:
                    tmp_x = tmp_x - 1
                tmp_y = tmp_y + 1
                y_en = 1
            elif(flag > 0 and y_en == 1): #go down
                tmp_y = tmp_y - 2
                y_en2 = 1
            elif(flag ==0):
                route.append(tmp_pt)
                tmp_gt_pt = mapper.from_map(tmp_pt[0],tmp_pt[1],0)
                loc.plotter.plot_point(tmp_gt_pt[0], tmp_gt_pt[1],ODOM)
                if(y_en == 1 and y_en2 == 0):
                    if(dis_y < 0):
                        flag_y = flag_y - 1
                    else:
                        flag_y = flag_y + 1
                elif(y_en == 1 and y_en2 == 1):
                    if(dis_y < 0):
                        flag_y = flag_y + 1
                    else:
                        flag_y = flag_y - 1
                else:
                    flag_x = flag_x + 1
                y_en = 0
                y_en2 = 0
        x_en = 0
        x_en2 = 0

        while(flag_y < abs(dis_y_f)):
            dis_x = g_pt[0] - tmp_x
            dis_y = g_pt[1] - tmp_y
            if(x_en ==0):
                if(dis_y <0):
                    tmp_y = tmp_y - 1
```

```
                        tmp_pt = [tmp_x, tmp_y]
                    else:
                        tmp_y = tmp_y + 1
                        tmp_pt = [tmp_x, tmp_y]
                else:
                    tmp_pt = [tmp_x, tmp_y]
                flag = 0
                for i in range (len(r_list)):
                    if(np.any(tmp_pt == r_list[i])):
                        flag = flag + 1
                    elif(tmp_pt[0] >= 21 or tmp_pt[0] <= -1):
                        flag = flag + 1
                    elif(tmp_pt[1] >= 21 or tmp_pt[1] <= -1):
                        flag = flag + 1
                    else:
                        flag = flag + 0
                if(flag > 0 and x_en == 0): #go right
                    if(dis_y <0):
                        tmp_y = tmp_y + 1
                    else:
                        tmp_y = tmp_y - 1
                    tmp_x = tmp_x + 1
                    print(tmp_x, tmp_y)
                    x_en = 1
                elif(flag > 0 and x_en == 1): #go left
                    tmp_x = tmp_x - 2
                    print(tmp_x, tmp_y)
                    x_en2 = 1
                elif(flag == 0):
                    route.append(tmp_pt)
                    tmp_gt_pt = mapper.from_map(tmp_pt[0],tmp_pt[1],0)
                    loc.plotter.plot_point(tmp_gt_pt[0], tmp_gt_pt[1],ODOM)
                    if(x_en == 1 and x_en2 == 0):
                        if(dis_x < 0):
                            flag_x = flag_x - 1
                        else:
                            flag_x = flag_x + 1
                    elif(x_en == 1 and x_en2 == 1):
                        if(dis_y < 0):
                            flag_x = flag_x + 1
                        else:
                            flag_x = flag_x - 1
                    else:
                        flag_y = flag_y + 1
                    x_en = 0
                    x_en2 = 0
            if(tmp_x == g_pt[0] and tmp_y == g_pt[1]):
                con = 1
        s_pt = g_pt

    if(r_flag == 1):
        route = route[::-1]
    return route
```

In [ ]:
```
def robotcontrol(route):
    #calculate the approximate time for velocity
    dist = []
    for i in range (len(route)-1):
        if(route[i+1][1] == route[i][1]): #x change
            if(route[i+1][0] == route[i][0] + 1):
                dist.append(1)
            else:
                dist.append(-1)
        else: #y change
            if(route[i+1][1] == route[i][1] + 1):
                dist.append(2)
            else:
                dist.append(-2)
    v_control = []
    t_control = []
    v_control.append(dist[0])
    t = 0
    for i in range (len(dist)):
        if(i > 0 and dist[i] != dist[i-1]):
            v_control.append(dist[i])
            t_control.append(t)
            t = 0
        t = t + 1
    t_control.append(t)

