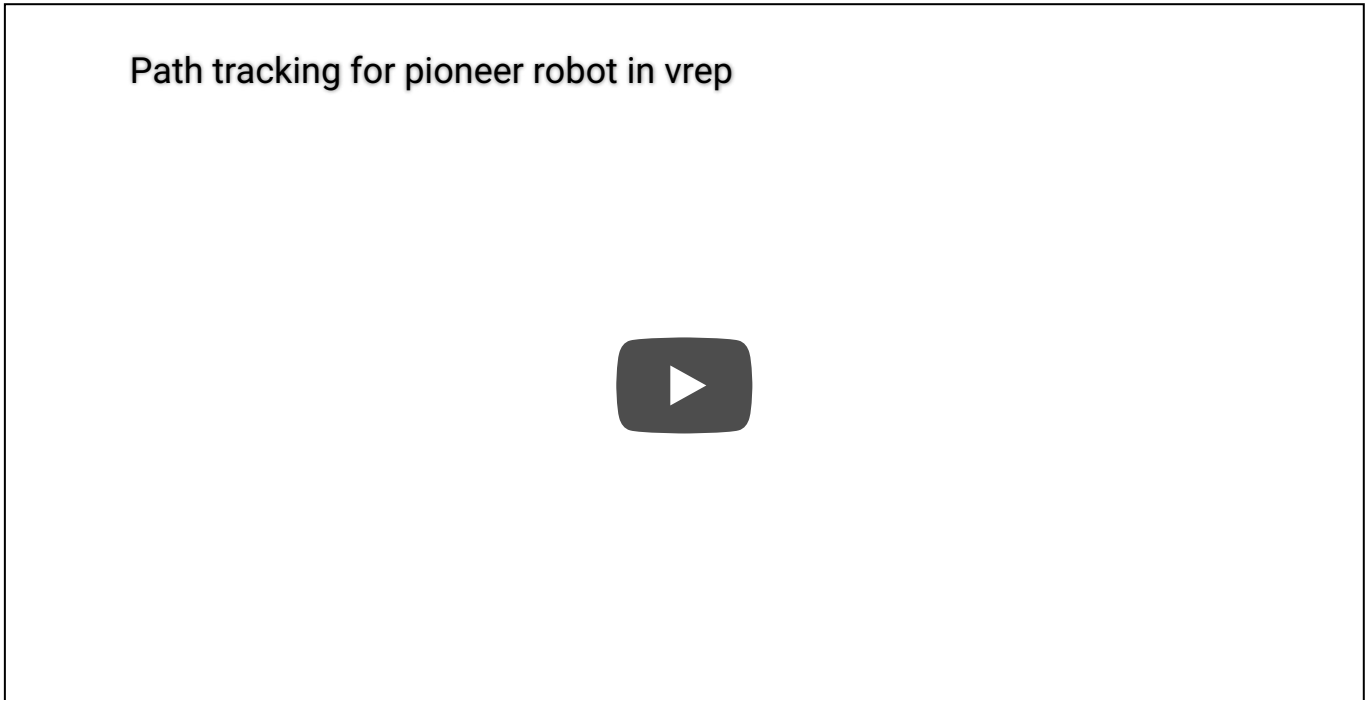
Path tracking tutorial for pioneer robot in vrep

Ahmed Essam El Fakharany Apr 16, 2018 · 12 min read



The video is 3x the speed

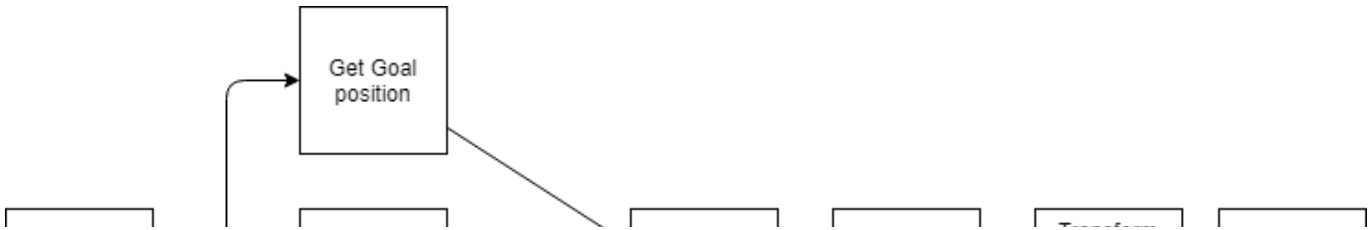
This tutorial is aimed at explaining how to do path tracking using an algorithm called follow the carrot. The aim is to make the robot reach for the goal position from its current position following a path made for it, the path is taking into account the position of theobstacles. The simulation environment used was Vrep and the all the algorithms were implemented in python.

