An Integrated System of 3D Pose Estimation and Primitive Robot Actions for Cluttered Manipulation - Motion Planning - Final Report

Samuel Chandler sxchandl@andrew.cmu.edu

Andrew Tracy atracy@andrew.cmu.edu

Juan Pablo Vega jvega@andrew.cmu.edu

Menghan Zhang menghanz@andrew.cmu.edu

Zihao Zhang zihaoz1@andrew.cmu.edu

1 Problem Definition

This project focused on designing and implementing part of a system for detecting and manipulating objects in a cluttered environment. The system, which was proposed by the Manipulation Lab, would use an ABB Robotics IRB 140 robot, a YCB object set, and multiple RGB-D cameras. Using the cameras, the system would perform 3D pose estimation for a subset of the YCB objects when the 3D CAD models and sample RGB images for those objects were provided. The objects would be placed in a bin/drawer without occlusions. After finishing the 3D pose estimation stage, the ABB 140 robot would plan and execute simple actions to move to a pregrasp pose for a selected object.

Our team focused on the second half of the project–motion planning. The team was mentored by Jiaji Zhou, Robbie Paolini, and Gilwoo Lee.

1.1 Robots and Objects

1.1.1 ABB Robotics IRB 140 robot

The IRB 140 robot is a small, fast, and powerful industrial robot arm manufactured by ABB. It has 6 DOFs. The robust design with fully integrated cables adds to the overall flexibility.

1.1.2 YCB Object Set

The Yale-CMU-Berkeley (YCB) Object and Model set has been widely used to facilitate benchmarking in robotic manipulation, prosthetic design, and rehabilitation research. The objects in the set are designed to cover a wide range of aspects of the manipulation problem; they include objects that would be encountered in an average person's daily life with different shapes, sizes, textures, weight, and rigidity.

1.2 Goals and deliverables

Given an object's ID from the YCB object set and its pose in the world reference frame, the motion planning module shall generate a plan to move to a pre-grasp pose for grasping a specific object. The correctness of the plan shall be demonstrated by execution both in the simulation and in the real-world environment. Planning and simulation will be done in the Aikido environment developed by the Personal Robotics Lab.

2 Related work

2.1 TSR

The Task Space Regions (TSRs) is a novel manipulation planning framework developed by the Personal Robotics Lab. It is designed to provide convenience and efficiency to sampling-based planning tasks in the presence of constraints, particularly constraints on end-effector pose of the robot.

The framework consists of three main components, namely the "constraint representation, constraint satisfaction strategies, and a general planning algorithm" [6]. A TSR is represented using the transform from the world frame to the object frame, the transform from the end-effector to the object (in the object frame), and a matrix of bounds B specifying the allowable motion of the end-effector along x, y, z and row, pitch, yaw axes (in the object frame).

We mainly took advantage of the direct sampling strategy presented in this framework for our project, where a pose is sampled from the bounds defined by the matrix B and passed to the Inverse Kinematics solver to generate a valid goal configuration for planning the robot.

2.2 HERB

HERB, the Home Exploring Robot Butler, is used to test for all of the algorithms in Personal Robotics Lab. He is a bimanual mobile manipulator comprised of two Barret WAM arms on a Segway base equipped with a suite of cameras as well as range senors. Most of the algorithms we have used from Aikido have been tested and implemented on Herb which gives us references and lots of help.

2.3 Aikido

Aikido is a C++ library (with Python bindings) developed by the Personal Robotics Lab for solving robotic motion planning and decision making problems. This library is tightly integrated with DART for kinematic/dynamics calculations and OMPL for motion planning. It can also integrate into ROS to get use of ROS communication and visualization tools.

2.4 OMPL

OMPL, the Open Motion Planning Library, is a library used for motion planning. It consists of many state-of-the-art sampling-based motion planning algorithms, such as PRM, RRT, EST, SBL, KPIECE, SyCLOP, and several variants of these planners. All planners in OMPL operate in state space. Most commonly used state spaces are already implemented and different state samplers can be used in each state space. OMPL itself does not contain any code related to collision checking or visualization. The library is designed so it can be easily integrated into systems like Aikido.

2.5 DART

DART (Dynamic Animation and Robotics Toolkit) is an open source library used for kinematic and dynamic applications in robotics and computer animation. It was created by the Georgia Tech Graphics Lab and Humanoid Robotics Lab. The library provides data structures and algorithms which have reliable accuracy and stability and the library is suitable for real time controllers. For developers, DART gives full access to internal kinematic and dynamic quantities, such as the mass matrix, transformation matrices and their derivatives. This allows DART to be easily integrated into Aikido for checking collisions and calculating kinematics and dynamics when planning.

