# Foundations of Robotics, Fall 2018

### Coding Project 4: Control Design and Optimization Methods
(CS 4750: 95 points, CS 5750: 100 points)

**Due Mon, Nov. 12 at 3:00 PM**

**Objective:**

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

**Collaboration Policy:**

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 project with.

**Using the Simulator Tool:**

Use the simulator tool to run the provided code for the assignment. **For this assignment, you will always need to run `./simulator-tool run p4 main` to run the framework code. Remember that to start the simulation, you must hit the play button.**

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.

**Debugging Tips:**

If you add `output="screen"` to your launch file after you specify `type="something.py"`, the print statements from that Python file will print to your terminal. We recommend using `rostopic echo` to see what messages are being published. Refer to the ROS documentation for more information.

We will have a FAQ and clarifications post for this assignment pinned on Piazza, so please check that post regularly for updates and information.

**Assignment:**

The goal of this assignment is to implement code 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 p4 problem3_solution.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 `launch/problem3.launch` with your implementation of the service.

When you have finished the assignment, compress your complete ROS package with the command `./simulator-tool package p4` and upload the compressed package (named `p4_solutions.tar.gz`)

to CMS. Note that you can upload to CMS as many times as you'd like up until the deadline; the most recent submission will override all the previous ones. Keep this in mind and don't wait until the last minute to submit! We recommend uploading at least a half hour before the deadline to ensure that your solutions are submitted on time.

Remember to run `./simulator-tool check p4` prior to submitting, which verifies that your submission is runnable and can be graded correctly. If this command returns errors or warnings, you will lose points on the assignment. Bear in mind that a successful return from the `check` command does not guarantee full points on this assignment.

**Tips:**

- 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.
- There are 5 blocks in the simulation. We will not be changing this number during evaluation of your submissions.

# 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

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

## 1a: Unconstrained Optimization (*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. Your service should be called `problem1a` and be of type `p4/PositionBucket`. To get the position of block `X`, where $1 \leq \texttt{X} \leq 5$, 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`. All code for this problem should go in `src/problem1a.py`.

## 1b: Constrained Optimization (*10 points*)

Create a service named `problem1b` of type `p4/PositionBucket` that solves the same optimization problem as in problem 1a 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. All distance units in V-REP are in meters. **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

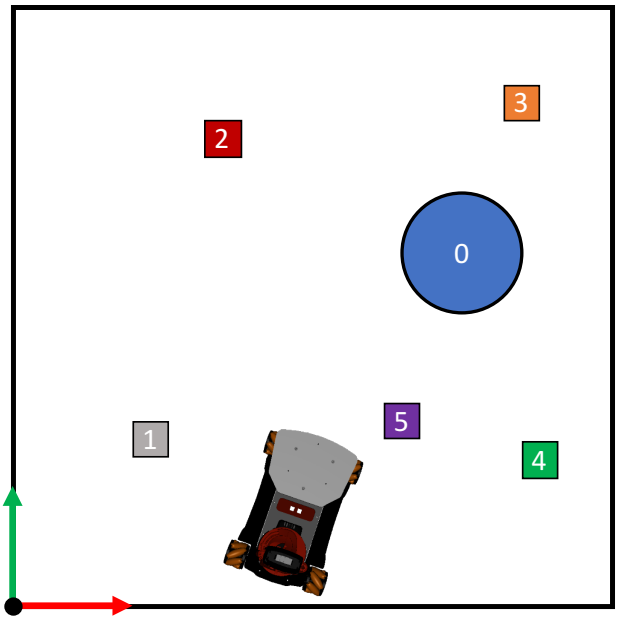


Figure 1: Overhead view of a KUKA youBot, circular bucket, and five blocks in a workspace.

at http://docs.ros.org/melodic/api/rospy/html/rospy.client-module.html. Your solution should go in `src/problem1b.py`.

**Grading:**

We will grade Problem 1a and Problem 1b based on the distance between the position your service gives and the position our service gives. In addition, for Problem 1b, part of the grade will be based on whether you satisfied the constraint.

## 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.

### 2a: Path Planning (*35 points*)

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 short length. The path will be a set of $n$ waypoints $P = (p_1, \ldots, p_n)$ starting with $p_1 = S$ and ending with $p_n = G$. This path describes a set of segments $T = ((p_1, p_2), \ldots, (p_{n-1}, p_n))$. 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

$$C(P) = \sum_{i=0}^{m} \sum_{s \in T} \begin{cases} \frac{1}{(d_{i,s} - r_i)^2} & \text{if } d_{i,s} > r_i + \epsilon \\ \frac{1}{\epsilon^2} K^{r_i + \epsilon - d_{i,s}} & \text{if } d \leq r_i + \epsilon \end{cases}, \tag{2}$$

where $s$ is a path segment between two adjacent waypoints in $P$, $d_{i,s}$ is the distance between segment $s$ and obstacle $i$, and $r_i$ is the radius of the circumscribed circle of obstacle $i$. $K$ is some very large constant, and $\epsilon$ is some very small constant.

