Faculty of Engineering Science
Department of Mechanical Engineering
Celestijnenlaan 300 box 2420 B-3001 Heverlee

Robot arm planning

Date	24/05/2017
Course	H04P9A H00S1A
Type	Final report
Access	Public
Authors	Xiyu Fu Wouter Van Grimbergen
Commissioned by	Intermodalics
Commissioning supervisors	Nick Vanthienen
Supervisors KU Leuven	Herman Bruyninckx Jan Swevers

Abstract

Most robot arms can only track external predefined trajectory. This blindness makes operating in a shared working space impossible and redeploying a robot is difficult. A motion planner can greatly increase the flexibility of robots. There are many motion planning algorithms. Each has its own advantages and disadvantages. The question is which one to use in a dynamic environment. The goal of this project is to test and compare different motion planners in different planning scenarios.

For calculations of the motion planner, inverse kinematics and collision detection, MoveIt! is used in this report. It is a well known ROS node for motion planning. All the generated trajectories in MoveIt! are then visualized in Rviz. In this project, a UR10 from Universal Robots is used for the robot arm.

The first goal of this project was comparing two different inverse kinematic solvers, TRAC-IK and KDL. These inverse kinematic solvers are important because most planners work in the configuration space and the planning request is in the Cartesian space. So, calculation of inverse kinematic is needed. The planning time is dependent of the calculation time of the inverse kinematic solvers. To test the success rate and the calculation time of the inverse kinematic solvers, 1000 random reachable poses are generated and the joint values are calculated. From the test, it was clear that TRAC-IK is faster and has a higher success rate then KDL.

To compare the motion planners, three scenarios are defined. The first scenario is a straight motion that must be followed from one point to another point. The second scenario is a ping pong scenario, which is a table with in the middle a wall, where the ball needs to go from one side of the wall to the other side. The last scenario is a ball, which needed to be picked from a shelf with five closet shelves.

In this report, all the sample based motion planners, included in the OMPL library, were tested. Also, some tests are done with CHOMP motion planner and SBPL. For the straight motion scenario, some tests are done with the Cartesian interpolation in MoveIt!. The metrics, for comparing the different planners in all the scenarios, are the success rate, planning time, path smoothness and path length.

Out of the results we conclude that for fast changing environments RRTConnect is the best choice. If the environment is changing slowly and without narrow free space, it is better to choose PRMstar. Also, it was clear from the results that Cartesian space interpolation is never recommended and the CHOMP planner is not better than sample based planners. For the SBPL, further work is needed to fully implement it in MoveIt!.

In this report, we also include a recommendation about parameter tuning for the motion planners. Firstly, that the maximum planning time doesn't need to be taken too large, since the most successful planning attempts finish in a short time. Also, that the tolerance of the planners depends on task requirement, but it must be larger than the tolerance of inverse kinematic solver. The last recommendation is about the longest valid segment fraction for algorithms that using discrete collision detection.

Table of content

List of Figures

List of tables

1. Introduction

Robot arms are widely used in industrial environments. They have greatly improved the productivity and released workers from repeated and boring tasks. However, most of these robots can only track an externally defined trajectory which is aimed at a specific task. The robot have no knowledge about surrounding environment let alone correctly react to the environment. This blindness makes it a costly job to redeploy a robot in another working scene. And working in a dynamic environment, for example working in a shared working space with human, is also impossible.

To cure this blindness, many problems have to be solved. One important problem is the motion planning problem. In general, to plan a motion means to divide a desired motion into a sequence of simple motion primitives which could be performed by low level controllers. In the robot arm case, it means to find a sequence of collision free joint value states from the initial state to the goal state that could be followed by the robot arm. There are many motion planning algorithms such as search-based algorithms, geometric algorithms, artificial potential fields and sample-based algorithms. The sample-based algorithm is widely used .

Because the low level controller in a robot arm can only control the state of motors, we have to find a map that links joint space and working space (3D Cartesian space). This is the kinematic problem. We call the map from joint space to working space as the forward kinematic and the map from working space to the joint space as inverse kinematic problem.