    bond_x = 4
    bond_y = 4
    spa_x = bond_x / 20
    spa_y = bond_y / 20
    for i in range (len(v_control)):
        if(v_control[i] == 1 or v_control[i] == -1):
            t_control[i] = t_control[i]*spa_x
        elif(v_control[i] == 2 or v_control[i] == -2):
            t_control[i] = t_control[i]*spa_y
    return v_control, t_control
```

In [ ]:
```
def plan_route(v_control, t_control,v):
    send_list = []
    time_list = []
    i_angle = 9
    for i in range (len(v_control)):
        ang_flag = 0
        if(v_control[i] == 1):
            angle = 9 #index 9 -- 0 degree
        elif(v_control[i] == -1):
            angle = 0  #index 0 ---180 degree
        elif(v_control[i] == 2):
            angle = 13 #index 13 --90 degree
        elif(v_control[i] == -2):
            angle = 4 #index 4 -- -90 degree
        diff_ang = i_angle - angle
        if(abs(diff_ang) >= 2):
            if(diff_ang < 0):
                diff_ang = abs(diff_ang)
                #w = abs(w)
            else:
                diff_ang = 18 - diff_ang
                #w = -abs(w)

            t_turn = diff_ang // 4

            send_list.append("t")
            time_list.append(t_turn)
            i_angle = angle

        send_list.append("f")
        #i_angle = pose[2]
        v_t = t_control[i]/v*10
        time_list.append(round(v_t))
        time.sleep(0.1)
    return send_list, time_list
```

In [ ]:
```
class RealRobot(BaseRobot):
    """A class to interact with the real robot
    """

    def __init__(self):
        super().__init__()
        print("Initializing Real Robot")

    def get_pose(self,data_set):
        """Get the latest odometry pose data in the map frame.

        Do NOT change the arguments or return values of this function.

        Returns:
            (x, y, a) (float, float, float): A tuple with latest odometry pose
in the map frame
                                                in the format (x, y, a) with unit
s (meters, meters, degrees)

        """
        pose =[data_set[18],data_set[19],data_set[20]]
        return pose

    def perform_observation_loop(self, observation_count, data_set):
        """ Implement a Bluetooth command, that tells your robot to
        start an anti-clockwise, rotational scan using PID control on
        the gyroscope. The scan needs to be a full 360 degree rotation with
        at least 18 readings from the TOF sensor, with the first reading taken
        at the current heading of the robot. At the end of the scan,
        have your robot send back the TOF measurements via Bluetooth.

        If you haven't already, write an automated script to pair down your
        measurements to 18 approximately equally spaced readings such that
        the first reading was taken at the heading angle of the robot.
        Use a reasonable rotational speed to achieve this behavior.

        Do NOT change the arguments or return values of the function since it
 will
        break the localization code. This function is called by the member fun
ction
        "get_obseration_data" in the Localization class (robot_interface.py),
        with observation_count = 18 and rot_vel = 30.
        You may choose to ignore the values in the arguments.

        Args:
            observation_count (integer): Number of observations to record
            rot_vel (integer): Rotation speed (in degrees/s)

        Returns:
            obs_range_data (ndarray): 1D array of 'float' type containing obse
rvation range data
        """
        mes_list = []
```

```
        for i in range(18):
            mes_list.append(data_set[i])
        return mes_list

    def set_vel(self, v, w):
        """Set a linear and an angular velocity for your robot.

        You will use this function to move the robot.
        It is not used by the Localization class and so you
        may change the arguments and/or return types as needed.

        Args:
            v (integer): Linear velocity
            w (integer): Angular velocity
        """
        v = 0.3
        w = 0.1

    def get_gt_pose(self, data_set):
        # Do not change this function
        """Get the latest ground truth pose data

        Do NOT change the arguments or return values of this function.

