Motion Module for the Amazon Picking Challenge 2016 - Team Delft

Mukunda Bharatheesha, Ruben Burger, Maarten De Vries,
Wilson Ko and Jethro Tan

August 22, 2016

The main responsibility of Team Delft's motion module for Amazon Picking Challenge (APC) 2016 is to ensure that our robot can move to all commanded locations to accomplish both the picking and stowing tasks in the challenge. The picking task consists of grasping an object of interest in the presence of other objects in the bin and placing the grasped object into a tote. The stowing task involves moving a certain object of interest in the tote to one of the bins in the shelf. The setup in reality is shown in Fig. 1.

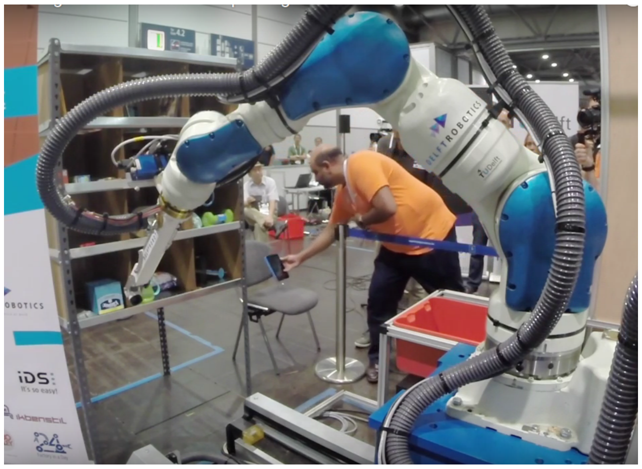

Figure 1: Team Delft Robot Setup for APC 2016.

The motion module is built on two fundamental motion primitives namely, *coarse motions* and *fine motions*. Coarse motions are essentially offline generated trajectories between pre-defined start and goal positions in the operational workspace of our system. On the other hand, fine motions involve online (cartesian) path planning for performing object manipulation in the bins or the tote. In the following sections, these two primitives will be explained in further detail.

A full technical article on our motion module along with graphical representations of our pipeline, collision avoidance with Octomaps and all details of the MoveIt APIs we used will be made available in the coming months. The text in this article is intended to provide a generic overview of our motion module.

1 Coarse Motions

The functional robot workspace for the APC is static. This forms the basis for the coarse motion primitive. In other words, there are no dynamic obstacles (at least in the challenge) that would obstruct the path of the robot after the robot has started moving. Thus, we define a coarse motion primitive as a trajectory that can be computed offline between a predefined start and goal location.

As a consequence, we implement a *trajectory cache*[1] which is populated with trajectories between around 250 different start and goal configurations for the robot. We call these configurations as *Master Pose Descriptors*. In principle, these are robot joint states set at appropriate values in front of each bin of the shelf. Our choice of having the camera mounted on the manipulation tool (see Fig. 1 entailed that we have two master pose descriptors per bin, namely, the camera master pose descriptor and bin master pose descriptor. Similar master pose descriptors are also defined for the tote drop-off locations. All trajectories that were used during APC 2016 were generated using RRT-Connect randomized path planner via MoveIt! The other planner option we tried was RRT-star which did not have any significant benefit over RRT-Connect in the given planning environment.

An example of a coarse motion trail from one of the bins to the *home* position of our setup is shown in Fig. 2.

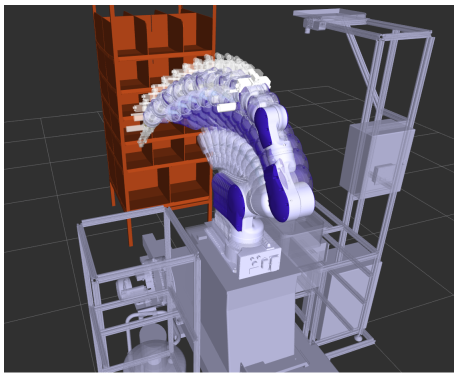

Figure 2: Coarse motion as a trail from "Bin D" to "Home".

[1]The use of *cache* is a misnomer because in the current implementation, we do not compute a coarse motion online, if a requested coarse motion does not exist in the cache. A trajectory database would have been a better name, in hindsight.

2 Fine Motions

Fine motions are the only part of the APC motion module that involve online (cartesian) path planning. This idea is a simplified implementation of the approach in the standard pick and place pipeline of MoveIt, where cartesian path planning is used during the pre-grasp approach and the post-grasp retreat stage of the pipeline.The simplification is the fact that we remove the evaluation of the reachable and valid pose filter stage of the standard pick and place pipeline. This process of finding and evaluating reachable and valid poses are done in a two-step filtering. The first filtering step is done in the grasp synthesizer module where impossible grasps are eliminated heuristically. The second filtering step consists of multiple `MoveGroup` API calls to `computeCartesianPath` after some key cartesian waypoints are evaluated for collisions using the `getPositionIK` service. Further details on this step will be more coherent to read once the following background information concerning our grasp strategy is explained.