3 Approach

We divided the system into different functional units. A diagram of the full architecture can be seen in figure 1.

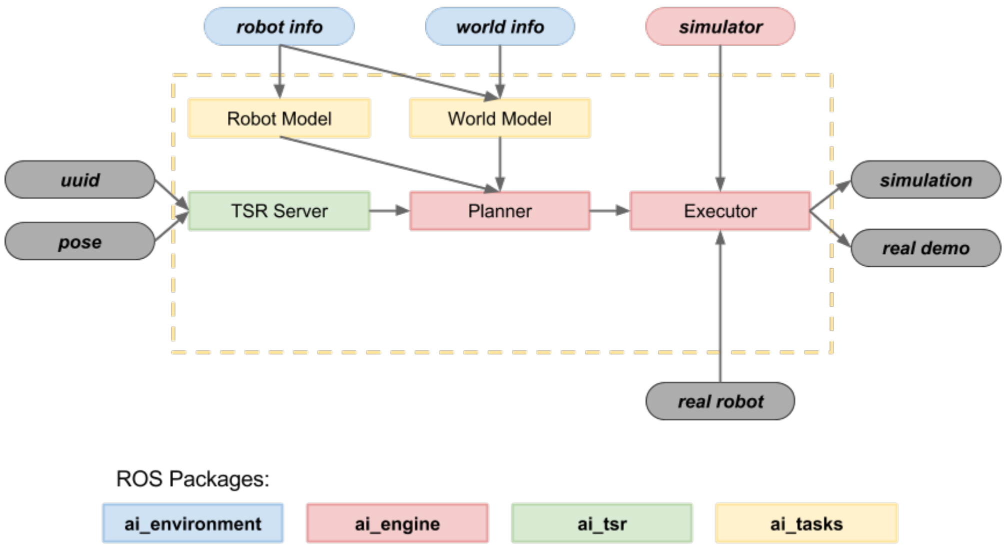


Figure 1: The system architecture.

The components of the system were created using Aikido, DART, and OMPL, and the code was written in a combination of Python and C++. Each major component was constructed in a ROS package, and the communication between components was handled with ROS. ROS was chosen for its relative ease of development and to promote modularity of the components.

The system that we implemented is based in large part on the pantry-loading demo provided by the Personal Robotics Lab. As such, there were some components that we did not have experience with at the beginning of the project. Over the course of the semester, as we gained experience with the tools, we were able to appreciate design decisions that we may have made differently had we started from scratch. However, due to the time limitations of the project, we largely borrowed from the existing work.

3.1 TSR

For our project, we proposed the novel idea of modeling each YCB object as a combination of geometric primitives. After reviewing and selecting a subset of the YCB objects we would like to include for our grasping tasks, we carefully chose the sphere, the cylinder, and the cuboid as the three geometric primitives we would use for modeling the objects.

Instead of assigning a TSR directly to each single object, we first started with defining the TSR of each of the three geometric primitives. We took the dimension and configuration of the gripper into account while modeling the TSR. We developed visualization tools in Python to help us better verify the correctness of each constructed TSR. In the visualization, we generated random samples, each shown as a coordinate frame, from the defined TSR. Each sample represents a possible pre-grasp pose for the gripper. The visualization results are shown in figure 2 below.

Next, we modeled the YCB objects in terms of the primitives. Specifically, we would like to identify the graspable region on each object and find one primitive or a combination of different primitives to best represent the geometry of the actual object. For example, we identified only the middle section of the banana object as the graspable region and modeled the region as a cylinder. Therefore, the pre-defined TSR for the cylinder could be transferred and applied on the banana objects, reducing any repeated work on assigning a new TSR for similar graspable regions. Most often, however, only a subset of the pre-defined TSR can be directly applied to the actual object because of the additional geometric constraints on the object (for example, we cannot grasp from either the top or bottom of the cylinder representing the graspable region of the banana because they are connected to other solid bodies). We also incorporate these constraints while assigning TSRs for the objects. We selected nine objects from the YCB model set, along with a sphere object representing an orange, and have created the TSR for each of them.