        Returns:
            (x, y, a) (float, float, float): A tuple with latest ground truth
 pose
                                                in the format (x, y, a) with unit
s (meters, meters, degress)

        """
        # Since there is no mechanism to find out the ground truth pose of you
r real robot,
        # it simply returns the odometry pose.
        # This function exists to comply with the design model of the Localiza
tion class
        pose =[data_set[18],data_set[19],data_set[20]]
        return pose
```

In [ ]:
```
#  Reset the plot, initializes the belief with a uniform distribution,
# performs the rotation behaviour, and runs the update step
def init_bayes_filter(data_set):
    # Reset Plots
    loc.plotter.reset_plot()

    # Initiize belief
    loc.init_uniform_distribution()

    # Get Observation Data by executing a 360 degree rotation motion
    loc.get_observation_data(18, data_set)

    # Update Step
    loc.update_step()
    pose = loc.print_update_stats(data_set, plot_data=True)
    return pose

# One iteration of the Bayse filter algorithm
def step_bayes_filter(current_odom, prev_odom, data_set):
    # Prediction Step
    loc.prediction_step(current_odom, prev_odom)
    loc.print_prediction_stats(data_set, plot_data=True)

    # Get Observation Data by executing a 360 degree rotation behavior
    loc.get_observation_data(18, data_set)
    # Update Step
    loc.update_step()
    pose = loc.print_update_stats(data_set,plot_data=True)

    return pose

# Records the odom before a robot motion,
# moves the robot, and records the odom again after motion
def move_robot():
    prev_odom = robot.get_pose(data_set)

    # Code to move your robot goes here

    current_odom = robot.get_pose(data_set)

    return current_odom, prev_odom
```

In [ ]:
```
#discover BLE
loop = asyncio.get_event_loop()
asyncio.gather(robotTest(loop))
```

In [ ]:
```
await theRobot.sendMessage("s") #start
await theRobot.sendMessage("c") #start scan

time.sleep(20)

data_set = await theRobot.sendCommand(Commands.REQ_FLOAT) #get the scan data
#get the first position
pose = init_bayes_filter(data_set)

#assign start point and goal point
s_pt = [pose[0],pose[1]]
g_pt = [4,4] #define a goal location
#g_pt = [17,10] #define a goal location
#path planning
route = pathsearch(s_pt,g_pt)

[v_control,t_control]=robotcontrol(route)

#speed control
v = 0.45
[send_list,time_list]=plan_route(v_control, t_control,v)

#plot inital point
a_pt = route[0]
a_gt_pt = mapper.from_map(a_pt[0],a_pt[1],0)
loc.plotter.plot_point(a_gt_pt[0], a_gt_pt[1],GT)

#start process
prev_odom = pose
for i in range (len(send_list)):
    await theRobot.sendMessage(send_list[i])
    print(send_list[i])
    time.sleep(0.5)
    if(time_list[i]>=15):
        t_str = str(15)
    else:
        t_str = str(time_list[i])
    await theRobot.sendMessage(t_str)
    con = 0
    time.sleep(15)
    '''while(con == 0):
        data_set = await theRobot.sendCommand(Commands.REQ_FLOAT)
        time.sleep(0.5)

        if(data_set[0] == 100.0){
            con = 1
            time.sleep(0.2)
        }'''

time.sleep(1)
#identify whether the robot reach the last point
await theRobot.sendMessage("c") #start scan
time.sleep(20)
data_set = await theRobot.sendCommand(Commands.REQ_FLOAT) #get the scan data
current_odom = robot.get_pose(data_set)
pose = step_bayes_filter(current_odom, prev_odom, data_set)
```

```
close_enough = 0.5

end_pt = route[-1:][0]
real_loc = mapper.from_map(end_pt[0],end_pt[0],pose[2])
exp_loc = mapper.from_map(pose[0],pose[1],pose[2])
dis_err = (real_loc[0] - exp_loc[0]) ** 2 + (real_loc[1] - exp_loc[1]) ** 2

if(dis_err <= (close_enough **2)):
    print("reach the point")
```