Since there are many motion planning algorithms and inverse kinematic solvers available, which one should we choose to help our robot arm become smarter and more flexible? We will try to give some suggestions about this problem.

In this project, we are going to run simulations with different motion planners in different planning scene. Our robot model is a UR10 robot. It is a robot arm with 6 rotational joints. These simulations are running in ROS environments. We will use Moveit for planning and Rviz for visualizing the results. Then we will compare the results between different planners and try to give some suggestions on planner selection.

The rest of this report is organised as follows: In the second part, we will give a short introduction to the simulation environment - ROS, Moveit and Rviz. In the third part, we will compare two different inverse kinematic solvers namely KDL and trac_ik. In the fourth part, we will test and compare different planners. And the last part is the conclusion.

2. Simulation Environment

2.1. ROS

Robot Operating System (ROS) is a meta-operating system for robots which is open-source. It provides services that would be expected from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message passing between processes, and package management. It also provides tools and libraries for obtaining, writing and running code across multiple computers. [12]

The philosophy of ROS is to make a piece of software that could work with other robots by making little changes in the code. This makes sharing of functionalities, that are created or used, between different robots easy. [13]

The Computation Graph is the peer-to-peer network of ROS processes that are processing data together. The Computation Graph concept consist of nodes, messages, topics, services, Master, Parameter Server, all of which provide data to the Graph in different way. [14]

Nodes are processes that perform computation. A robot control system usually comprises many nodes. For example, one node controls a laser range-finder, one node performs localization, one node performs path planning, and so on. Nodes communicate with each other by passing messages. Those messages are routed via a transport system with publish/subscribe semantics. A node sends out a message by publishing it to a given Topic, this is shown in figure 1. The topic is a name that is used to identify the content of the message. A node that is interested in a certain kind of data will subscribe to the appropriate topic. There may be multiple concurrent publishers and subscribers for a single topic, and a single node may publish and/or subscribe to multiple topics. Request or reply, of messages on a topic is done via services. [14]

The ROS Master provides name registration and lookup to the rest of the Computation Graph. Without the Master, nodes would not be able to find each other or exchange messages. A part of the master allows data to be stored by key in a central location. This data storage is done by the Parameter server. [14]

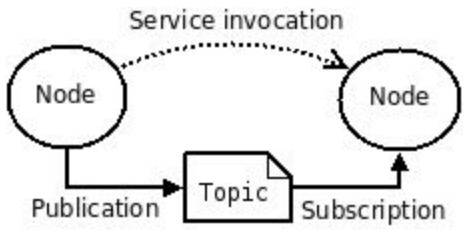

Figure 1: Schematic of how nodes communicate between each other by topics. [14]

2.2. MoveIt!

MoveIt! is state of the art software for mobile manipulation, incorporating the latest advances in motion planning, manipulation, 3D perception, kinematics, control and navigation. It provides an easy-to-use platform for developing advanced robotics applications. MoveIt! is the most widely used open-source software for manipulation. MoveIt supports more than 65 robots, such as the Universal Robots UR10 that is used in this report. [15]

The primary component of MoveIt! is the move_group node. This node serves as an integrator: pulling all the individual components together to provide a set of ROS actions and services for users to use. Figure 2 gives a schematic overview, of the system architecture, for the primary node move_group. [16]

