B4M36UIR & BE4M36UIR

Task05 - Randomized sampling-based algorithms

The main task is to implement the basic randomized sampling-based algorithms - RRT and PRM.

Deadline	17. November 2018, 23:59 PST
Points	6
Label in BRUTE	Task05
Files to submit	archive with `samplingplanner` directory
	Minimal content of the archive: `samplingplanner/PRMPlanner.py` , `samplingplanner/RRTPlanner.py`
Resources	Task05 resource package
	The resource package has been updated on 5. November 2018. The original version of the package is accessible here

Additional environments to test the planner

The assignment and deadline for this task are postponed for one week to Lab06 and 17.11. respectively

Assignment

Implement the Probabilistic Roadmap (PRM) and Rapidly Exploring Random Trees (RRT) randomized sampling-based path planning algorithms according to the description and pseudocode presented in the Lecture 5. Randomized Sampling-based Motion Planning Methods. The algorithms shall provide a **collision free** path through the environment represented by a geometrical map.

In file `PRMPlanner.py` implement the PRM algorithm.
In file `RRTPlanner.py` implement the RRT algorithm.

The implementation requirements are as follows:

1. The `PRMPlanner` and `RRTPlanner` plan a path in 6-DOF
2. The `PRMPlanner` and `RRTPlanner` implements function `plan` that takes following arguments on the input:
   - `environment` - an instance of the `Environment` class that provides the interface for collision checking between the robot and the obstacles
   - `start` and `goal` - the initial and `goal` configurations of the robot. Each configuration is given as a tuple of 6 state-space variables $(x, y, z, \phi_x, \phi_y, \phi_z)$, where x, y, z represent the position of the robot in the environment. $\phi_x, \phi_y, \phi_z \in (0, 2\pi)$ represent the orientation of the robot as the rotation angles around the respective axis
3. The output of the `plan` function is a list of robot poses given in $SE(3)$ which codes the full configuration of the robot into a single matrix
   - The pose $\mathbf{P} \in SE(3)$ is given as

     $$\mathbf{P} = \begin{bmatrix} \mathbf{R} & \mathbf{T} \\ [0,0,0] & 1 \end{bmatrix},$$

     where $\mathbf{R} \in \mathcal{R}^{3\times3}$ is the rotation matrix for which $\mathbf{R} \cdot \mathbf{R} = \mathbf{I}$ and $\det(\mathbf{R}) = 1$. $\mathbf{T} \in \mathcal{R}^3$ is the translation vector
   - The individual poses are the rigid body transformations in the global reference frame. Hence, the position of the robot r is given as the transformation of the robot base pose $\mathbf{r}_b$ in homogeneous coordinates, given as:

     $$\begin{bmatrix} \mathbf{r} \\ 1 \end{bmatrix} = \mathbf{P} \cdot \begin{bmatrix} \mathbf{r}_b \\ 1 \end{bmatrix}.$$

     Which can be also written as:

     $$\mathbf{r} = \mathbf{R} \cdot \mathbf{r}_b + \mathbf{T}.$$
4. The boundaries for individual configuration variables are given during the initialization of the planner in `self.limits` variable as a list of lower-bound and upper-bound limit tuples, i.e. `list( (lower_bound, upper_bound) )` , for each of the variables $(x, y, z, \phi_x, \phi_y, \phi_z)$
5. In both the PRM and RRT approaches the individual poses shall not be further than $\frac{1}{250}$ of the largest configuration dimension and the orientation between two consecutive path points shall not change for more than $\frac{\pi}{6}$ in any axis, i.e., the maximum translation between two poses is given by the maximum span of the x, y, z limits and two consecutive configurations may not differ for more than $\frac{\pi}{6}$ in any axis:

```
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]
max_translation = 1/250.0 * np.max([ x_upper-x_lower, y_upper-y_lower, z_upper-z_lower ])
```

   Note, The configuration space sampling is not affected by this requirement. Individual random samples may be arbitrarily far away; however, their connection shall adhere to the given constraint on the maximum distance and rotation to ensure sufficient sampling of the configuration space and smooth motion of the robot
6. The collision checking is performed using the `self.environment.check_robot_collision` function that takes on the input an $SE(3)$ pose matrix. The collision checking function returns `True` if there is collision between the robot and the environment and `False` if there is no collision.

