Foundations of Robotics, Fall 2017

Coding Homework 3: Control Design and Optimization Methods

(CS 4750: 80 points, CS 5750: 100 points)

Due at the start of class, Mon, 13 Nov

Objective:

Demonstrate understanding of methods of control design and optimization for robotics applications.

Instructions:

This assignment can be discussed as a group of arbitrary size. You may work with up to one partner on this assignment, but each student is responsible for writing and submitting a separate solution, and **no student should see another student's solution**. Include in a comment at the top of each file your name and the names of all classmates you discussed any part of the homework with.

Use the simulator tool to run the provided code for the assignment. **For this assignment, you will always need to run `./simulator-tool run hw3 main` to run the framework code.** We suggest using the timeout option (see Piazza post @424) so that the simulator runs for long enough.

If you encounter difficulty using the simulator tool, please contact the course staff. Note that you can also run `./simulator-tool --help` to get a listing of commands and `./simulator-tool <command> --help` to get more information on the usage of a specific command.

Assignment:

Implement code to solve the following problems, using the provided simulator framework. Start by making a `catkin` workspace for your code. Create a new ROS package for this project in the workspace (call it "hw3"); write your code for each problem in a corresponding file in the package (named according to the convention "pX.py", where X is the problem number).

The overall goal of this assignment is to enable a KUKA youBot to (a) navigate in a workspace in a collision-free fashion and (b) pick up blocks from the ground and place them into an optimally-positioned bucket. To see how the various components of this assignment will come together to accomplish this task, launch the provided implementation with `roslaunch foundations_hw3 p3.launch` (note that the main framework code must also be running). As you complete the various components of this assignment, you can test each one by replacing the corresponding service we provide in the launch file `foundations_hw3/p3.launch` with your implementation of the service.

Recommended Milestone Schedule (if you intend to complete all problems):

Oct. 27 Workspace Optimization

Nov. 3 Navigation I: Path Planning

Nov. 8 Navigation II: Pure Pursuit

Nov. 13 Pick-and-Place Robot

Submission Format:

Your `hw3` package will include four services (`p1a`, `p1b`, `p2a`, and `p2b`, each implemented in a separate Python file named according to the convention given above), a script called `p3.py` that combines a subset of these service, and a launch file called `p3.launch`. We will test each of your services with `rosrun` (e.g. `rosrun hw3 p1a.py`) and your launch file with `roslaunch hw3 p3.launch`. Ensure that your code runs with these commands prior to submission. Compress your `hw3` package into a .tar.gz archive with the command `tar -czvf solutions.tar.gz hw3`. Double-check that your submission is a single, correctly-structured ROS package and that it builds. **We reserve the right to penalize for incorrectly structured submissions.**

Warnings:

The KUKA youBot moves very slowly in the V-REP simulator. If you speed up the simulation, please be advised that V-REP may miss ROS messages. In addition, we would also like to note that for the last problem on this assignment, it is okay if the youBot fails to grasp a block as long as it is reaching for the right point. Finally, it is OK if the youBot runs into objects on its way to a block; the cost function we provide for generating paths does not guarantee collision-free paths. If there are any changes to the assignment, we will post on Piazza and update this document on CMS. If you aren't sure how to use or call one of the provided services, start by reading the `.srv` file defining the service format, and remember that you can use `rosservice call` to experiment with services on the command line.

Other helpful notes:

- There are 5 blocks in the simulation. We will not be changing this number during evaluation of your submissions.
- You should experiment with the number of waypoints for your paths in problem 2(a). We have found 3 to be a good number, but it may be interesting for you to play around with this a bit. See how adding waypoints changes the paths the robot takes!

1 Workspace Optimization (35 points)

Consider the workspace of Fig. 1. Given the positions of the square blocks, your goal is to determine where to place the circular bucket in order to minimize the distance traveled by the youBot. You can treat this problem as a minimization of the average distance between the bucket and the blocks. Let $x_0 \in \mathbb{R}^2$ denote the center of the bucket and $x_i \in \mathbb{R}^2$ the center of the i-th block. Then, given a workspace with m blocks, you can formally define this problem with the optimization scheme

$$x_0^* = \underset{x_0 \in \mathbb{R}^2}{\operatorname{argmin}} \frac{1}{m} \sum_{i=1}^{m} ||x_0 - x_i||^2. \tag{1}$$

(a) (*25 points*) Using the CVXOPT optimization software (`http://cvxopt.org`) that has been installed on your VM, create a service that solves the aforementioned optimization problem. To get the position of block `X`, where $1 \leq$ `X` ≤ 5, call the service `/vrep/blockX/pos/get`. The service you write should set the position of the bucket to its optimal location with a call to the `/vrep/bucket/pos/set` service. All positions are of type `geometry_msgs/Point`. Your service should have the type `foundations_hw3/PositionBucket` (we have provided the necessary service definition).