It is important to highlight that, there is a serious limitation with our approach when we consider object manipulation inside a bin in general. This is because, we disallow any kind of collision with neighbouring objects inside a bin (or tote) and also end up in situations where no valid grasp candidates are found. However, allowing for *useful* collisions with other objects in a bin is certainly something we would consider in the future.

2.1 Grasp strategy

From a motion perspective, the grasp strategy for all objects in APC 2016 consisted of a combination of linear segments. We call these segments as *Approach*, *Contact*, *Lift* and *Retreat*. The segment names are indicative of the corresponding motions that those segments are meant for. Once the cartesian pose of the object of interest (for both pick and stow tasks), in a certain frame is estimated by the pose estimation algorithm, the grasp synthesizer uses this pose to generate a set of *key waypoints* for the start and end of each segment and in some cases more than just a pair of waypoints to limit any possibility of configuration changes while manipulating the object in the bin. These cartesian waypoints form an important input to the second step of grasp pose filtering and the fine motion generation.

2.2 Motion segment generation

The key waypoints corresponding to Approach, Contact, Lift and Retreat along with a potential grasp candidate are all input to the motion generation module where the following checks are conducted to generate the complete sequence of motions:

1. The grasp candidate, and the key waypoints are sequentially checked for collision using the `getPositionIK` service call. If any one of them is in collision, the corresponding grasp pose and the key waypoints are all discarded due to collision.

2. Once all the key waypoints are collision free, each linear motion segment is computed using the `MoveGroup` API, `computeCartesianPath` with collision checking enabled. Similar action is taken if any of these segments is in collision.

3. If all the linear motion segments are collision free, a final planning call is done using the `MoveGroup` API, `plan`[2]. This is required because, the final joint configuration at the end of the Retreat segment resulting from the call to `computeCartesianPath` need not necessarily match the starting joint configuration in front of the bin or tote from where the coarse motions start. This is natural because of the redundancy in the system. The call to `plan` is precisely to ensure that such a mismatch does not exist which would otherwise lead to a motion safety violation[3]

These steps ensure that all the desired motion segments for manipulating the object of interest are generated as required and are collision free. These segments are now *stitched* together before being executed on the robot. The stitching module is explained in the following section.

3 Motion stitching and execution

The motion stitching module accepts all the motion segments that are generated as explained in the previous section. Additionally, the coarse motion trajectories from the corresponding bin to the tote drop-off location (or vice-versa for stowing) are also input to the stitching module. The stitching process, in principle, is just the process of combining the joint state configurations from each segment into one single motion plan and time parameterizing it so that it results in a executable trajectory for the robot. We use the `computeTimeStamps` from the `TrajectoryProcessing` API for this purpose. This also allows us to use object specific velocity scaling (for instance, moving at low speeds when carrying heavy objects such as the dumbbell, socks and paper towels). An additional advantage of stitching multiple motion segments is that we can get rid of overheads such as goal tolerance checks at the end of each segment execution. This indeed provided us quite a significant time gain while executing the motions.

The final and a critical component of our motion module is the I/O handling to ensure the end effector (suction or pinch) is actuated at the right times along the trajectory.

3.1 Input-Output (I/O) handling

In the setting of the APC, it is critical to ensure the I/O is activated accurately and in a timely manner. For instance, the vacuum pump needs a couple of seconds before full suction power is realized. However, turning the suction on

[2]RRT-Connect is used here as the planner configuration due to fast solution times.

[3]A motion safety violation is triggered whenever the starting configuration of the robot does not match the starting configuration in the trajectory that is about to be executed.

too early can pose significant problems while approaching certain objects which have a very light outer covering. Basically, the outer cover would get suctioned in way too early before the approach is completed leading to either an unstable or a failed grasp. In order to address this, we built a custom trajectory tracking module based on the `/joint_states` topic.

The trajectory tracking module uses a gradient descent based approach to detect *events* along the trajectory based on distance to key joint state waypoints. These events and waypoints have a direct association to the key waypoints of the motion segments that we described earlier. As a matter of fact, these events are created exactly at the same point in the code where the key waypoints are created for the motion segments. The term "event" is used to emphasize that they are actual events along the trajectory such as beginning of approach segment, beginning of contact segment, end of retreat segment and so on. All these events also are used as feedback to evaluate the success or failure of a grasp. For instance, the vacuum sensor is read at the end of the retreat from the bin (or tote) to determine whether the grasp succeeded.

A commonly used alternative for I/O handling in MoveIt is the definition of I/O as joints with minimal displacement and providing them target joint values at appropriate times via regular planning calls. We do not use this approach as it does not guarantee adequate synchronization with the events that we define along the trajectory. Another note of caution is that the implementation of the trajectory tracking module was not completely straight-forward considering the limitation of only one `AsyncSpinner` per ROS Node.

Finally, the time parameterized trajectories are executed using the `MoveGroup` API, `execute`.