Baxter Robot Block Picking and Placing

CS 5335 Robotic Science and System

Final Technical Report

Northeastern University, 2019 Spring

Prof. Robert Platt

Team:

Haojie Huang

Yiwei Gan

Ziyi Yang

Problem Statement:

In this project, the general goal of this team is utilizing Baxter robot and Gazebo GUI to perform a block pick and place while avoiding any obstacle. Previous Matlab projects has given us general view of the algorithms for path planning and robot kinematics in a relatively controlled simulation environment. This project will be performed in a scenario which is closer to reality.

Specifically, the robot need to pick multiple blocks at a known locations and drop them to a trash can at a known location. There would be a ball shape obstacle at a known location. The robot needs to avoid it during operation. This project was simulated on Gazebo GUI and all algorithms are written in a ROS package. Multiple algorithms include forward kinematics and two RRTs were developed for motion planning. Baxter motion control package were used for motion control.

Algorithm Description and Explanation:

Overall Algorithm Explanation:

The major parts of this algorithm are the PickandPlace Class a load_gazebo_model function, a delete_gazebo_model function, and the main function. The pnp (PickAndPlace) class generally include all necessary attributes and methods for this project:

- . __Init__: Collects all necessary variables for robot control and enable the robot
- .move_to_start: Moves the robot arm to initial position
- .ik_request is a service which input cartesian position and orientation in quaternion
- ._guarded_move_to_joint_position: Operates the robot's arms to reach commanded joints angles, and report back to user if no angles are entered which could mean the desired location is not available
- .gripper_open: Open the robot end effector gripper
- .gripper_close: Close the robot end effector gripper
- ._approach: Use .RRT to calculate a series of milestones which make sure the robot does not collide with given position and radius of spherical obstacles. Then, use ._guarded_move_to_joint_position to move the robot arm in sequence of these milestones and eventually make the end effector reach a location right above the desired location. The distance between the desired and actual end effector location is call hover_distance. This location is called the hover location.
- ._servo_to_pose: Operate the robot to reach the actual desired position
- ._retract: Use gik_request to calculate the position right above the desire location, then use ._guarded_move_to_joint_position function to move the robot to this location
- .pick: Open the gripper, approach the hover location right above the object, move the end effector to the object, close the gripper, and return to hover location
- .place: Move the arm to the hover location above the container, approach the container and open the gripper, then retract to the hover location

- .RRT: Rapidly-exploring random tree motion planning method used in this project to plan a route for the arm to follow without collide with obstacles

Beside the pnp class, load_gazebo_models function loads all materials needed for this project. That includes a ball shape obstacle, a table, a trash can, and 3 blocks with different colors (blue, red, and green). Then, delete_gazebo_models was used to remove all models.

In the main function, it initializes this ROS node. Then, it uses load_gazebo_models to load all required materials and tells the system to delete them when the system is shut down. The function will declares which arm to use, starting joints angles, and gripper orientation. Once all robot parameters is all set, the baxter robot would be told the locations and orientations of all target blocks and trash can. Eventually, it uses a for loop to pick and place all blocks into the trash can.

.RRT is the most important method developed during this project. It finds the best route to the target while making sure the end effector and the entire arm doesn't conflict with any obstacles. To achieve collision check, .RRT need 3 more functions, .fk_request, .robotcollision, and .checkedge.

- .fk_request: With given joints angles and joint number, calculates a 4x4 transfer matrix. Since we don't have access to the internal code of baxter.interface, we have to develop our own forward kinematics. The parameters for baxter such as link length and maximum and minimum joint angles are basically from Baxter Humanoid Robot Kinematics by Ohio University. The 7 joints of baxter has been simplified to 4 joints in terms of shoulder, elbow, wrist and gripper. (To transfer world frame to Gazebo frame we measure and imply a calibration value to make it precisely matched)
- .robotcollision: Use .fk_request to acquire position of each joint of baxter robot in certain joint angles, and then we can get the geometry pose of each robot arm with a 3x1 vector. After that, evenly sample 10 points on each arm and check all the points if they are in collision with the obstacle. Eventually, this function will return a bool value in which 1 means in collision and 0 means not.
- .checkedge: Use similar algorithm like what we done in .robotcollision: evenly sample 10 points between start angles and goal angles in configuration space, then use .robotcollision to check all the poses. This function return a bool value in which 1 means in collision and 0 means not.

The actual RRT algorithm utilized the 3 functions above to operate. There were 2 different RRT algorithms created with diverse performance characteristics.

