B4M36UIR & BE4M36UIR

Task06 - Curvature-constrained local planning in RRT

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

Deadline	24. November 2018, 23:59 PST
Points	5
Label in BRUTE	Task06
Files to submit	archive with `samplingplanner` directory
	Minimal content of the archive: `samplingplanner/DubinsPRMPlanner.py` , `samplingplanner/DubinsRRTPlanner.py`
Resources	Task06 resource package
	Additional environments to test the planner

The assignment and deadline for this task are postponed for one week to Lab07 and 24.11. respectively

Assignment

Extend the previous work on Task05 - Randomized sampling-based algorithms to planning with a non-holonomic Dubins vehicle. Implement the Probabilistic Roadmap (PRM) and Rapidly Exploring Random Trees (RRT) randomized sampling-based path planning algorithms. The algorithms shall provide a **collision free** path through the environment represented by a geometrical map.

In file `DubinsPRMPlanner.py` implement the PRM algorithm that respects motion constraints of the Dubins vehicle.
In file `DubinsRRTPlanner.py` implement the RRT algorithm that respects motion constraints of the Dubins vehicle.

The implementation requirements are as follows:

1. The `DubinsPRMPlanner` and `DubinsRRTPlanner` plan a path in 3-DOF only - x, y position of the robot and ϕ heading
2. The `DubinsPRMPlanner` and `DubinsRRTPlanner` 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 3 state-space variables (x, y, ϕ), where x, y represent the position of the robot in the environment and $\phi \in (0, 2\pi)$ represent the orientation of the robot
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, ϕ)
5. In both the PRM and RRT approaches the individual poses shall not be further than $\frac{1}{250}$ of the largest configuration dimension, i.e., the maximum translation between two poses is given by the maximum span of the x, y limits:

   ```
   x_lower = limits[0][0]
   x_upper = limits[0][1]
   y_lower = limits[1][0]
   y_upper = limits[1][1]
   max_translation = 1/250.0 * np.max([ x_upper-x_lower, y_upper-y_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.
7. During the Dubins maneuver, the robot shall not leave the boundaries of the configuration space, i.e., the feasibility of each Dubins maneuver has to be checked for collision with the obstacles as well as for the given configuration limits.

Approach

The Dubins vehicle model connects two configurations of the robot by a shortest curve, with a constraint on the curvature of the path and with prescribed initial and terminal tangents to the path. Hence, each Dubins path consists of either turns with a predefined radius (right turn (R), left turn (L)) or straight-line segments (straight (S)). The optimal path type can be described using these primitives and it will always be at least one of the six types: RSR, RSL, LSR, LSL, RLR, LRL.

We will use the Python implementation of the Dubins path (`pip3 install dubins`) https://github.com/AndrewWalker/pydubins

For our purpose of the randomized sampling-based motion planning we need to sample the path into individual configurations. An example of such a sampling can be done as follows

```
#!/usr/bin/env python3

import dubins as dubins
import math
import matplotlib.pyplot as plt

x0 = 2
y0 = 2
theta0 = math.pi

x1 = 0
y1 = 0
theta1 = 0 #math.pi

turning_radius = 1.0
step_size = 0.1

plt.ion()

for i in range(0,100):
    theta1 = i*math.pi/50.0

    q0 = (x0, y0, theta0)
    q1 = (x1, y1, theta1)

    #calcuating the shortest path between two configurations
    path = dubins.shortest_path(q0, q1, turning_radius)
    print("Path maneuver: " + str(path.path_type()) + ", Path length: " + str(path.path_length()))

    #sampling the path
    configurations, _ = path.sample_many(step_size)

    #quiver plot of the path
    l = 0.2
    xx = [x[0] for x in configurations]
    yy = [x[1] for x in configurations]
    uu = [math.cos(x[2])*l for x in configurations]
    vv = [math.sin(x[2])*l for x in configurations]

    plt.clf()
    plt.quiver(xx,yy,uu,vv)
    plt.axis('equal')
    ax = plt.gca()
    ax.axis([-3,3,-3,3])
    plt.draw()
    plt.pause(0.1)
```

Evaluation

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 DubinsPRMPlanner as dubinsprm
import DubinsRRTPlanner as dubinsrrt


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,math.pi/2), [(-3,3), (-3,3), (0,0), (0,0), (0,0), (0,2*math
                 ("environments/squares", (2,-3,0,0,0,0), (4,8,0,0,0,0), [(0,5), (-5,10), (0,0), (0,0), (0,0), (0,2*math.pi)]),
                 ("environments/maze1", (2,2,0,0,0,math.pi/4), (35,35,0,0,0,math.pi/2), [(0,40), (0,40), (0,0), (0,0), (0,0), (0,2*ma
                 ("environments/maze2", (2,2,0,0,0,math.pi/4), (35,26,0,0,0,3*math.pi/2), [(0,40), (0,40), (0,0), (0,0), (0,0), (0,2*
                 ("environments/maze3", (2,2,0,0,0,0,0), (25,36,0,0,0,0), [(0,40), (0,40), (0,0), (0,0), (0,0), (0,2*math.pi)]),
                 ("environments/maze4", (2,2,0,0,0,0), (27,36,0,0,0,math.pi/2), [(0,40), (0,40), (0,0), (0,0), (0,0), (0,2*math.pi)])

    #enable dynamic drawing in matplotlib
    plt.ion()

    ########################################
    ## EVALUATION OF THE DUBINS 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 = dubinsrrt.DubinsRRTPlanner(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(0.01)

    ########################################
    ## EVALUATION OF THE DUBINS 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 = dubinsprm.DubinsPRMPlanner(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(0.01)
```

courses/b4m36uir/hw/task06.txt · Last modified: 2018/11/20 08:56 by cizekpe6