Principles of Autonomy and Decision Making Final Project

Harvey Merton and Kevin Becker

Mechanical Engineering Department
Massachusetts Institute of Technology
Cambridge, MA
December 2022

1. Introduction

A wide array of approaches can be employed to solve sub-problems in mobile robotics and manipulation. In this project, we explore action planning, sample-based motion planning and trajectory optimization. These problems are solved in a simulated PyBullet kitchen environment (see Fig 1) to move boxes, with an arm mounted on a mobile base. The algorithms described to achieve this are custom-implemented in Python (accessible on GitHub).

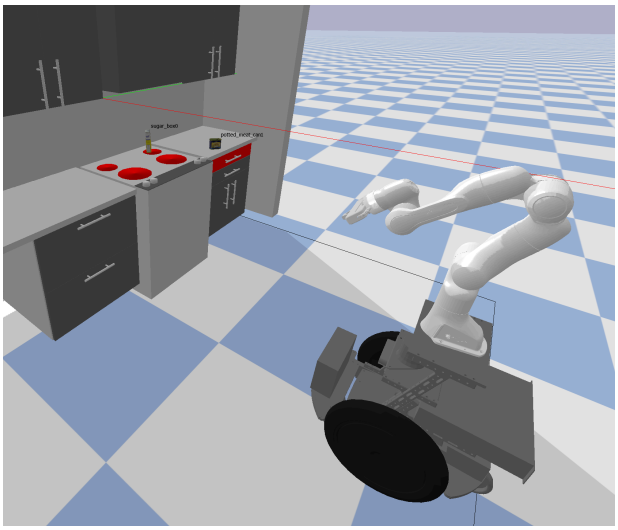

Figure 1: Kitchen environment

2. Overview of Key Files and Functions

Report section 3 - Activity Planning

- **part1_domain.pddl**
  - Defines the kitchen environment
- **part1_problem.pddl**
  - Defines the initial and final conditions.
- **part1_ff_planner.py**
  - FF_Planner class with a solve function that uses the fast-forward planner to create an action plan.
- **part1_bfs_planner.py**
  - BFS_Planner class with a solve function that uses a breadth-first search to create an action plan.

Report section 4 - Motion Planning

- **part2_rrt.py**
  - Contains the core rapidly exploring random tree (RRT) solver and various helper functions (e.g. detecting collisions and a random sampling function).

    - Also contains a TreeNode class to store a configuration (config) and parents in the RRT tree.
- **execute_plan.py**
    - Maps the action planner actions (part 1) to the motion planner (part 2) using 'action functions' e.g. navigate, close_drawer, pick_up etc.
    - Change the motion planning method (RRT or trajectory optimization) in line 285.

Report section 5 - Trajectory Optimization

- **part3_traj_opt.py**
    - Contains the Trajectory class that defines a desired start and goal config. This class contains the optimize function to set up the mathematical program and run the nonlinear optimizer.

Additional code

- **main.py**
    - The main function to run the action planner followed by the motion planner/trajectory optimizer to execute the plan.
- **utils.py**
    - Useful utility functions to interact with the world (e.g. hardcoded base movements, getting positions of items in the world etc.)

3. Part 1: Activity Planning

Defining the Problem

The problem is defined in accordance with the "Planning Domain Definition Language" (PDDL).

Assumptions

1. Robot gripper can only hold one item at a time.
2. Countertop, burner and drawer can each only hold a single item at a time.

Definition - domain

The domain has a few different **types** and *predicates:*

1. **Robots** can have one *item-in-grasp* at a time, and they can be near anything in the **environment** *(robot-near)*;
2. **Environments** have a location, and are either **items, item-holders,** or **item-holder-manipulators**;
3. **Item-holders** can hold items (*item-in-holder*);
4. **Item-holder-manipulator** makes it possible to "lock/unlock" an **item-holder** (see the *can-manipulate* predicate). An **item-holder** must not be closed (*item-holder-closed*) if

you want to grab an **item** from, or place an **item** on (holder must also not be full: *item-holder-full*), an **item-holder**;

5. *Can-manipulate* indicates if an **item-holder-manipulator** (drawer handle) can manipulate a given **item-holder**;
6. **Items** can be in a **robot's** hand (*item-in-grasp*) or in an **item-holder** *(item-in-holder)*.

Several actions are defined with these types and predicates:

- navigate moves a specified **robot**'s arm from a starting location to a goal location.
- open-drawer **and** close-drawer uses a given **robot** (must be currently at the **item-holder-manipulator)** to open/close the **item-holder.**
- pick-up puts an item in the **robot's** gripper.
- place removes an item from the **robot's** gripper and places it on the given **item-holder**.