(b) (*10 points*) Create a service that solves the same optimization problem as before but with the added constraint that the bucket cannot be placed more than one meter from the starting location of the youBot, where the distance is measured using the Manhattan distance metric. Note that all distance units in V-REP are in meters, and **note that you should use Manhattan distance instead of Euclidean distance for the constraints only – The cost function should still be computed with Euclidean distance**. The current pose of the youBot is published to the topic `/vrep/youbot/base/pose`; we suggest using `rospy.wait_for_message()` to obtain the pose since you need only a single message from this topic, not the continuous stream of messages that a subscriber would provide. Documentation for this method is available at http://docs.ros.org/lunar/api/rospy/html/rospy.client-module.html.

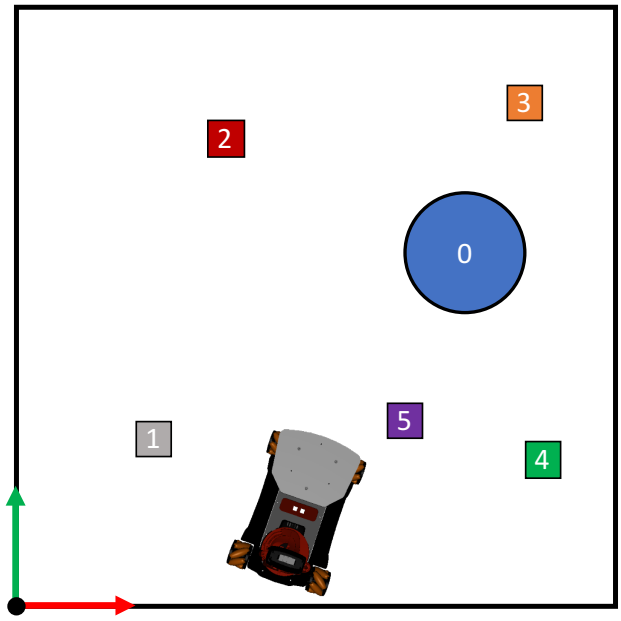

Figure 1: Overhead view of a KUKA youBot, circular bucket, and five blocks in a workspace.

2 Navigation (55 points)

Consider the problem of navigating the youBot through the workspace without colliding into obstacles (the blocks and the bucket) while also avoiding unnecessarily long paths. Your task involves planning which path to take and designing a controller that allows the robot to follow this path.

Step One: Path Planning

Given the positions $x_i \in \mathbb{R}^2$, $i = 0, \ldots, m$ of the $m + 1$ obstacles in the workspace (that is, the positions of the m blocks and the position of the bucket) as well as the youBot's start position s and intended goal position g, your objective is to determine a collision-free path of small length. The path will be a set of n waypoints $P = (p_1, \ldots, p_n)$ starting with $p_1 = s$ and ending with $p_n = g$. We can consider this to be an optimization problem in an appropriately-defined space of paths $\mathcal{P}$ that compromises between two considerations:

1. a cost $C : \mathcal{P} \to \mathbb{R}$ that penalizes small distances to obstacles, and
2. a cost $L : \mathcal{P} \to \mathbb{R}$ that penalizes long paths.

The first cost can be implemented as

$$C(P) = \sum_{i=0}^{m} \sum_{j=2}^{n-1} \frac{1}{d_{ij}^2}, \tag{2}$$

where d_{ij} is the Euclidean distance between waypoint j and obstacle i.

The second cost can be implemented as

$$L(P) = \sum_{j=2}^{n} ||p_j - p_{j-1}||^2. \tag{3}$$

In order to find the right balance between these two costs, we can introduce a weighting factor a. Then we can formally define this optimization problem as

$$P^* = \underset{P \in \mathcal{P}}{\operatorname{argmin}}\, a * C(P) + (1 - a)L(P). \tag{4}$$

The minimization of this weighted sum will result in a collision-free path of small length, as shown in Fig. 2.

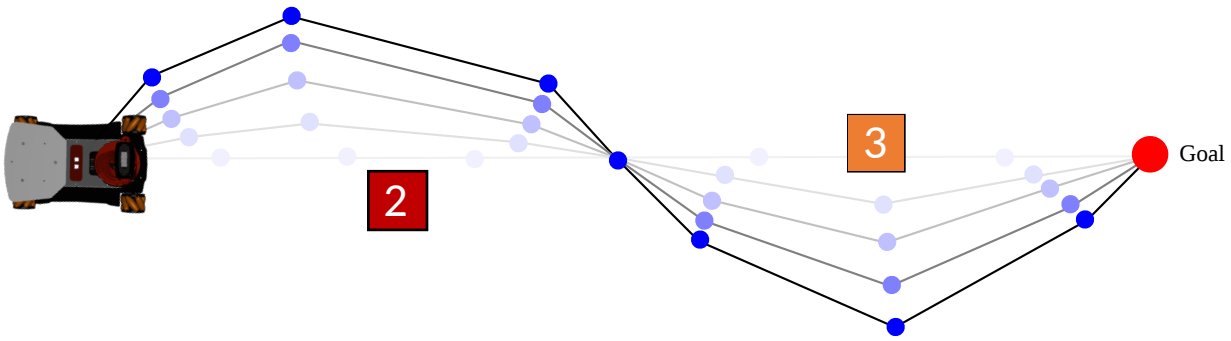

Figure 2: Gradient descent on the shape of a path using a set of waypoints and an objective function that trades off between minimal path length and maximal distance from obstacles.

