B4M36UIR & BE4M36UIR

Lab05 - Randomized Sampling-based Algorithms

Motivations and Goals
Become familiar with sampling-based motion planning
Understand the probabilistic roadmap approach and rapidly growing random tree approach
Tasks (teacher)

Motivations and Goals

Implement the RRT and PRM approaches for robot path planning in the maze

Lab resources

Task05 resource package

Lab code for task05 PRM algorithm

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import sys
import math
import numpy as np

import collections
import heapq

import matplotlib.pyplot as plt

import Environment as env

class PRMPlanner:

    def __init__(self, limits):
        """
        Parameters
        ----------
        limits: list((float, float))
            translation limits in individual axes
        """
        self.limits = limits
        x_lower = limits[0][0]
        x_upper = limits[0][1]
        y_lower = limits[1][0]
        y_upper = limits[1][1]
        z_lower = limits[2][0]
        z_upper = limits[2][1]
        #calculate the maximum step length
        self.max_translation = 1/250.0 * np.max([ x_upper-x_lower, y_upper-y_lower, z_upper-z_lower ])
        self.max_rotation = math.pi/6

    def plan(self, environment, start, goal):
        """
        Method to plan the path

        Parameters
        ----------
        environment: Environment
            Map of the environment that provides collision checking
        start: numpy array (4x4)
            start pose of the robot given in SE(3)
        goal: numpy array (4x4)
            goal pose of the robot given in SE(3)

        Returns
        -------
        list(numpy array (4x4))
            the path between the start and the goal Pose in SE(3) coordinates
        """
        path = []
```

```
        #random sample x points in the configuration space
        #for RRT iterate for the specific number of iterations
        n_points = 300

        #sample the points in the full configuration space - we are sampling in the full space
        samples = np.random.rand(6,n_points)

        #apply the limits in individual axes
        i = 0
        for limit in self.limits:
            scale = limit[1] - limit[0]
            samples[i,:] = samples[i,:]*scale + limit[0]
            i += 1

        #add start and goal sonfigurations to the samples
        samples = np.append(samples, start.reshape(6,1), axis=1)
        samples = np.append(samples, goal.reshape(6,1), axis=1)

        #debug plot samples
        #environment.plot_environment(P_start)
        #plt.plot(samples[0,:],samples[1,:],samples[2,:],'r.')

        #check the transition between individual configurations and filter the ones that are not feasible
        #it is recommended to construct the navigation graph in this step, for that, you will need a container to save the informatio

        tx = samples.shape[1]
        #in this implementation make the full transition graph
        for i in range(0,tx-1):
            for j in range(i+1,tx):
                p1 = samples[:,i]
                p2 = samples[:,j]

                #check, that there is a feasible path between the two configurations
                dist, edge = self.check_path_for_collision(environment, p1, p2)

                #if there is the path, add it to the list of feasible edges
                if dist > 0:
                    #add it to the oriented graph

                    #debug plot the path
                    #xx = [x[0,3] for x in edge]
                    #yy = [x[1,3] for x in edge]
                    #zz = [x[2,3] for x in edge]
                    #plt.plot(xx,yy,zz,'b', linewidth=0.5)

        #plan the path on the resulting graph -> result is a sparse path

        #reconstruct the dense path to adhere to the distance limit between the samples, i.e. recalculate the proper path pose matric

        return(path)

    def construct_pose(self, state):

        R = self.rotation_matrix(state[5],state[4],state[3])
        T = state[0:3]

        #construct the SE(3) matrix
        P = np.hstack((R,T.reshape((3,1))))
        P = np.vstack((P,[0,0,0,1]))

        return P

    def check_path_for_collision(self, environment, sample1, sample2):
        dist = 0
        edge = []
```

```
        #Note, decouple the translation and rotation parts ...

        #get the translation direction towards the target
        T_start = sample1[0:3]
        T_goal = sample2[0:3]
        T_dir = (T_goal-T_start)

        #get how many samples there is necessary to sample along the line to adhere to the maximum translation limit
        dist = np.linalg.norm(T_dir)
        samples_translation = int(dist/self.max_translation)

        #get the difference in the rotation angles
        dphi_x = sample1[3] - sample2[3]
        dphi_y = sample1[4] - sample2[4]
        dphi_z = sample1[5] - sample2[5]

        #get how many samples there is necessary to sample the rotation to adhere to the maximum rotation limit
        samples_rotation = int(np.max([np.abs(dphi_x/(self.max_rotation)), np.abs(dphi_y/(self.max_rotation)), np.abs(dphi_z/(self.ma

        #select the maximum of sampling
        n_points = np.max([samples_translation, samples_rotation])

        # iterate and check for collision
        for i in range(0,n_points):

            #construct the translation part of the pose
            T = T_start + T_dir*i/n_points

            #construct the rotation part of the pose
            yaw = ...
            pitch = ...
            roll = ...

            #construct the SE(3) pose matrix
            P = ...

            #check for collision with the environment
            ret = environment.check_robot_collision(P)

            #if in collision return...
            if ret:
                return 0, []
            else:
                edge.append(P)

        #don't forget to calculate the distance

        #return
        return dist,edge

    def rotation_matrix(self, yaw, pitch, roll):
        """
        Constructs rotation matrix given the euler angles
        yaw = rotation around z axis
        pitch = rotation around y axis
        roll = rotation around x axis

        Parameters
        ----------
        yaw: float
        pitch: float
        roll: float
            respective euler angles
        """
        R_x = np.array([[1, 0, 0],
                        [0, math.cos(roll), math.sin(roll)],
```

```
                [0, -math.sin(roll), math.cos(roll)]])
R_y = np.array([[math.cos(pitch), 0, -math.sin(pitch)],
                [0, 1, 0],
                [math.sin(pitch), 0, math.cos(pitch)]])
R_z = np.array([[math.cos(yaw), math.sin(yaw), 0],
                [-math.sin(yaw), math.cos(yaw), 0],
                [0, 0, 1]])

R = R_x.dot(R_y).dot(R_z)
return R
```