Definition - problem

The robot, "franka" starts at the environment location "nowhere."
The "drawer" is initially closed, but it can be manipulated by the "handle."
The "sugar_box" starts on the "burner" and the "spam_box" starts on the "counter."

The goal state sees the "sugar_box" on the countertop, the "spam_box" in the "drawer", and the "drawer" closed.

Activity Planner

The main activity planner used is a slightly modified version of the fast forward (FF) planner. This planner is based on graph plan and enforced hill climbing (EHC) where an action is selected and taken if the state it leads to has a smaller fast-forward heuristic (h_FF) than is associated with the current state. A breadth-first search (BFS) planner is implemented to fall back on when an h_FF plateau is reached.

h_FF is calculated by counting the number of actions in the relaxed graph plan. As some actions have negative pre-conditions (e.g. "place"), a separate list of 'removed predicates' is maintained for generating a better relaxed plan (rather than simply removing 'delete' effects - this was a **challenge**). Actions are restricted to one at a time (one per layer) to remove the need for mutex checking.

Results

The plan generated by the planner is shown in Fig 2.

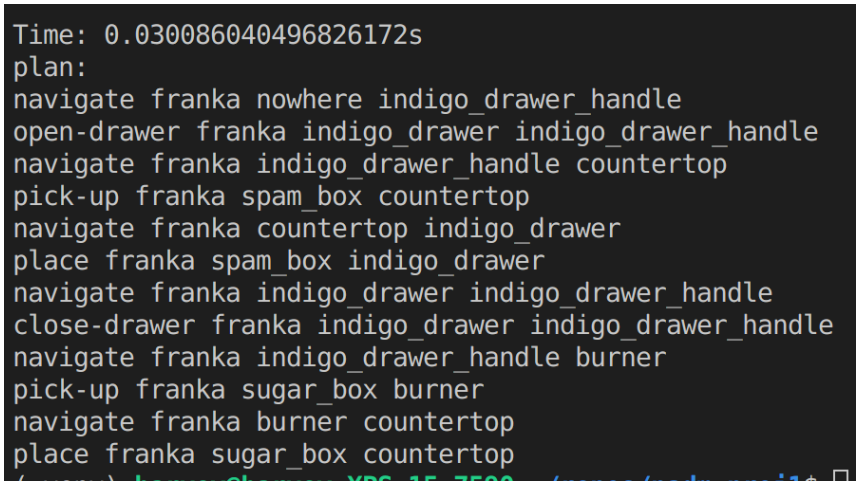

Fig 2: Action plan generated in 0.03s

4. Part 2: Motion Planning

Given the large, continuous configuration space, it was determined that a rapidly-exploring random tree (RRT) should be implemented for motion planning. We use inverse kinematics to find an appropriate goal configuration from a goal tool position, then use an RRT to find a suitable path to reach this end configuration.

Assumptions/simplifications

1. Each item can be picked up and placed with the gripper being flat.
2. The precision controls for grabbing and placing items were excluded in this project, so once the item is close enough to the robot it can “grab” the item which will pop the object into its gripper.
3. The world is completely stationary unless it is acted upon by the robot.
4. The robot base is in front of the kitchen counter and once there, remains stationary for the duration of the simulation.

Implementation/Method

Using a **sampling function**, RRT generates a random point within the configuration space (all possible joint settings across the arm’s seven joints) with a 20% chance of this point being the end configuration (**goal sampling**). A set of configurations linearly spaced between the new configuration, and the closest configuration already in the tree, are created (using **interpolate configs**). If the new node is the goal sampled node, the nearest node to the goal that hasn’t yet been added to the *tested nodes* set is used as the starting node (prevents double-checking the same node). If every configuration along the path is collision free (found using pyBullet’s ‘single collision’ **collision detection** function), the node is added to the tree. Once a node at the goal

has been successfully connected to the tree, a config path is returned by tracing each node to its parent.

Integration

To integrate the activity planner with the motion planner, a loop iterates through each action, passing the action name and all of its variables into the “perform_action” function. Then, each action as defined in the PDDL was given its own ‘action function’:

- **navigate**
  - Uses inverse kinematics to get the goal configuration from tool pose. The start and end configs are then fed into one of two path planners:
    - RRT is the default since it has collision checking.
    - The trajectory optimization function as described in section 5.
- **open-drawer/close-drawer**
  - Assumes the gripper is at the drawer handle position. It then moves the gripper +/- 0.5 m in the X direction, actuating the drawer with the gripper.
- **pick-up**
  - Moves an item’s pose to the robot’s gripper pose, adds the item to the ‘item in hand’ list and removes the item from the corresponding index in the ‘item in holder’ list.
- **place**
  - Removes an item from the robot’s arm (and the ‘item in hand’ list) and places it on the given surface (so adds to the ‘item in holder’ list).