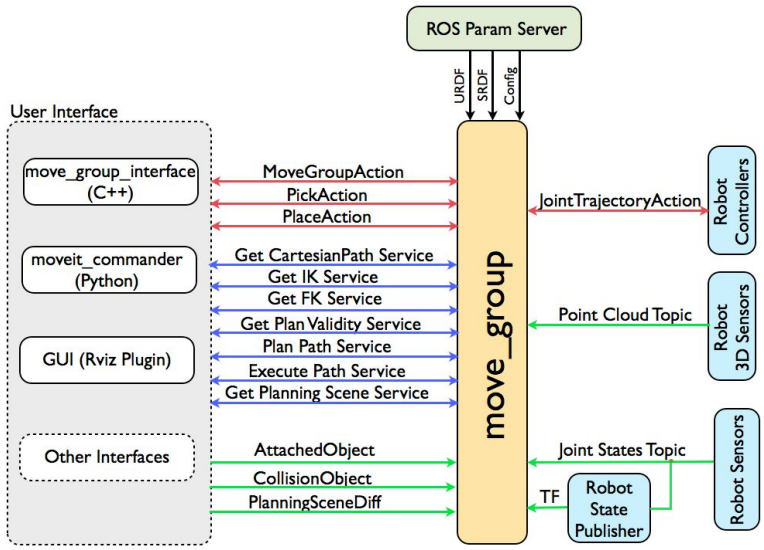

Figure 2: High-level system architecture for the primary node provided by MoveIt! called move_group. [16]

The move_group ROS node that uses the ROS parameter server (Param Server) to get information about the robot. There are three kinds of information that it gets from the param server. First, it gets the Universal Robotic Description Format (URDF) that describes all elements of a robot. the second information that the move_group node get from the param server is the Semantic Robot Description Format (SRDF). The SRDF represent the information about the robot that is not included in the URDF. The third and last information, that is get from the param server, is the MoveIt! configuration. This include joint limits, kinematics, motion planning, perception and other information about the robot. [16] [17] [18]

Move_group talks to the robot through ROS topics. It communicates with the robot to get current state information (position of the joints, etc.), sensor data from the robot and to talk to the controllers on the robot. To have a representation of the world and the current state of the robot, move_group uses the Planning Scene Monitor. [16]

MoveIt! works with motion planners through a plugin interface. This allows MoevIt! to communicate with and use different motion planners from multiple libraries, making MoveIt! easily extensible. [16]

When a motion plan is requested, like move robot arm to a different location or the end-effector to a new pose, the move_group will generate a desired trajectory. This trajectory will move the arm, or any group of joints, to the desired location. Figure 3, gives an overview of the complete motion planning pipeline chain. The planning request adapters allow for pre-processing motion plan requests and post-processing motion plan responses. Pre-processing is useful in several situations, e.g. when start state for robot is slightly outside the specified joint limits for the robot. Post-processing is needed for several other operations, e.g. to convert paths generated for a robot into time-parameterized trajectories, motion plans with velocity and acceleration constraints. [16]

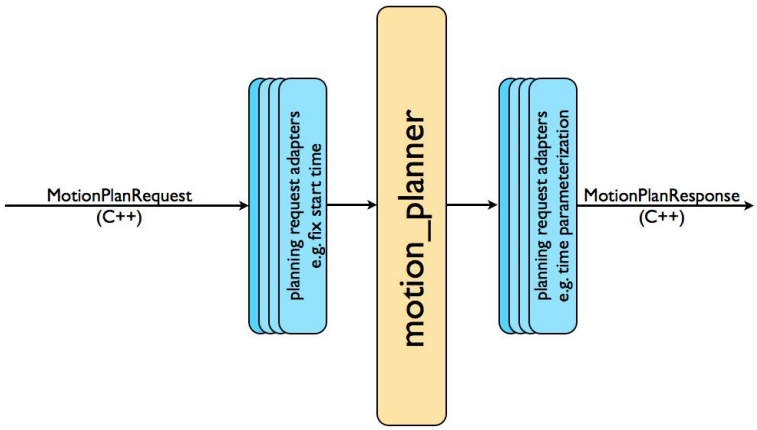

Figure 3: Schematic overview of the planning pipeline chain. [16]

2.3. Rviz

ROS visualization, Rviz, is a 3D visualizer for displaying sensor data and state information from ROS. Using Rviz, current configuration of the UR10 can be visualized on a virtual model of the robot. It is also possible to plot trajectories in Rviz, coming from a ROS topic, that are planned with MoveIt!. Data from sensors and other data can be visualized with Rviz. [20]

3. Inverse Kinematic Solvers

Inverse kinematics (IK) determine the joint parameters (joint velocity, joint position) calculated from a given Cartesian pose of the robot end effector. When a motion plan is generated, the inverse kinematics transforms this plan into joint actuator trajectories, for the robot.