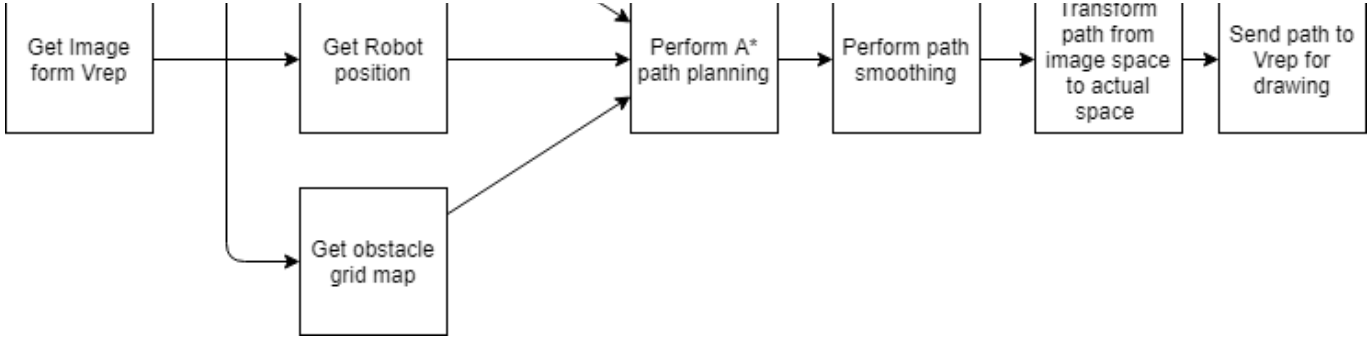
Everything could be found in the GitHub repo.

System Diagram

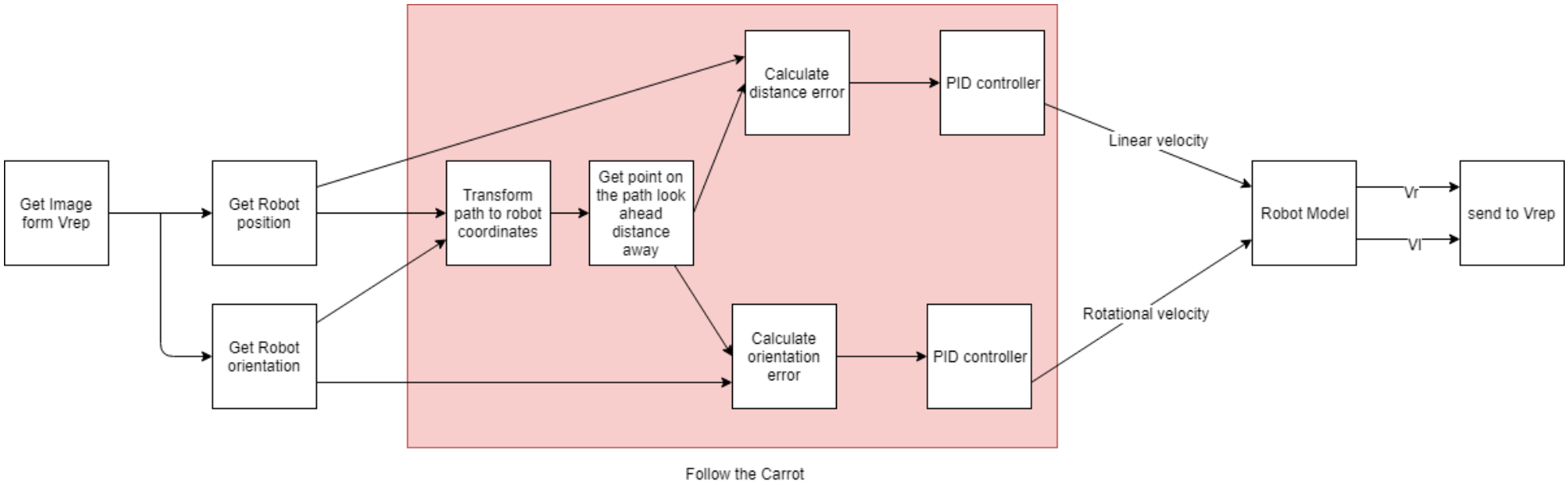
First of All we will discuss the system diagram.