Challenges

Goal sampling

- Added "nodes checked" list when goal sampling so a closer node to the goal won't be re-checked if the path from that node to the goal is already found to have collisions.

Base movement

- Base movement is "hardcoded" rather than using RRT. This allows no collision checking when the base is moved - movement function and goal position designed for the kitchen such that no collisions will occur and all objects will be reachable.

Visual artifacts

- Cloning and removing the body added notable computing time. This is fixed by using the actual robot itself for the forward kinematics, which can give a ‘glitchy’ appearance. RRT’s performance is not impacted however, so this is left in the program.

Results

RRT generates a feasible path with no collisions. The path is not close to optimal however (could be improved with RRT*). See video of results here: https://www.youtube.com/watch?v=zNdOW5OerSA.

5. Part 3: Trajectory Optimization

RRT, by its nature, generates abstract paths. An alternative approach to motion planning is trajectory optimization. pyDrake's nonlinear mathematical solver is used for this. The path chosen for optimization was the arm movement from empty space to the drawer handle, as no collisions occur along this path.

Problem Formulation

$$q \in \mathbb{R}^{N \times 7}$$

$$min_{q_0, q_1 .. q_N} \sum_{n=0}^{N-1} (q_{n+1} - q_n)^2 \quad \text{(o1)}$$

s.t.

$$q_0 = q_{start} \text{ (c1)}$$

$$|q_{N-1} - q_{goal}| <= q_{error} \text{ (c2)}$$

$$q_{n,x} <= q_{x,max} \, \forall x \in [0..6], \, n \in [0..N-1] \text{ (c3)}$$

$$-q_{n,x} <= -q_{x,min} \, \forall x \in [0..6], \, n \in [0..N-1] \text{ (c4)}$$

- Given a predetermined number of steps (N) in configuration space, discretize the path from start to end config in an Nx7 matrix (7 is the number of joints in the arm).
- Optimization function: minimize the sum of difference between adjacent configs squared (**o1**).

Constraints:

- The start (q_0) and end configurations (q_N) are set (**c1** and **c2** respectively), where the end configuration has an error margin for each joint (1% of the upper limits minus lower limits for each respective joint).
- The max and min configuration values are set for each of the 7 arm joints (**c3** and **c4** respectively).

Assumptions

A path where no collision avoidance is required was selected. This is because pyDrake is not compatible with our collision detection functions, so a large amount of rewriting code would be required to run collision detection within pyDrake.

Implementation/Method

pyDrake's 'mathematical program' class is used to implement the objective function and constraints defined above. This formulation is then passed to pyDrake's solve function that calls the snOPT nonlinear optimizer (as o1 is nonlinear).

Results
Using config interpolation (smoothed visuals) here.
Not using config interpolation here.

Comparison to sample-based motion plan
The optimized path appears almost the same as the RRT path for the movement from free space to the drawer. This is because the optimization function for trajectory optimization minimizes distance traveled in configuration space and so essentially draws a 'straight line' in configuration space from the initial to final config. As RRT starts with a goal sample, it determines that it can move directly from the start to the goal without collisions. The 'interpolate configs' function then also draws a straight line trajectory from the start to the final config, resulting in a trajectory that looks almost identical to that generated by trajectory optimization.

The differences in trajectories are slightly more obvious when looking at the 'non config interpolated' results from trajectory optimization as the points generated in this trajectory are slightly sparser. They are also extremely obvious when looking at other motion plans that involve collision avoidance as RRT moves to 'random' intermediate configurations but trajectory optimization still draws a straight line from start to goal, as no collision detection is implemented.

6. Conclusion

We learned about the implementation of a high-level action planner and how to integrate this with a motion planner. There are certain challenges that come with implementing these algorithms which go further than the basic algorithms themselves. For example, we had to identify how and when to convert from the tool pose space to the arm configuration space. Defining our goal-sampling functions, and trying to balance computing time vs fidelity are real world challenges, and learning how to deal with them now will serve us well in the future. One final issue we had was our screen recording software. It took too much computing power, and made the results jittery. So, we recorded the screen using a phone to properly demonstrate the performance. Both of us contributed in parallel on all parts of the project whilst bouncing ideas off of each other. This level of collaboration was enabled through our use of git.

All demo videos

"It's about the journey, not the destination"
-RRT, probably