There are both an analytical and a numerical solutions for the inverse kinematics. Analytical solutions suffer from an inability to generalize to tool-use scenarios or changes in robot configuration, as the solver must be constructed beforehand. Typically numerical IK solvers are used. [1]

In this project, two different methods of numerical inverse kinematic solvers are used. These two inverse kinematic solvers use different methods to solve the joint parameters.

3.1. KDL

Kinematics and Dynamics Library (KDL), library from Orocos, is today the most commonly used numerical IK solver in the robotics community. KDL is the standard inverse kinematic solver in the MoveIt planning library.

KDL is an inverse Jacobian solver, that uses the Newton method. Given a seed value for joints q_{seed} (often the current joint values), forward kinematics can be used to compute 1) the Cartesian pose for the seed, 2) the Cartesian error vector p_{err} between the seed pose and the target pose, and 3) the Jacobian J, which defines the partial derivatives in Cartesian space with respect to the current joint values. After inverting the Jacobian, J^{-1} defines the partial derivatives in joint space with respect to the Cartesian space. Consequently, an Inverse Kinematics solution is simply computed by iterating the function: [1]

$$q_{next} = q_{prev} + J^{-1} p_{err} \qquad [1]$$

Where Forward Kinematics of q_{next} is used to compute the new value p_{err} . When all elements of p_{err} fall below a stopping criteria, the current joint vector q is the returned IK solution. [1]

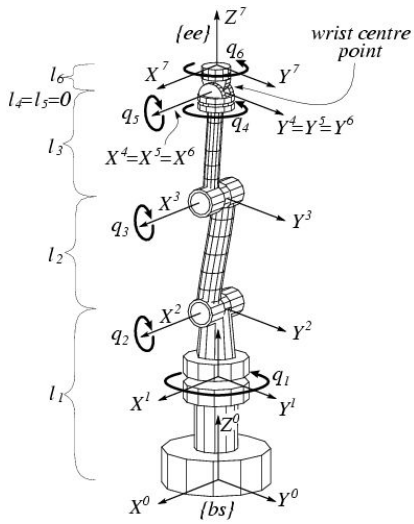

Figure 4: Kinematic chain [20]

This inverse Jacobian IK method can't handle very well physical joint limits. Then the inverse Jacobian IK can get stuck in local minima during the iteration process. Because of the non-smooth search space that are created by the joint limits. If this, get stuck in local minima, occurs the correct IK solution can't be found and the solution failed. [1]

Another disadvantage of this method is the computational time to solve the inverse kinematics. In general, the time to compute and utilize the Jacobian will depend on the size and complexity of the manipulation chain. [1]

3.2. TRAC-IK

To improve the performance of an IK solver, to keep computation time low and to reduce failures, the TRAC-IK algorithm was introduced by traclabs. For a single IK solver request, TRAC-IK spawns two solvers, one running SQP-SS and one running KDL-RR. Once either finishes with a solution, both threads are immediately stopped, and the resulting solution is returned. [1]

3.2.1. KDL with random restarts: KDL-RR

One of the biggest disadvantages of the KDL algorithm is that it can get stuck in a local minima. There is nothing provided, in the KDL algorithm, to detect a local minima. However, local minima are easily detectable, when q_{next} - $q_{prev} \approx 0$ there is a local minima. In an improved implementation of the pseudoinverse Jacobian IK, local minima are detected and mitigated by changing the next seed. Consequently, the performance of the KDL IK solver can be significantly improved by simply using random seeds for q to "unstick" the iterative algorithm when local minima are detected. This implementation of KDL, with random restarts, is referred as KDL-RR. [1]

Even with adding random restarts, with the KDL IK solver, the failure rate for many kinematic robot chains is still to high. This, failure rate, is due to joint limits. [1]