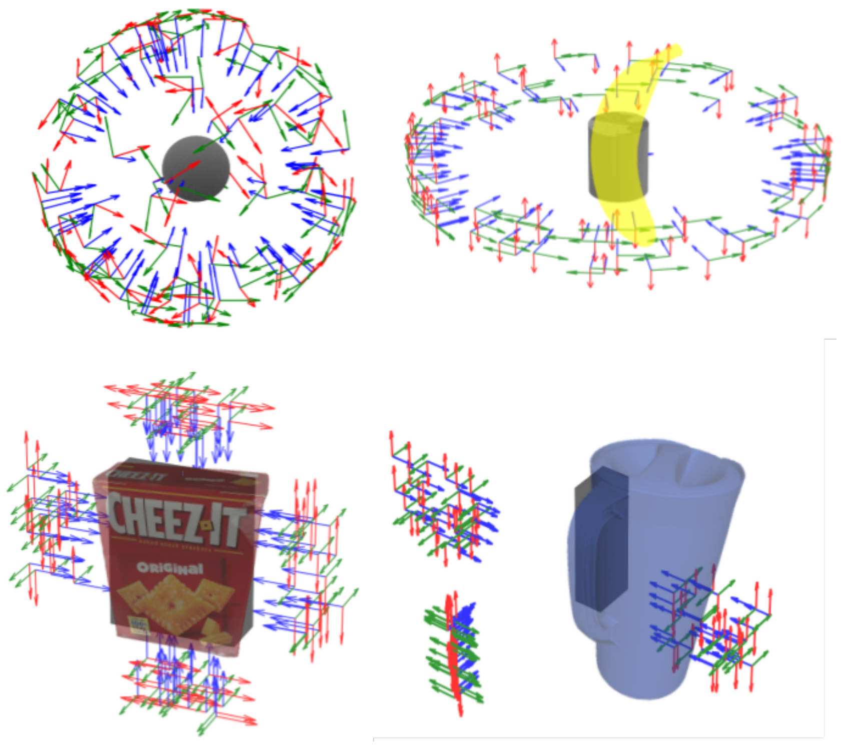


Figure 2: Primitives and objects with assigned Task Space Regions (TSRs).

We created a ROS package that contains our modeled TSRs and works as a standard ROS service. It has a config folder which contains several blueprints that define the primitives for each object and the TSRs for each primitive. The package first receives the object UUID (0 to 18) and pose as x,y,z,roll,pitch,yaw. Then it queries the primitives and TSRs based on the UUID from the blueprint. After that it composes the relative pose of the primitive with the input pose to put the TSRs in the world frame so that we can get the pose of each TSR relative to our robot. Finally, it returns a list of valid TSRs (given the geometric constraints of the gripper).

On the client side, we parse the TSR message into a vector of TSR objects for further usage in the planner.

3.2 Environment

The environment and robot modeling used URDF models. Some of the models were provided by the robot manufacturers, some were provided by the Manipulation Lab, and some were created by the team. For the robotic arm, we manually combined the ABB and Robotiq URDFs into a single URDF.

We were provided a model of the table from the Manipulation Lab. We had to model the bin separately and add it to the table model; we did this in Solidworks. We encountered an issue with the STL file format, which took some time to debug. In the end, we determined that the format for some of the provided files was too old and had to be re-exported using a different 3D modeling program like Sketchup.

Even though we used the existing YCB object dataset, we still had to spend time modeling the objects so that they could be used with the TSR generator, as described in the TSR section.

We wrote separate code for modeling the robotic arm in an object-oriented fashion, mimicking the way that HERB is modeled in the libherb package from the Personal Robotics Lab. The code allowed the arm object to return information about its state and geometry, and exposed the ability to set the robot joint positions. However, unlike HERB, the robot did not include any planning abilities; for that we built a separate package (see below).