The system has 2 stages, one that is path planning and smoothing and it is done once in the beginning of the program.





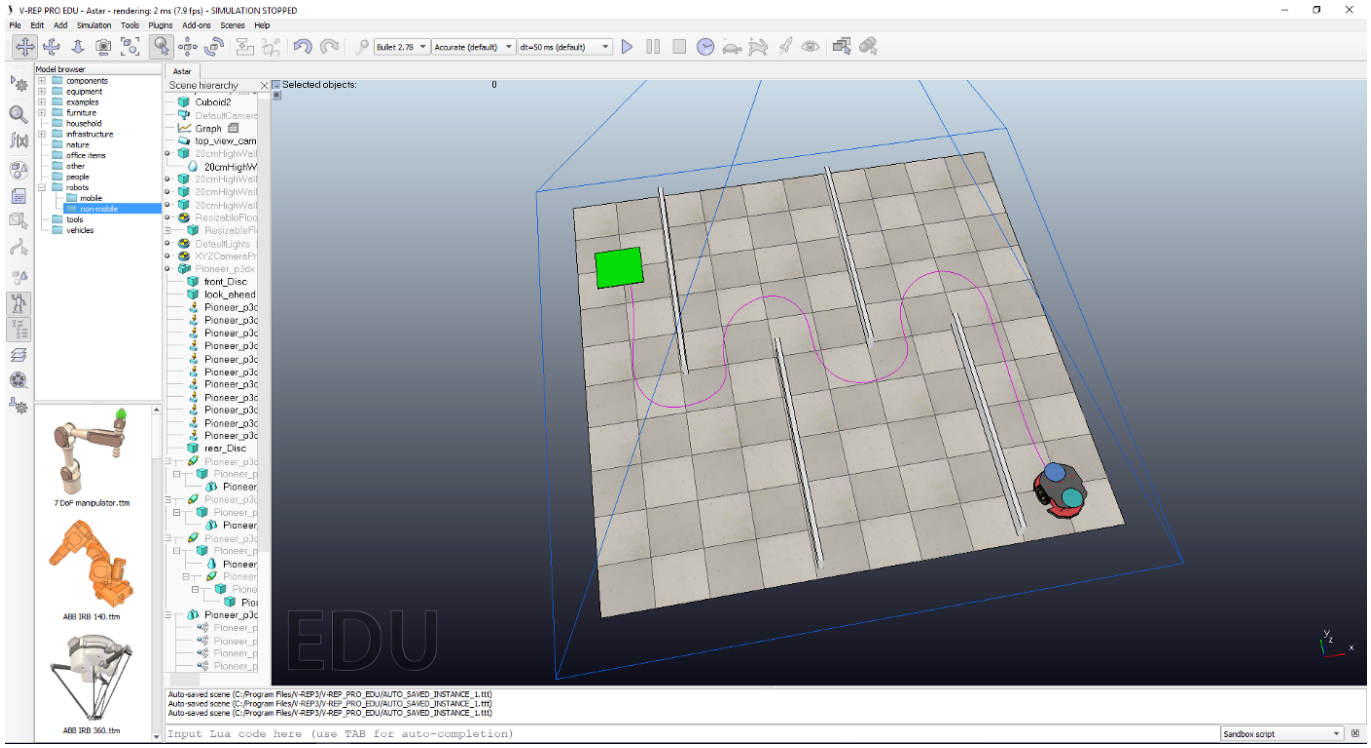
The other stage is what happens every iteration of the program that actually makes the robot follow the path layed for it



Path tracking

The path tracking algorithm used is called follow-the-carrot in which a point on look ahead distance away from the robot on the path is used as an aiming point for the robot, then control the robot's linear velocity and rotational velocity to reach that point, when the vehicle moves a new carrot point is chosen for the robot to aim at, more details are discussed here.

Vrep Scene



This tutorial assumes familiarity with Vrep and how to use the python remote API, if not please check the Vrep official tutorials and some really good videos by Nikolai K.

The scene can be found in the file named ' Astar.ttt' found in the GitHub repo, it consists of multiple elements, first is the camera named 'top_view_camera', it is put on top of the scene and is used to get the position of obstacles , the goal , the robot's position and the robot's orientation. Then we have the obstacles that are simply 20cm high whitewalls. Then the Pioneer robot which has 2 disks attached to it with 2 different colors to let us get its orientation and position, there is also a sphere attached to it used to show the point on the path the robot is aiming -more on the later ;) -, a graph object is attached to the robot to draw the path it followed.