3.2.2. Sequential quadratic programming with Sum of Squares: SQP-SS

If robot joints have hard limits, the failure rate of KDL IK solvers are still too high, because of the KDL solver can get stuck in a local minimum. To get a lower failure rate, another IK solving method has to be used. One solution, to avoid failure because of joint limits, is using sequential quadratic programming (SQP). This method is better in handling constraints, introduced by the joint limits. The SQP is an iterative method to solve nonlinear optimization problems. [1]

The SQP formulation is characterized as follows [1]:

$$\begin{aligned} \arg\min_{q \in \mathbf{R}^n} \quad & (q_{seed} - q)^T (q_{seed} - q), \\ s.t. \quad & f_i(q) \leq b_i, \ i = 1, ..., m, \end{aligned} \qquad [1]$$

where q_{seed} is the n-dimensional seed value of the joint, and the inequality constraints $f_i(q)$ are the joint limits, the Euclidean distance error, and the angular distance error. [1]

SQP-SS is the top overall algorithm in terms of IK solve rate. However SQP-SS can have a much longer solve time than the pseudo-inverse Jacobian methods for certain robotic configurations. [1]

3.3. Comparison KDL and TRAC-IK

The 2 inverse kinematic solvers, KDL and TRAC-IK, have some advantages and disadvantages. Those are listed below. This is found in the paper of P. Beeson and B. Ames "*TRAC-IK: An Open-Source Library for Improved Solving of Generic Inverse Kinematics"* [1].

- TRAC-IK is more reliable and faster in computational time then KDL
- KDL works well for unbounded joints and long kinematic chains since redundancy avoids local minima caused by joint limits. But TRAC-IK still overclasses KDL in this situations.
- With TRAC-IK there is a smoother trajectory, in other words, there are fewer suddenly jump. And KDL tends to give a solution that suddenly jump from one pose to another.
- Neither KDL nor TRAC-IK can handle singularities very well. No solution will be found, with both IK solver, near the singularity. (For example: position=(0, 0.5, 1.0), orientation=(0.707, 0, 0.707, 0)).

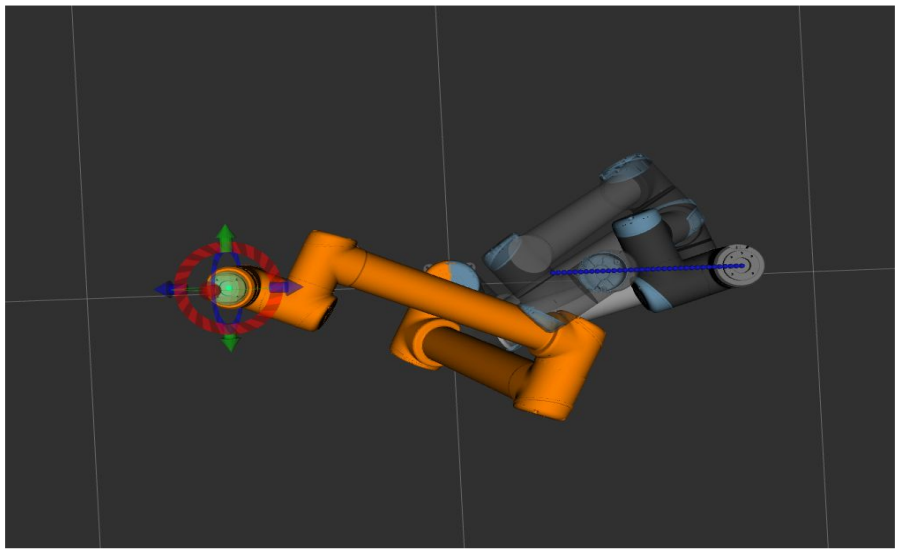

Figure 5: Cartesian path stops when singularity occurs. This happens for both IK solvers.