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

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

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

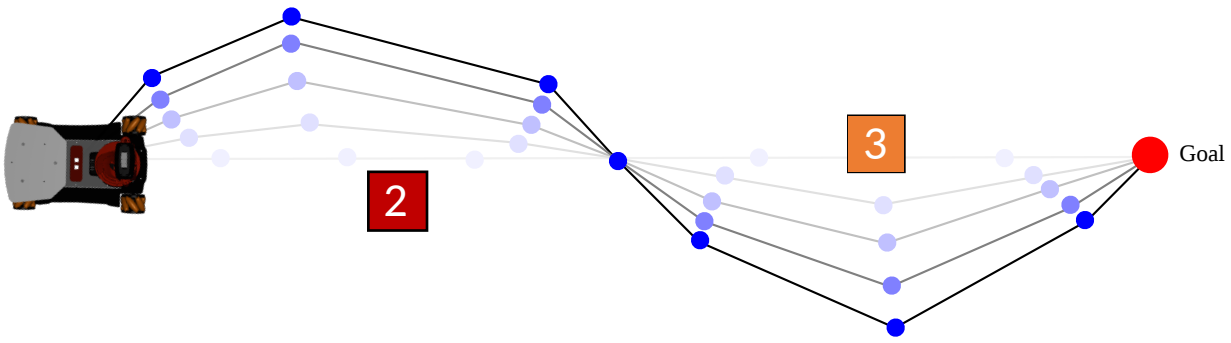


Figure 2: Gradient descent on the shape of a path using a set of waypoints and an objective function that trades off between minimal path length and maximal distance from obstacles.

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. This radius is the $r_i$ used in Equation 2.

**Implementation Instructions.** Create a service called `problem2a` of type `p4/FindPath` 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). Be sure to read the description file `srv/FindPath.srv` before starting your implementation to see what the inputs and outputs of your service need to be.

**Grading:**

For Problem 2a, your grade will be based on the difference in cost between the path that your service finds and the path that our own service finds. For full credit, your service will need to return a path in a reasonable amount of time.

If you want to visualize a path, publish a list of points of type `p4/PointArray` to the topic `/vrep/youbot/base/path`. You can visualize the path that is returned by your `FindPath` service, and you can also visualize the steps in the iteration. Only the most recent path so published will be drawn.

You should experiment with the number of waypoints for your paths. We have found three to be a good number, but try out different numbers to see how it changes the path that the robot takes.

### 2b: Waypoint Following (*20 points*)

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 5.3 of the course notes (Path-Following Control). The algorithm gets its name from the analogy that the robot is "pursuing" a moving point some distance in front 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.

**Implementation Instructions.** Design a service called `problem2b` of type `p4/FollowPath` that drives the youBot along a set of waypoints using the pure pursuit algorithm. Prevent the youBot from moving along arbitrary vectors by restricting your velocity commands to use a linear velocity in the $x$-axis (forward) and an angular velocity in the $z$-axis. Assuming that the goal will be the position of an object that the robot will interact with, you should make the robot stop just before reaching the goal. We have found a distance of 0.4m to be successful; try different distances near 0.4m to find the best threshold for your controller.

**Grading:**

For Problem 2b, we will grade your solution based on the average and maximum path-following error (cross-track, along-track, and heading errors). We will compare your solution's performance to our own algorithm's performance on these error scores, averaged over all paths followed.

Note that it is okay 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. For this reason, we have made the youBot base non-collidable with the blocks and bucket.

## 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 `problem3.py`. Then create a launch file called `problem3.launch` that launches your `problem3.py` script as well as the nodes for your services.

1. Optimally place the bucket with your `problem1a` service.
2. Plan a path to a block with your `problem2a` service.
3. Navigate to the block using your `problem2b` 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 `problem2a` service.
9. Navigate to the bucket using the `problem2b` 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. Publish to the arm joint angles so that the youBot's arm is pointing straight up in the air.
13. Repeat steps 2–12 until all of the blocks are in the bucket.

**Grading:**

We will grade this problem based on your implementation of the above algorithm, *not* on the performance of services you implemented in previous problems. Therefore, if you would like to use our implementation rather than your own for any the `problem1a`, `problem2a`, and `problem2b` services, you may do so. Our implementations can be found in `p4/bin`.

For full points, your implementation of the algorithm should make the robot navigate to each block and then to the bucket. It is okay if the youBot fails to grasp a block as long as it is reaching for the right point, and it is okay if the block misses the bucket when being deposited as long as the youBot was holding it above the bucket.

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.