There is a thread script -in lua- attached to the lights in the scene -I know it's not the best thing to do- to start the remote API server and to draw the received points from the python script showing the path the robot should follow which are the red dots in the above video, that function is :

```
function sysCall_threadmain()
    -- Put some initialization code here
    simRemoteApi.start(19999)
```

```
    lineSize=2 -- in points
 maximumLines=9999
 red={1,0,0}

drawingContainer=sim.addDrawingObject(sim.drawing_points,lineSize,0,
-1,maximumLines,red) -- adds a line
-- Put your main loop here, e.g.:
    --
    while
sim.getSimulationState()~=sim.simulation_advancing_abouttostop do
        myData=sim.getStringSignal("path_coord")
        sim.clearStringSignal('path_coord')
        if (myData) then
            data=sim.unpackFloatTable(myData)
            data[#data+1]=0.01
            sim.addDrawingObjectItem(drawingContainer,data)
        end

    end

end
```

Python notebook

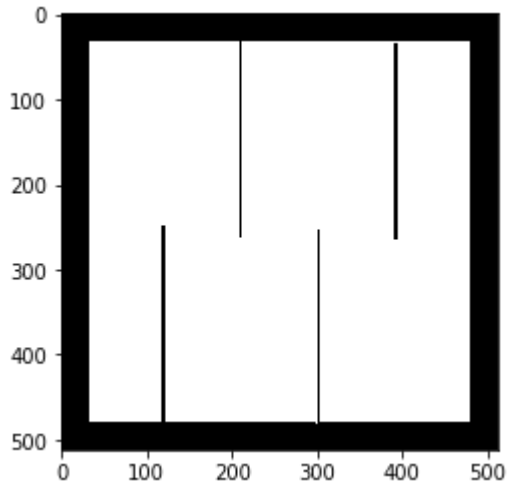
The python notebook is named ‘path_tracking.ipynb’ in the GitHub repo and in it all the action happens.

Obstacle Grid

The ‘top_view_camera’ is used to obtain the obstacle grid by the use of color thresholding, the function below takes in the image of the scene and

returns the obstacles grid where an obstacle has a value of ‘0’ and the non obstacle area has the value of ‘1’:

```
def obstacles_grid(img):
    # getting the walls
    mask_wall = cv2.inRange(img, np.array([230,230,230]),
                                np.array([240,240,240]))
    # getting the rims
    mask_rim = cv2.inRange(img, 0, 0)
    mask_total = cv2.bitwise_or(mask_wall,mask_rim,mask_rim)
    mask_total = cv2.bitwise_not(mask_total)
    return mask_total
```



Robot position and orientation

Also the robot orientation and position are obtained from the image from the ‘top_view_camera’, color thersholding is used to identify the 2 disks on top of the robot and then the center of contour is computed for each disk of them and then the position of the robot center is at the middle between them and the orientation is calculated by getting the angle between the vector connecting the 2 centers together and the vertical axis.

```
def get_front_disk_position (img):
    front_disk_mask = cv2.inRange(img,
                                  np.array([100,140,205]),
                                  np.array([105,145,210]))
    im2, contours, hierarchy =
           cv2.findContours(front_disk_mask.copy(),
                cv2.RETR_EXTERNAL,
                cv2.CHAIN_APPROX_SIMPLE)
    #no need for the for loop but it is used to avoid errors
    for c in contours:
        # compute the center of the contour
        M = cv2.moments(c)
        cX = int(M["m10"] / M["m00"])
        cY = int(M["m01"] / M["m00"])
    return np.array([cY,cX])


def get_back_disk_position (img):
    rear_disk_mask = cv2.inRange(img,
                         np.array([75,185,185]),
                         np.array([80,190,190]))
    im2, contours, hierarchy =
                    cv2.findContours(rear_disk_mask.copy(),
                                     cv2.RETR_EXTERNAL,
                                     cv2.CHAIN_APPROX_SIMPLE)
    #no need for the for loop but it is used to avoid errors
    for c in contours:
```

```
        # compute the center of the contour
        M = cv2.moments(c)
        cX = int(M["m10"] / M["m00"])
        cY = int(M["m01"] / M["m00"])
    return np.array([cY,cX])
def get_robot_orientation(img):
    front_disk_position = get_front_disk_position (img)
    rear_disk_position =  get_back_disk_position (img)
    orientation = np.arctan2(
         rear_disk_position[1]-front_disk_position[1],
         rear_disk_position[0]-front_disk_position[0])
    return orientation
def get_robot_position(img):
    front_disk_position = get_front_disk_position (img)
    rear_disk_position =  get_back_disk_position (img)
    robot_pos = (front_disk_position+rear_disk_position)/2
    return tuple(robot_pos.astype('int'))
```