Note that this formulation treats all objects as points. To avoid collisions, we need to introduce geometrical models of the objects. A simple and practical way to do this is to enclose each object in a "bubble". For example, each block may be enclosed in a circle with a radius equal to its diagonal, and the youBot may also be enclosed in an appropriately-defined circle. Fig. 3 illustrates how the distance computation differs when accounting for object size in this manner.

Step Two: Waypoint Following

Once a path has been planned, the youBot needs to follow the set of waypoints in order to get from the start position to the goal position. An algorithm that will accomplish this is the *pure pursuit* algorithm, described in Section 6.6 of the course notes (Path-Following Controllers). The algorithm gets its name from the analogy that the robot is "pursuing" a moving point some distance in front

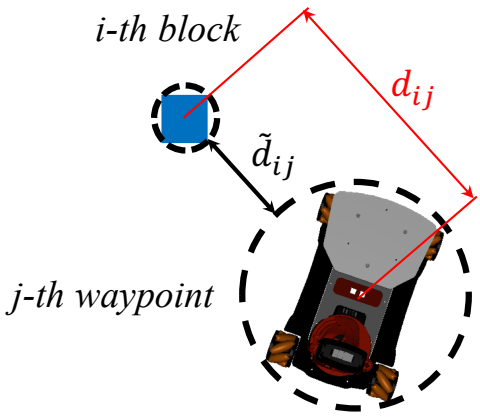

Figure 3: Computing the distance between the i-th block and the j-th waypoint as d_{ij} could result in the youBot colliding with the block if x_i and p_j are too close. To prevent these collisions, we can use a distance approximation $\tilde{d}_{ij}$ that accounts for the size of each object.

of it along the path, much like how a person driving a car looks at the road some distance ahead and drives toward that spot. The algorithm works by calculating the curvature that will move the robot from its current position to the point it is pursuing.

Two services are provided to help with your implementation. The `/interpolate_path` service takes a path and a distance along the path, and it returns a `geometry_msgs/Point` located at that distance from the start of the path. The `/closest_point_path` service takes a path as well as a point not necessarily on the path. It returns a closest point that lies on the path (note that this is not necessarily unique!), the distance along the path from the start to this closest point, and the distance between the two points.

(a) (*35 points*) Create a service that uses gradient descent to plan a path of n waypoints from a start position s to a goal position g. To get the cost of a set of waypoints, use the provided service `/compute_cost`, which computes the weighted sum given in (4) except that it replaces the distance d_{ij} with the distance approximation $\tilde{d}_{ij}$. The service that you write should have type `foundations_hw3/FindPath`; be sure to read the corresponding description file before starting your implementation to see what the inputs and outputs of your service need to be.

(b) (*20 points*) Design a service that drives the youBot along a set of waypoints using the pure pursuit algorithm. The service you write should have type `foundations_hw3/FollowPath`. Prevent the youBot from moving along arbitrary vectors by restricting your velocity commands to use a linear velocity in the y-axis and an angular velocity in the z-axis.

3 Pick-and-Place Robot (10 points)

Now you will use your services (and some of ours) to build a pick-and-place robot. To do this, implement the following algorithm in a Python script called `p3.py`. Then create a launch file called `p3.launch` that launches your `p3.py` script as well as the nodes for your services.

1. Optimally place the bucket with your `p1b` service.

2. Plan a path to a block with your `p2a` service.
3. Navigate to the block using your `p2b` service.
4. Orient the youBot to be orthogonal to the block; that is, turn the youBot by publishing to `/vrep/youbot/ base/cmd_vel` until its heading (which you can get from the topic `/vrep/youbot/base/pose`) is parallel to one of the axes of the block's orientation (which you can get from the service `/vrep/block#/pos/get` for a particular block number.).
5. Move the youBot's end effector to the block with the `/vrep/youbot/arm/reach` service.
6. Close the gripper on the block with the `/vrep/youbot/gripper/grip` service.
7. Publish to the arm joint angles so that the youBot is holding the block straight up in the air.
8. Plan a path to the bucket with your `p2a` service.
9. Navigate to the bucket using the `p2b` service.
10. Move the youBot's end effector over the bucket with the `/vrep/youbot/arm/reach` service.
11. Open the gripper with the `/vrep/youbot/gripper/grip` service.
12. Repeat steps 2–11 until all of the blocks are in the bucket.

Note that if you do not complete one of the previous problems and are therefore missing a service, use our corresponding implementation of the service in your launch file so that you are able to complete the algorithm. Our implementations may be found in `foundations_hw3/bin`. Also note that the order in which you pick up the blocks does not matter. Our implementation starts with `block1` and goes in order of index; feel free to also use this ordering.