3.4. Test Results

For comparing[1] the two IK solvers, an Universal Robots UR10 robot arm, 6 DOF, is used. The test setup[2] is shown in figure 6. In the test setup, 1000 random reachable poses in the workspace, without any obstacles, are imposed. Then, for all the 1000 poses, the joint configuration is calculated with KDL and TRAC-IK inverse kinematic solvers.

The results can be found in table 1. It is clear, from the results, that TRAC-IK has a larger solvable rate and an lower average time of solving the IK then the KDL solver. So TRAC-IK surpasses KDL, for the UR10 robot arm, as kinematic solver. We also found that trac_ik performs better than KDL near the joint limit. And in the middle of joint space, the two success rates are more or less the same.

	KDL	**TRAC-IK**
average time	3.3 ms	0.6 ms
solve rate	36.3 %	99.4 %

Table 1: Comparison KDL and TRAC-IK for the Universal

[1] Computer specifications: OS: Ubuntu 16.04 (64bit), ROS Kinetic, CPU: Intel i7-6700hq, RAM: 8GB

[2] KDL and TRAC-IK default settings: timeout = 0.005, solver attempts: 3, search resolution: 0.005

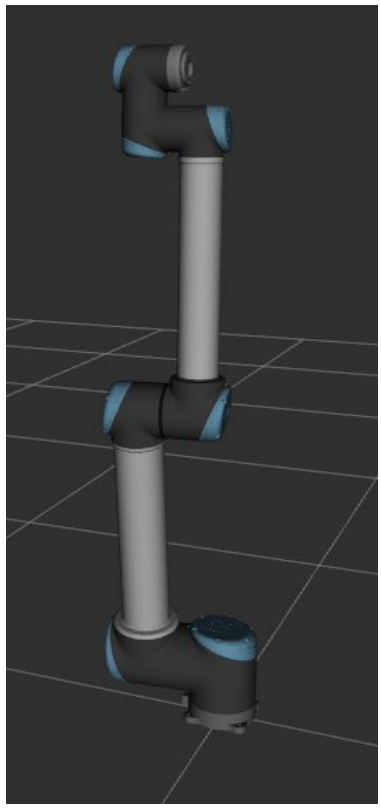

Figure 6: Kinematic Chain used for comparison KDL and TRAC-IK

4. Comparing Motion Planners

We are going to compare different existing planners in three different planning scenarios. The majority of our work focus on some sample based planners which are PRM, PRMstar, RRT, RRTConnect, KPIECE and LBKPIECE. But simple interpolation approach and search based planners are also briefly introduced and tested.

4.1. OMPL

For sample based planners, we are going to use implementations in the Open Motion Planning Library (OMPL)[2]. Sampling-based algorithms are the method of choice for many motion planning problems. [3] The key idea of sample-based algorithms is that the planner don't interact with obstacles directly but use a collision detector to do it. This makes such algorithms applicable to many kinds of problems.

The sample based planners provide probabilistic completeness. This means that if a solution exists, the probability of finding a solution converges to one as the number of samples reasoned over increases to infinity [3]. The disadvantage is that we can't tell whether a failure is caused by insufficient planning time or by impossible query. Another disadvantage

of sample-based planners are the jerky non-smooth produced path [6]. This could be improved using asymptotically optimal algorithms like PRMstar.

4.1.1. PRM

Probabilistic Roadmap Method (PRM) is a multi-query planner. It first try to build a roadmap through sampling states. Then search the roadmap to find a trajectory. A roadmap is a graph whose nodes are collision free states that randomly distributed over the configuration space. The edges of this graph are constructed by a local planner. Once we get the roadmap, it can be used for multiple queries.

4.1.2. PRMstar

PRMsatr (PRM*) is the asymptotically optimal version of PRM. Instead of having fixed numbers of connection of one node, it will gradually increase the number of connections and converge to the optimal path with time increasing.