Goal Position

Now we need to get the goal's position, we do the same thing we did with the robot, color thresholding and computing the center of a contour.

```
def get_goal_position(img):
    goal_mask = cv2.inRange(img[:,:,1], 240, 255)
    goal_mask = cv2.GaussianBlur(goal_mask, (9, 9), 0)
    im2, contours, hierarchy = cv2.findContours(goal_mask.copy(),
                                        cv2.RETR_EXTERNAL,
                                        cv2.CHAIN_APPROX_SIMPLE)
    #no need for the for loop but it is used to avoid errors
    for c in contours:
        # compute the center of the contour
        M = cv2.moments(c)
```

```
            cX = int(M["m10"] / M["m00"])
            cY = int(M["m01"] / M["m00"])


    return goal_mask, (cY,cX)
```

Path Planning

Now we have the obstacles grid, the robot's position and orientation and the goal's position; all we need now is to plan a path from the robot's position to the goal's position taking into account the obstacles, for that we are using A* algorithm, I could try to explain the algorithm but I believe I won't top the best explanations out there, so I think it's better to direct you to some of the best tutorials in case you don't know what A* is, one of the best tutorials out thers is Amit Patel's website about A* and another one is Sebastian Thrun's videos on motion planning.

In the code bellow, different heuristic functions are defined but only one is used, that is to compare the results between them. Also the G function isn't varies depending on the proximity of the robot to the obstacle, the closer the robot is the higher is the cost. Also high change in the robot's motion has a high cost.

The function for the path planning is below but I think checking it on the GitHub repo would be better.

```
def search(grid,init,goal,cost, D = 1, fnc='Euclidean', D2 = 1):

    def Euclidean_fnc(current_indx, goal_indx,D = 1):
        return np.sqrt( ( (current_indx[0]-goal_indx[0])**2 +
(current_indx[1]-goal_indx[1])**2 ) )
    def Manhattan_fnc(current_indx, goal_indx,D = 1):
        dx = np.sqrt((current_indx[0]-goal_indx[0])**2)
        dy = np.sqrt((current_indx[1]-goal_indx[1])**2)
        return D * (dx + dy)
    def Diagonal_fnc(current_indx, goal_indx,D = 1):
        dx = np.sqrt((current_indx[0]-goal_indx[0])**2)
        dy = np.sqrt((current_indx[1]-goal_indx[1])**2)
        return D * (dx + dy) + (D2 - 2 * D) * min(dx, dy)

    if fnc=='Euclidean':
        hueristic_fnc = Euclidean_fnc
    elif fnc == "Manhattan":
        hueristic_fnc = Manhattan_fnc
    elif fnc == "Diagonal" :
        hueristic_fnc = Diagonal_fnc


    def near_obstacles(point, half_kernel = 5):
        x_start = int(max(point[0] - half_kernel, 0))
        x_end = int(min(point[0] + half_kernel, grid.shape[0]))
        y_start = int(max(point[1] - half_kernel, 0))
        y_end = int(min(point[1] + half_kernel, grid.shape[1]))
        return np.any(grid[x_start:x_end, y_start:y_end]<128)

    def delta_gain(gain = 1):
        delta = np.array([[-1, 0], # go up
                          [-1,-1], # up left
                          [ 0,-1], # go left
                          [ 1,-1], # down left
                          [ 1, 0], # go down
                          [ 1, 1],  # down right
                          [ 0, 1], # go right
                          [-1, 1] # up right
         ])
        return delta*gain
```

```
    delta = delta_gain(gain = 5)
    front = PriorityQueue()
    G = 0
    H = hueristic_fnc(init, goal, D)
    F = G+H
    front.put((F, G, init))
    discovered = []
    discovered.append(init)

    actions = np.ones_like(grid)*-1
    count = 0
    path = []

    def policy_draw(indx):
        indx_old = tuple(indx)
        indx_new = tuple(indx)
        path.append(tuple(goal))
        while indx_new != init:
            indx_new = tuple( np.array(indx_old) -
delta[int(actions[indx_old])] )
            path.append(indx_new)
            indx_old = indx_new


    while not front.empty() :
        front_element = front.get()
        G = front_element[1]
        indx = front_element[2]
        if ((indx[0] >= goal[0]-20) and (indx[0] < goal[0]+20)) and
((indx[1] >= goal[1]-20) and (indx[1] < goal[1]+20)):
            policy_draw(indx)
            print("found goal")
            print(count)
            print(front_element)
            break
        else:
            for y in range(len(delta)) :
                indx_new = tuple(indx + delta[y])
                if ((np.any(np.array(indx_new) < 0)) or (indx_new[0]
> grid.shape[0]-1) or (indx_new[1] > grid.shape[1]-1)) :
```

```
                    continue
                if (grid[indx_new] >= 128) and (indx_new not in
discovered) :
                    count += 1
                    # if the obstacle is inside the robot :D, have a
really high cost
                    if near_obstacles(indx_new, half_kernel = 35):
                        g_new = G + 1500*cost
                    # if the obstacle is about a robot's length near
it , have a high cost
                    elif near_obstacles(indx_new, half_kernel = 70):
                        g_new = G + 15*cost
                    # as before
                    elif near_obstacles(indx_new, half_kernel =
100):
                        g_new = G + 10*cost
                    # as before
                    elif near_obstacles(indx_new, half_kernel =
110):
                        g_new = G + 5*cost
                    else:
                        g_new = G + cost
                    #trying to increase the cost of rapidly changing
direction
                    if y == actions[indx]:
                        g_new = g_new
                    elif (y-1)%len(delta) == actions[indx] or
(y+1)%len(delta) == actions[indx]:
                        g_new = g_new + 5*cost
                    else :
                        g_new = g_new + 10*cost
                    h_new = hueristic_fnc(indx_new, goal, D)
                    f_new = (g_new + h_new)-0.0001*count
                    front.put((f_new, g_new, indx_new))
                    discovered.append(indx_new)
                    actions[indx_new] = y
    else:
        print(count)
        print("fail")
    return actions, np.array(path[::-1])
```