Algorithm Comparison:

RRT_1:

1. Randomly sample 100 points in configuration space, and use .ik_request to check and delete all the point that the baxter robot is unable to reach or in collision with obstacle.
2. Create a point matrix which include the starting point as the first one, desired point as the last one, and the 100 sample points in between them. So that would make a 102x102 matrix.
3. Create a distance matrix. Each element in the matrix represents a distance between two points i.e. Matrix(i,j) is the distance between the i-th point and the j-th point
4. Within this matrix, we set the lower triangle matrix to be zero while the diangle will be zero itself. Then, the system will find the minimal value of each columns and set that value to 1 while using checkedge function to make sure the line is not colliding with obstacle. If there is a collision, there will spawn a new point which is at the midpoint, and the point being compared to would be deleted. Repeat this process until a non-colliding point is found. Then, the other points in this column are set to be 0. That way, this matrix would became a binary matrix, which contains the distance relationships between points.
5. Based on the distance relationship matrix, connect all points to the point which is closest to it.
6. Find a path from the starting point to desired point. If no path is found, go back to step 1 and redo the process until it converge to a clear path and return it.

RRT_2:

1. Set the starting point to be the first point and desired point to be the end point. Connect this two point and use checkedge to check if the path is in collision. If it is not colliding, return this path. If not, randomly sample a point within the configuration space.
2. Use robotcollision check if this point is inside the obstacle, if that is the case, delete the point and re-spawn another point until a free point is found.
3. Connect the two points, use chechedge to make sure if this line is not colliding with the obstacle. If that is the case, delete the latest point and go back to step 2.
4. Once a free point is created, connect it to the end point and chechedge, if the path is clear, the path is found. If it is not the case, spawn another point and go back to step 2. Repeat this process until a clear path to end point is found and return the clear path.

The two algorithm have very diverse characteristics, as RRT_1 intends to consume more time than RRT_2 does, but RRT_1 is able to explore the entire world thoroughly very rapidly. If we have a workspace with multiple complicated obstacles, RRT_1 should return a better results. However, in this project's operation condition as we only have one obstacle. The RRT_2 showed a better efficiency.

Simulation Results:

One of the most obvious difference between this two RRT algorithms is time consumed to find a valid route. In RRT_1, it takes average time for approximately 28:35 to complete the whole process (based on ten experiments). In RRT_2, it takes average time for 1:53 to complete the whole process. Both of these two algorithm have failure rate at 15%, it may be explained by the collision check algorithm we used, because we only check collision of robot arm by treating it as a straight line, but in reality the shape of arms also have certain radius. Both algorithms may lead the robot arm to a strange pose (see Fig.1 and Fig.2). This might be caused by random sample of points, and it could be solved by limit the configuration space of Baxter robot, but on the other hand it may also make it hard to find a valid path. Please also refer to a video attached to see the complete process of this operation.