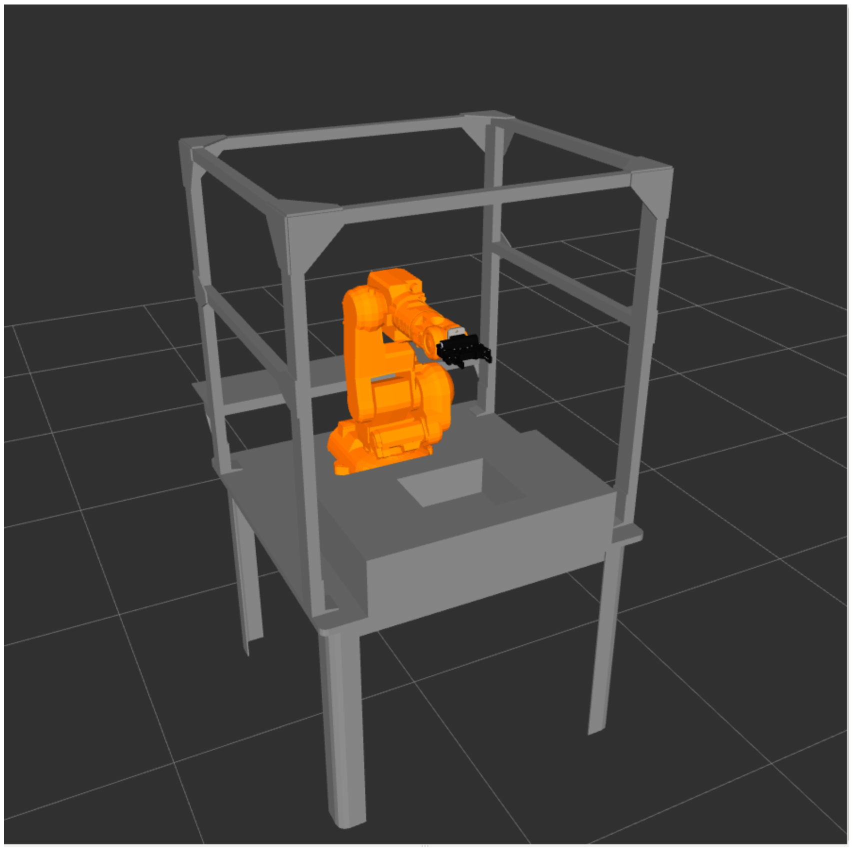


Figure 3: The ABB robot in the environment.

3.3 Engine

The planning part of the engine package takes a model of the robot and its environment as an input and outputs a plan to one of the sampled TSRs. A planning object is created with the robot and environment, which can then plan to one or more TSRs with the planToTSR or planToMultiTSR functions. If either of these methods are called, the planner will first try to linearly plan to a TSR (a snap planner). Then, if this fails, it will start using an RRT planner from the OMPL library.

Once a plan is created, the executor parses the untimed trajectory into a series of waypoints in joint space. We provided two separate executors: one for sending messages to the simulator in rviz and one for sending messages to the real robot. For executing on the real robot, we used a set of services that were created by the Manipulation Lab, which made executing the trajectory on the real arm very simple.

3.4 Tasks

In order to input task-specific information and run the system using our architecture, we wrote an executable script. The script takes a number of command line arguments:

- Object UUID
- Object pose (x, y, z, roll, pitch, yaw)

Given the user input, the script loaded the object, robot, and environment models; called the TSR server and planner; and sent the generated trajectory to the executor. Because we were using Aikido and DART for our physics simulation framework, all the simulation-related models were loaded in the script.

The script borrowed heavily from the pantry-loading demo, but was adapted to use our environment, TSR generator, and planner.

4 Results

4.1 Implementation Results

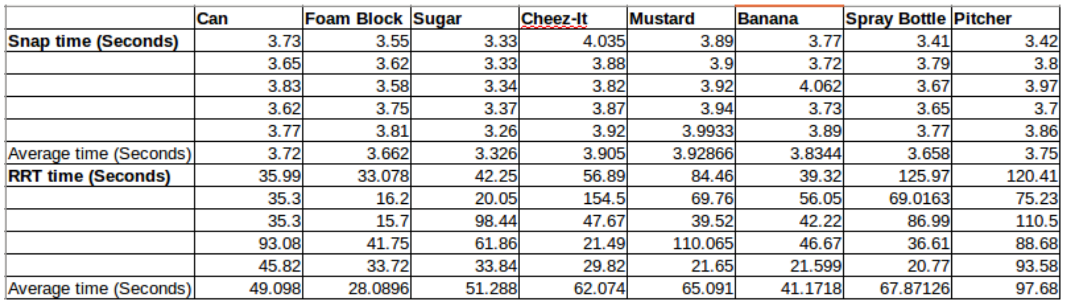
After developing the planning package, TSR package, environment package, and task package we were successfully able to plan a trajectory and execute it on the real robot as well as in simulation. The videos linked below show the snap planner being executed on the mustard bottle and the RRT planner being executed on the banana. Here the snap planner is very quick when coupled with the sampled TSRs and has an average planning time of 3.5-4 seconds if a grasp is feasible. The RRT planner, on the other hand, takes 20-100 seconds on average, and has an extremely high variance, see table 1.