Path smoothing

The path planned by the A* might contain sharp turns that are hard for a non holonomic robot to follow, that's why we use path smoothing to make it have less of these sharp turns. Basically what's happening is that we are trying to minimize the distance between the point and the next point, and also minimize the distance that the point moves away from its original position, we do that using gradient descent, it's explained in more details by Sebastian Thrun.

The function for the smoothing algorithm is below but again I think that checking it on the GitHub repo would be better.

```
def smooth(path,grid, weight_data = 0.5, weight_smooth = 0.1,
tolerance = 0.000001, number_of_iter = 1e3):
    newpath = np.copy(path).astype('float64')
    def get_near_obstacles(point, area = 5):
        x_start = int(max(point[0] - area, 0))
        x_end = int(point[0] + area)
        y_start = int(max(point[1] - area, 0))
        y_end = int(point[1] + area)
        points = np.argwhere(grid[x_start:x_end, y_start:y_end]<128)
        points [:,0] += x_start
        points [:,1] += y_start
        if not points.size:
            points = point.copy()
        return points

    def near_obstacles(point, half_kernel = 2):
        x_start = int(max(point[0] - half_kernel, 0))
```

```
            x_end = int(point[0] + half_kernel)
            y_start = int(max(point[1] - half_kernel, 0))
            y_end = int(point[1] + half_kernel)
            return np.any(grid[x_start:x_end, y_start:y_end]<128)

        error = np.ones(path.shape[0])*tolerance+tolerance
        num_points = path.shape[0]
        for count in range(int(number_of_iter)):
            for i in range(1,num_points-1):
                old_val = np.copy(newpath[i])
                update1 = weight_data*(path[i] - newpath[i])
                update2 = weight_smooth*(newpath[i-
1]+newpath[i+1]-2*newpath[i])
                newpath[i]+=update1+update2
                if near_obstacles(newpath[i], half_kernel = 35):
                    newpath[i] = old_val
                error[i] = np.abs(np.mean(old_val-newpath[i]))
            if np.mean(error) < tolerance:
                break
        print(count)
        return newpath
```

Now Here is a result of what we have done uptil now, we marked the centerr of the goal and the center of the robot, we drew the planned path in red and the smoothed path in magenta.

Some helping functions

So next we will define some functions that will be used to move the robot.

