B4M36UIR & BE4M36UIR

Task01 - Open-loop locomotion control

The main task is to implement a function that will steer the robot towards a given goal.

Deadline	13. October 2018, 23:59 PST
Points	3
Label in BRUTE	Task01
Files to submit	archive with `Robot.py` file and optionally `RobotConst.py` file
Resources	Task01 resource package

Assignment

In class `Robot.py` implement the `goto(coord, phi=None)` function. The purpose of the function is to navigate the robot towards the goal given by coordinates $coord = (x_{goal}, y_{goal})$ with an optional final heading ϕ_{goal}. The steering of the locomotion is achieved by the `goto` function by setting the differential steering command of the CPG locomotion controller (v_{left}, v_{right}). The function returns `True` when the robot is at the goal coordinates and `False` if it has collided with an obstacle en route.

Information about the current position (x, y), orientation ϕ and collision state is provided by the `RobotHAL` interface through the `self.robot` object. The respective functions are

```
#get position of the robot as a tuple (float, float)
self.robot.get_robot_position()
#get orientation of the robot as float
self.robot.get_robot_orientation()
#get collision state of the robot as bool
self.robot.get_robot_collision()
```

The `goto` function has a following prescription

```
def goto(self, coord, phi=None):
    """
    open-loop navigation towards a selected goal, with an optional final heading

    Parameters
    ----------
    coord: (float, float)
        coordinates of the robot goal
    phi: float, optional
        optional final heading of the robot

    Returns
    -------
    bool
        True if the destination has been reached, False otherwise
    """
```

Approach

The open-loop locomotion towards a given goal can be approached either using a discrete regulator, or using a continuous function.

The discrete regulator operates as follows (pseudocode).

```
while not goal_reached:
    if the difference between the current heading and the heading to the target is higher than ORIENTATION_THRESHOLD:
        full speed turn towards the targets
    else:
        go straight
```

On the other hand, the continuous navigation function is much more elegant and can look like e.g. (pseudocode):

```
while not goal_reached:
    dphi = the difference between the current heading and the heading towards the target
    v_left = -dphi*C_TURNING_SPEED + BASE_SPEED
    v_right = dphi*C_TURNING_SPEED + BASE_SPEED
```

Where `C_TURNING_SPEED` is a constant that defines the aggression with which the robot will turn towards the desired heading and `BASE_SPEED` is the default speed of the robot when it is heading directly towards the target. Note, the continuous navigation function is inspired by the Braitenberg vehicle model which will be discussed during Lab02 - Exteroceptive sensing, Mapping and Reactive-based Obstacle Avoidance.

Also note, that in a physical world it is impossible to get to a precise specific coordinates, therefore it is sufficient to navigate “close enough”. The sufficient distance should be comparable in size to the actual robot. In our case, this distance is given as the navigation parameter `DISTANCE_THLD = 0.1 #m` defined in the `RobotConst.py` file.

Evaluation

The code can be evaluated using the following script

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

import sys
import math
import numpy as np
sys.path.append('robot')

import Robot as rob

DISTANCE_THLD = 0.15 #m
ORIENTATION_THLD = math.pi/8

def check(pose_set, pose_real):
    """
    Function to check that the navigation towards the goal has been successfull

    Parameters
    ----------
    pose_set: (float, float, float)
        desired coordinates to reach (x,y,phi)
    pose_real: (float, float, float)
        real coordinates of the robot (x,y,phi)

    Returns
    -------
    bool
        True if the robot real position is in delta neighborhood of the desired position, False otherwise
    """
    ret = True
    (x1, y1, phi1) = pose_set
    (x2, y2, phi2) = pose_real
    dist = (x1 - x2)**2 + (y1-y2)**2
    #check the distance to the target
    if math.sqrt(dist) > DISTANCE_THLD:
        ret = False
    #check the final heading
    if not phi1 == None:
        dphi = phi1 - phi2
        dphi = (dphi + math.pi) % (2*math.pi) - math.pi
        if dphi > ORIENTATION_THLD:
            ret = False

    return ret


if __name__=="__main__":
    robot = rob.Robot()

    #navigation points
    route = [(1,1,None), (-1,1,math.pi/2), (-1,-1,None), (1,-1,None)]

    #navigate
    for waypoint in route:
        pos_des = waypoint[0:2]
        ori_des = waypoint[2]

        print("Navigation point " + str(pos_des) + " " + str(ori_des))
        #navigate the robot towards the target
        status1 = robot.goto(pos_des,ori_des)

        #get robot real position
        pose_real = robot.get_pose()

        #check that the robot reach the destination
        status2 = check(waypoint, pose_real)
        #print the result
        print(status1, status2)
```

The expected output on `obstacle.ttt` map is

```
Connected to remote API server
Robot ready
Navigation point (1, 1) None
True True
Navigation point (-1, 1) 1.5707963267948966
True True
Navigation point (-1, -1) None
True True
Navigation point (1, -1) None
False False
```

courses/b4m36uir/hw/task01.txt · Last modified: 2018/10/08 14:42 by cizekpe6