4.2 Videos

Videos showing the planner in the simulated environment and trajectory execution in simulation and on the real robot can be seen here:

RRT simulated execution with banana: `https://youtu.be/MDsQfe2N1l8`
RRT real robot execution with banana: `https://youtu.be/ZjizzjHcy50`
Snap simulated execution with mustard: `https://youtu.be/CyIeV80lRBo`
Snap real robot execution with mustard: `https://youtu.be/mqKcMxCoOuw`

Table 1: Planning time results for all objects located in the the center of the bin and with a TSR in the z direction.



	Can	Foam Block	Sugar	Cheez-It	Mustard	Banana	Spray Bottle	Pitcher
Snap time (Seconds)	3.73	3.55	3.33	4.035	3.89	3.77	3.41	3.42
	3.65	3.62	3.33	3.88	3.9	3.72	3.79	3.8
	3.83	3.58	3.34	3.82	3.92	4.062	3.67	3.97
	3.62	3.75	3.37	3.87	3.94	3.73	3.65	3.7
	3.77	3.81	3.26	3.92	3.9933	3.89	3.77	3.86
Average time (Seconds)	3.72	3.662	3.326	3.905	3.92866	3.8344	3.658	3.75
RRT time (Seconds)	35.99	33.078	42.25	56.89	84.46	39.32	125.97	120.41
	35.3	16.2	20.05	154.5	69.76	56.05	69.0163	75.23
	35.3	15.7	98.44	47.67	39.52	42.22	86.99	110.5
	93.08	41.75	61.86	21.49	110.065	46.67	36.61	88.68
	45.82	33.72	33.84	29.82	21.65	21.599	20.77	93.58
Average time (Seconds)	49.098	28.0896	51.288	62.074	65.091	41.1718	67.87126	97.68

4.3 Future Improvements

4.3.1 TSR Improvements

We modeled some TSRs for object primitives, but we did not model all of the TSRs for a given object. For this we would need to incorporate several different geometric shapes to come up with more TSRs.

4.3.2 Software Architecture Improvements

While we tried to keep the architecture as modular as possible there are still some aspects that could be improved. For instance, the tasks package does some parsing of the TSRs and the trajectory, which could be moved into their respective packages.

Also, for the purposes of this project, the script used hard coded values that were specific to our setup. It is foreseeable that in future work, this could be abstracted somewhat to provide an easier experience for the user, so that they only had to specify the robot, environment, and objects. In that framework, the user would not have to worry about the implementation details of modeling objects in Aikido and Dart and could focus on the particular task they are interested in completing.

4.3.3 Planning Improvements

With our current configuration we use an RRT planner as the backup to our snap planner. The results we obtained with the RRT planner show that it is extremely slow with high variance. This is expected given that we are planning in high dimensional space. This planner should be replaced with something more deterministic and slightly faster like A*.

5 Division of Work

Work was divided according to Table 2.

Table 2: Final division of work among team members

Task	**Team Members**
Software environment setup	All
Architecture design	All
Implement environment and robots	Andrew, Zihao
Implement planning wrapper	Samuel
Implement execution wrappers (simulation and real demo)	Juan Pablo
Implement *TSR* run-time server	Juan Pablo, Menghan
Determine pre-grasp pose using *TSR*	Zihao, Menghan
Integrate software pipeline	Samuel, Andrew, Juan Pablo
Simulation tests	Samuel, Andrew, Juan Pablo
Planning time tests	Samuel
Integrate with physical robots	Samuel, Andrew, Juan Pablo
Real-life tests	Samuel, Andrew, Juan Pablo
Performance evaluation and project closure	All

6 References

1. Aikido
   `https://github.com/personalrobotics/aikido`
2. DART
   `http://dartsim.github.io/`
3. OMPL
   `http://ompl.kavrakilab.org/`
4. URDF files for ABB 140 arm
   `https://github.com/FreddyMartinez/abb_irb140_support`
5. URDF files for Robotiq C2 85mm gripper
   `https://github.com/ros-industrial/robotiq/tree/indigo-devel/robotiq_c2_model_visualization`
6. TSRs
   `http://www.ri.cmu.edu/pub_files/2011/10/dmitry_ijrr10-1.pdf`
7. YCB objects
   `https://arxiv.org/abs/1502.03143`
8. Project Code
   `https://bitbucket.org/mrsdteamai/`