First we will need to transform the path points from the image coordinates to vrep coordinates, this requires some translation, rotation and scaling

which is done in this function:

```
#transform from image frame to vrep frame
def transform_points_from_image2real (points):
    if points.ndim < 2:
        flipped = np.flipud(points)
    else:
        flipped = np.fliplr(points)
    scale = 5/445
    points2send = (flipped*-scale) +
                   np.array([2.0555+0.75280899, -2.0500+4.96629213])
    return points2send
```

Then we will need to transform the points from the vrep coordinates to the robot frame of reference

```
#transform from vrep frame to robot frame
def transform2robot_frame(pos, point, theta):
    pos = np.asarray(pos)
    point = np.asarray(point)
    T_matrix = np.array([
            [np.sin(theta), np.cos(theta)],
            [np.cos(theta), -1*np.sin(theta)],
            ])
    trans = point-pos
    if trans.ndim >= 2:
        trans = trans.T
        point_t = np.dot(T_matrix, trans).T
    else:
        point_t = np.dot(T_matrix, trans)
    return point_t
```

Then we will need to check if the robot is near a certain point by a certain distance, mainly the goal center to stop the robot

```
def is_near(robot_center, point, dist_thresh = 0.25):
    dist = np.sqrt((robot_center[0]-point[0])**2 +
                   (robot_center[1]-point[1])**2)
    return dist<=dist_thresh
```

Here is a function to get the euclidean distance between 2 points

```
def get_distance(points1, points2):
    return np.sqrt(np.sum(np.square(points1 - points2), axis=1))
```

Here is a simple class to perform pid control, we will use it to control the linear and rotaional velocities of the robot

```
class pid():
    def __init__(self, kp, ki, kd):
        self.kp = kp
        self.ki = ki
        self.kd = kd
```

```
        self.error = 0.0
        self.error_old = 0.0
        self.error_sum = 0.0
        self.d_error = self.error - self.error_old
    def control(self,error):
        self.error = error
        self.error_sum += error
        self.d_error = self.error - self.error_old
        P = self.kp*self.error
        I = self.ki*self.error_sum
        D = self.kd*self.d_error
        self.error_old = self.error
        return P+I+D
```

At last do you remember the thread script in lua we used to receive the points from the python API, it was way up in this post :D well here is the function responsible for sending the points

```
def send_path_4_drawing(path, sleep_time = 0.07):
    #the bigger the sleep time the more accurate the points are
    #placed but you have to be very patient :D
    for i in path:
        point2send = transform_points_from_image2real (i)
        packedData=vrep.simxPackFloats(point2send.flatten())
        raw_bytes = (ctypes.c_ubyte *
len(packedData)).from_buffer_copy(packedData)
        returnCode=vrep.simxWriteStringStream(clientID,
"path_coord", raw_bytes, vrep.simx_opmode_oneshot)
        time.sleep(sleep_time)
```

Robot Model

Remember at the PID part up there that we said we will use PID to control the linear and rotational velocities of the robot, well the robot has 2 motors and we need to convert the linear and rotational velocities into rotational velocities for each motor, and that's when the robot model comes in

```
d = 0.331 #wheel axis distance
r_w = 0.09751 #wheel radius
def pioneer_robot_model(v_des, omega_des):
    v_r = (v_des+d*omega_des)
    v_l = (v_des-d*omega_des)
    omega_right = v_r/r_w
    omega_left = v_l/r_w
    return omega_right, omega_left
```

Doing everything

Now lets put everything together to make this robot follow that path :D

First lets start by setting the look ahead distance and starting a vrep connection

```
lad = 0.5 #look ahead distance
print ('Starting Connection')
vrep.simxFinish(-1) # just in case, close all opened connections
clientID=vrep.simxStart('127.0.0.1',19997,True,True,5000,5) #
Connect to V-REP
print(clientID)
```

then we get the handles of all the objects in vrep and do some initialization

```
res,camera0_handle =
vrep.simxGetObjectHandle(clientID,'top_view_camera',vrep.simx_opmode
_oneshot_wait)
res_l,right_motor_handle =
vrep.simxGetObjectHandle(clientID,'Pioneer_p3dx_rightMotor',vrep.sim
x_opmode_oneshot_wait)
res_r,left_motor_handle =
vrep.simxGetObjectHandle(clientID,'Pioneer_p3dx_leftMotor',vrep.simx
_opmode_oneshot_wait)
res_las,look_ahead_sphere =
vrep.simxGetObjectHandle(clientID,'look_ahead',vrep.simx_opmode_ones
hot_wait)
indx = 0
err = 10 #to make a knida do while out of while
theta = 0.0
dt = 0.0
count = 0
om_sp = 0
d_controller   = pid(kp=0.5, ki=0, kd=0)
omega_controller = pid(0.5, 0., 0.)
```