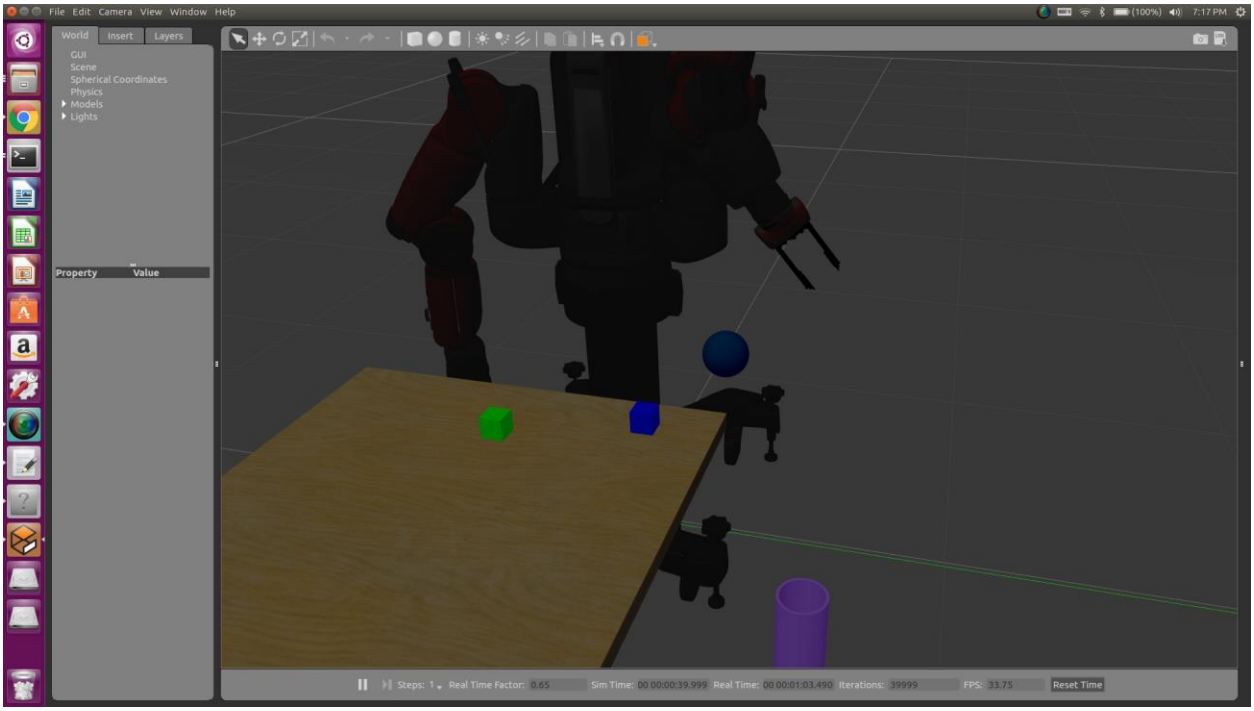

Figure 1 Baxter Trying to Avoid the Obstacle

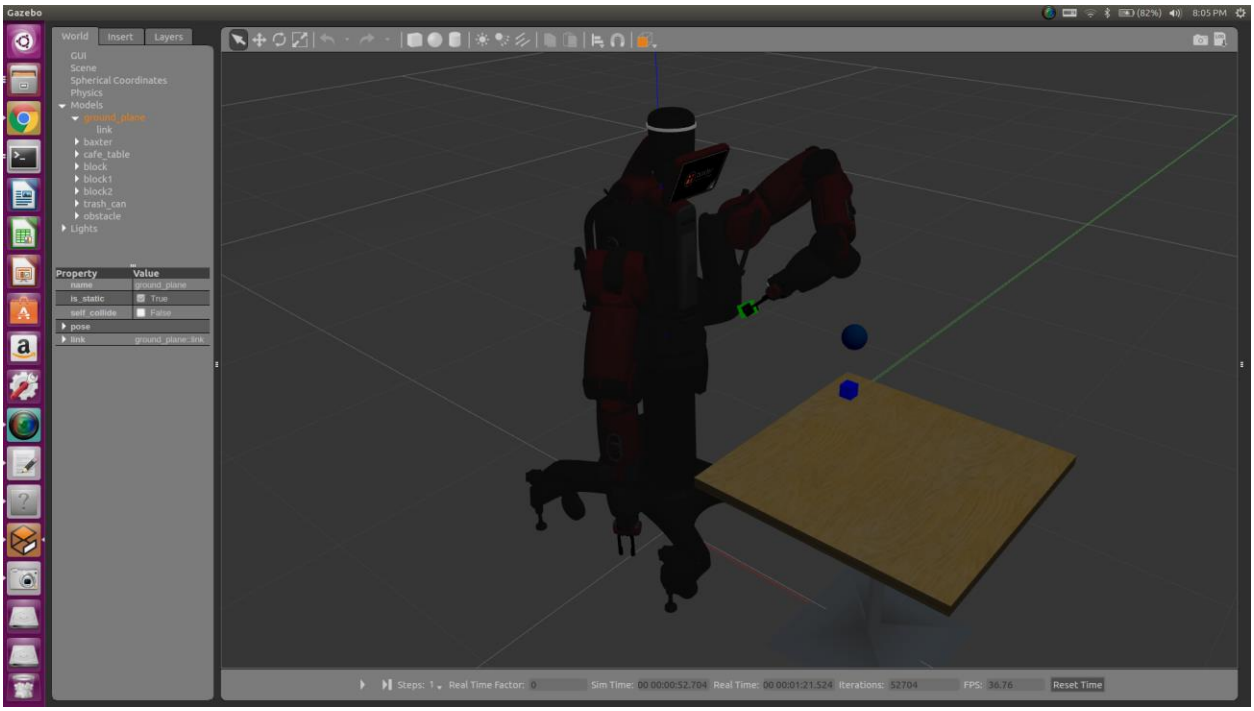

Figure 2 Baxter Getting into a wired position while trying to avoid the obstacle

Conclusion:

This project's goal is to simulate robot motion and planning in a more real environment. With multiple algorithms developed, the baxter robot was able to accurately pick a block with appropriate pose and orientation and then drop it to a wanted location while avoiding obstacle. There are 2 motion planning methods developed, even only one works well in this situation, the other one could also be a reasonable tool in motion planning like SLAM.

Unlike Matlab projects, we encountered more unexpected situations. For instance, the calibration of forward kinematics and unfamiliancy about ROS and Gazebo.

To improve this project, we are still working on the computer vision which could make the robot smarter. What's more, we can also use MoveIt! for a better path planning strategy.