4.1.3. RRT

Rapidly-exploring Random Trees (RRT) is the first single query planner. It will grow a tree from the start configuration until the goal configuration is reached by the tree. This algorithm is easy to understand. First, a random state q_{rand} is sampled. Then we search the entire tree to find the nearest node of q_{rand} in the tree. This node is called $q_{nearest}$. From $q_{nearest}$, randomly sample a input and duration to generate an edge dq. The end of dq is our new node q_{new} that will be added in the tree. However, experiments shows that RRT
tends to grow more new nodes near area that is already reached (The density of node is higher around initial state). We will see this disadvantage in our test.

4.1.4. RRTConnect

RRTConnect is the bidirectional version of RRT. It grows two trees. One from the start configuration and another from the goal configuration. It outperforms the original RRT planner in most cases.

4.1.5. KPIECE

Kinematic Planning by Interior-Exterior Cell Exploration (KPIECE) is the default planner of OMPL. It also grows a tree from start configuration like RRT. However, instead of sample a new state (a node or a point) it will sample a cell. A bias on exterior cell is setted so that the problem of RRT is solved. The cell is not randomly sampled but deterministically selected with a bias on exterior cells. Please refer to [4] for the exact formula. In high dimensional

configuration space, a projection to low dimension space is used so that we have less computational cost.

4.1.6. LBKPIECE

This is the lazy bidirectional version of KPIECE. Lazy means the planner will not check for collision before a trajectory is found. It will first assume all nodes and edges are collision free and search for the shortest path. Only after such a path is found will the collision detector be called. If there are collision nodes and edges, these nodes and edges will be deleted from the tree.[19]

4.2. CHOMP

Covariant Hamiltonian Optimization for Motion Planning (CHOMP) tries to minimize the objective functional called obstacle potential. This functional maps a trajectory to a real number. To avoid local optima, the Hamiltonian Monte Carlo (HMC) algorithm is applied to perturb the local optimal trajectory and restart the optimize process.

4.3. SBPL

Search-Based Motion Planning Library (SBPL) is a library that contains search-based planners that compute path using graph search method under a discrete representation of planning scene. There are two main steps in a SBPL planner: 1) find a discrete representation of the problem. 2) search the graph to find the best solution.

SBPL provides four classes for environment one 2D, two 3D and one 7D. Unfortunately, there is no environment for 6D robot arm like the ur10.

Name of planners	Type	Direction
PRM	sample-based multi-query	Unidirectional, one graph
PRMstar	sample-based multi-query	Unidirectional, one graph
RRT	sample-based single-query	Unidirectional, one tree
RRTConnect	sample-based single-query	Bidirectional, two trees
KPIECE	sample-based single-query	Unidirectional, one tree
LBKPIECE	sample-based single-query	Bidirectional, two trees

table 2: planners we're going to test.

4.4. Planning scenario

We will test 3 different scenario's for each planner.

4.4.1. Point to point scenario

Moving from one point to another. The shortest trajectory is a straight line in Cartesian space. Will a planner successfully find the straight line solution when there is no obstacles? In the straight line movement, the robot arm may encounter singularity problems. Near singularities, the inverse kinematic solver may not give the closest solution for next step, in other words, there might be a sudden jump from one pose to another. This sudden jump is impossible in real world. The challenge for planners is to find a smooth joint space trajectory so that the required movement is inside the limit of motors.

There is a different approach for finding a trajectory. Instead of planning in the configuration space, one can specify a set of waypoints in work space and try to find a (smooth) trajectory by interpolation. For a complicated case, finding waypoints is a challenge. But in straight line scenario, only the start pose and goal pose is needed. Therefore, this approach will be tested in this scenario and be compared with the planning in configuration space.