Approach

The provided source files provides only the ability to check for the collision between the robot and the environment. The collision avoidance software used is RAPID[1)] collision checking library. Following instructions might be used to help solve the given assignment:

1. Implement a function for the construction of the pose matrix from the configuration vector, i.e., function that takes $(x, y, z, \phi_x, \phi_y, \phi_z)$ on the input and provides the pose matrix $\mathbf{P} \in SE(3)$ on its output. Such a function helps to interface the collision checking function of the `self.environment` and is necessary to produce the desired output of the `plan` method
2. Implement a path checking function, that samples poses between two configurations `start` and `goal` given the requirement on the maximum distance and the maximum rotation. To simplify further tasks, the function may return the distance between the `start` and the `goal` configuration and also a list of interlying configurations.
3. Do the random sampling in the full configuration space, i.e., generate random configurations for $(x, y, z, \phi_x, \phi_y, \phi_z)$ and apply boundary limits to these configurations to not accidentally leave the configuration space, e.g., for PRM the random sampling can look like:

```
#random sample n_points in the configuration space
n_points = 30
#random sampling from uniform distribution between 0 and 1
samples = np.random.rand(6,n_points)
#change the sampling based on the limits in individual axes - scale and shift the samples
i = 0
for limit in self.limits: #for each DOF in configuration
    scale = limit[1] - limit[0] #calculate the scale
    samples[i,:] = samples[i,:]*scale + limit[0] #scale and shift the random samples
    i += 1
```

4. Always try to plot the result of each step to verify its correctness

Evaluation

Following marks will be considered in evaluation

1. **(2 points)** working implementation of the PRM planner with the following stages:
   - random sampling of the configuration space
   - construction of the transition graph between individual configurations, adhering to the constraint on maximum distance and maximum rotation
   - planning the path on the resulting graph
   - providing the collision free path in SE(3) coordinates
2. **(2 points)** working implementation of the RRT planner with the following stages:
   - at each step, random sampling the configuration space and growing the tree in the direction of the sample (regardless whether the whole path, or just an increment), adhering to the constraint on maximum distance and maximum rotation
   - when the goal position is reached, backtracking the path in the constructed tree
   - provide the collision free path in SE(3) coordinates
3. Both the algorithms shall adhere to the maximum step length and maximum rotation between poses in the resulting path given by the configuration limits
4. **(2 point)** - The RRT algorithm is able to solve the alpha-puzzle problem

The simplified evaluation script for testing of the implementation is following

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

import sys
import math
import time
import numpy as np
import matplotlib.pyplot as plt
from collections import deque

sys.path.append('environment')
sys.path.append('samplingplanner')
```

```
import Environment as env
import PRMPlanner as prm
import RRTPlanner as rrt


if __name__ == "__main__":
    #define the planning scenarios
    #scenario name
    #start configuration
    #goal configuration
    #limits for individual DOFs
    scenarios = [("environments/simple_test", (2,2,0,0,0,0), (-2,-2,0,0,0,0), [(-3,3), (-3,3), (0,0), (0,0), (0,0), (0,0)]),
                 ("environments/simple_test", (2,2,0,0,0,0), (-2,-2,0,0,0,math.pi/2), [(-3,3), (-3,3), (0,0), (0,0), (0,0), (0,2*math
("environments/alpha_puzzle", (0,5,0,0,0,0),(25,25,25,0,0,0), [(-40,70),(-40,70),(-40,70),(0,2*math.pi),(0,2*math.pi),(0,2*math.pi)])

    #enable dynamic drawing in matplotlib
    plt.ion()

    #####################################
    ## EVALUATION OF THE RRT PLANNER
    #####################################
    for scenario in scenarios:
        name = scenario[0]
        start = np.asarray(scenario[1])
        goal = np.asarray(scenario[2])
        limits = scenario[3]

        print("processing scenario: " + name)

        #initiate environment and robot meshes
        environment = env.Environment()
        environment.load_environment(name)

        #instantiate the planner
        planner = rrt.RRTPlanner(limits)

        #plan the path through the environment
        path = planner.plan(environment, start, goal)

        #plot the path step by step
        ax = None
        for Pose in path:
            ax = environment.plot_environment(Pose, ax=ax, limits=limits)
            plt.pause(0.1)

    #####################################
    ## EVALUATION OF THE PRM PLANNER
    #####################################
    for scenario in scenarios:
        name = scenario[0]
        start = np.asarray(scenario[1])
        goal = np.asarray(scenario[2])
        limits = scenario[3]

        print("processing scenario: " + name)

        #initiate environment and robot meshes
        environment = env.Environment()
        environment.load_environment(name)

        #instantiate the planner
        planner = prm.PRMPlanner(limits)

        #plan the path through the environment
        path = planner.plan(environment, start, goal)
```

```
#plot the path step by step
ax = None
for Pose in path:
    ax = environment.plot_environment(Pose, ax=ax, limits=limits)
    plt.pause(0.1)
```

RAPID collision checking library installation notes

On Linux (tested with Ubuntu 14.04, 16.04, 18.04)

1. Download the resource package and `make` the rapid library in `environment/rapid` directory

On MacOS

1. On line 7 of `environment/rapid/Makefile` change `TARGET=librapid.so` to `TARGET=librapid.dylib`
2. On line 11 of `environment/rapid/Makefile` change `-soname` to `-install_name`

1)

http://gamma.cs.unc.edu/OBB/

courses/b4m36uir/hw/task05.txt · Last modified: 2018/11/19 13:42 by cizekpe6