Now we perform the path planning part, first we set the motors velocities to zero just in case ;) then we get an image for vrep form which we get the robot’s position, the obstacle’s grid and the goal position, then we do planning and smoothing then we transform the path from image

coordinates to Vrep coordinates, then we send the path points to be drawn in vrep

```
while err != vrep.simx_return_ok:
    tick = time.time()
    errorCode_leftM = vrep.simxSetJointTargetVelocity(clientID,
left_motor_handle, 0, vrep.simx_opmode_oneshot)
    errorCode_rightM = vrep.simxSetJointTargetVelocity(clientID,
right_motor_handle,0, vrep.simx_opmode_oneshot)

err,resolution,image=vrep.simxGetVisionSensorImage(clientID,camera0_
handle,0,vrep.simx_opmode_streaming)
    if err == vrep.simx_return_ok:
     img = np.array(image,dtype=np.uint8)
     img.resize([resolution[1],resolution[0],3])
     img_obs = obstacles_grid(img)
     center_robot = get_robot_position(img)
     img_goal, center_goal = get_goal_position(img)
     _ , path = search(img_obs,center_robot,center_goal,cost = 1, D
= 1 , fnc='Manhattan')
     newpath = smooth(path,img_obs, weight_data = 0.1, weight_smooth
= 0.65, number_of_iter = 1000)
     path_to_track = transform_points_from_image2real(newpath)
     tock = time.time()
     dt = tock - tick
     print("planning takes : ", dt)
     send_path_4_drawing(newpath, 0.05)
     center_goal =
transform_points_from_image2real(np.array(center_goal))
```

Then there is the loop responsible for moving the robot, first we get a new image of the scene from which we get the robot's position and orientation,

then we transform the path to the robot's frame, then we get the point on the path which is look ahead distance ahead of the robot on the path, then we mark this point using a yellow sphere -just to visualize the point-, then we get the angle between the robot and the point and the distance between the robot and the point, then we pass the angle to a PID controller that controls the rotational velocity of the robot and the distance is passed to a different PID controller to control the robot's speed, then the robot's model is used to obtain the speed of each wheel which is then passed to Vrep to move the robot, we do all this till the robot is near the goal by a small distance then we stop the robot.

```
while not is_near(center_robot, center_goal, dist_thresh = 0.25):
            tick = time.time()

err,resolution,image=vrep.simxGetVisionSensorImage(clientID,camera0_
handle,0,vrep.simx_opmode_streaming)
            if err == vrep.simx_return_ok:
                img = np.array(image,dtype=np.uint8)
                img.resize([resolution[1],resolution[0],3])
                center_robot = get_robot_position(img)
                center_robot =
transform_points_from_image2real(np.array(center_robot))
                theta = get_robot_orientation(img)
                theta = np.arctan2(np.sin(theta), np.cos(theta))
                path_transformed =
transform2robot_frame(center_robot, path_to_track, theta)
                dist = get_distance(path_transformed,
np.array([0,0]))
                #loop to determine which point will be the carrot
                for i in range(dist.argmin(), dist.shape[0]):
```

```
                    if dist[i] < lad and indx <= i:
                        indx = i
                #mark the carrot with the sphere

returnCode=vrep.simxSetObjectPosition(clientID,look_ahead_sphere,-1,
(path_to_track[indx,0], path_to_track[indx,1],
0.005),vrep.simx_opmode_oneshot)
                orient_error = np.arctan2(path_transformed[indx,1],
path_transformed[indx,0])
                #the controllers
                v_sp = d_controller.control(dist[indx])
                om_sp =omega_controller.control(orient_error)
                vr, vl = pioneer_robot_model(v_sp, om_sp)
                errorCode_leftM =
vrep.simxSetJointTargetVelocity(clientID, left_motor_handle, vr,
vrep.simx_opmode_oneshot)
                errorCode_rightM =
vrep.simxSetJointTargetVelocity(clientID, right_motor_handle,vl,
vrep.simx_opmode_oneshot)
                count += 1
                tock = time.time()
                dt = tock - tick
                print(dt)
        else:
            print("GOAAAAAAALL !!")
            errorCode_leftM =
vrep.simxSetJointTargetVelocity(clientID, left_motor_handle, 0,
vrep.simx_opmode_oneshot)
            errorCode_rightM =
vrep.simxSetJointTargetVelocity(clientID, right_motor_handle,0,
vrep.simx_opmode_oneshot)
```

Robotics Path Planning Path Tracking Pid Vrep