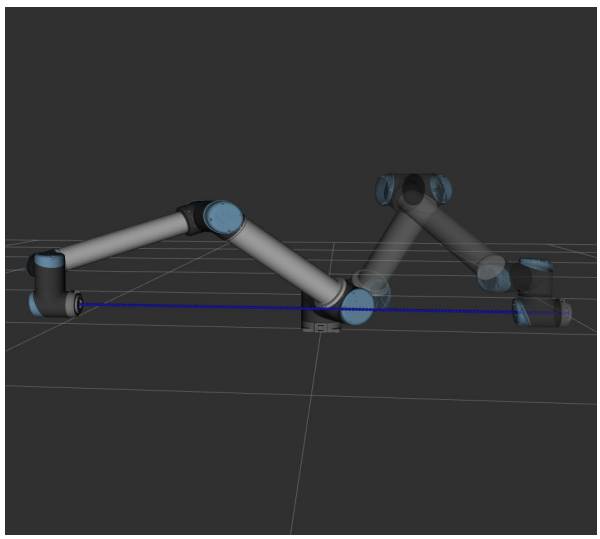

Figure 7: point to point scenario, robot follows a cartesian straight line

4.4.2. “Ping pong” scenario

A ball is placed on a ping pong table. The robot arm should first find a way to the ball and pick it up. Then move the ball to the other side of the table. Neither the robot nor the ball should touch the net in the middle of table.

Compared with the straight line case, we have obstacles in this scenario. The planner need to find a collision free path using the collision detector.

We will also ask the robot arm to move from the ball to the other side of the table (free space) and move back. We call such movements as “from the ball to the free space” and “from free space to the ball”. It turns out that although the coordinate is symmetric, the planners have different performance. Please see the test results for a detailed explanation.

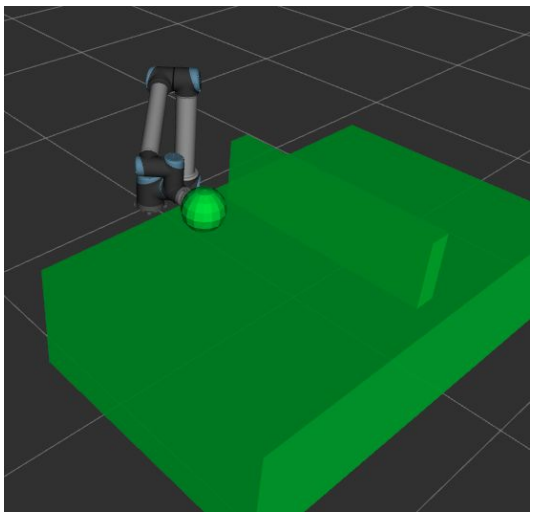

Figure 8: Ping pong scenario

4.4.3. Shelf scenario

A ball is placed inside a shelf. The planner should give a collision free path that guides the robot going inside the shelf and pick the ball. Then put the ball in a box outside the shelf.
The shelf and the box provide deep and narrow collision free space. Compared with the ping pong case, there are a lot more constraints which make finding a collision free path a lot more difficult.

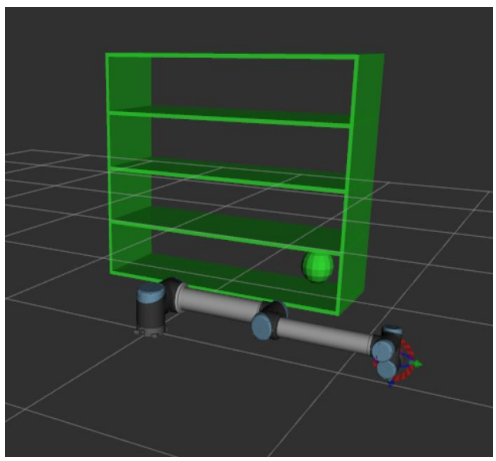

Figure 9: Shelf scenario

4.5. Metrics

Our use case is a dynamic work space shared with human and/or other robots. There are two phase in robot operation: planning and execution. We need a planner that give both a short planning time and a short execution time. For the first phase, we use planning time as the metric. And for the second phase. we measure the length of trajectory. Besides, the trajectory should be smooth which means little shaking and vibration. Of course, the chance to successfully find a trajectory is important.

For comparing each planner, we evaluate them on the following metrics:

- Success rate: This is defined as :
  (number of successful planning)/(total attempts)
- Planning time: How much time it takes for the planner to find a collision free path. It includes not only the time use for search a path but also the time for inverse kinematic solver.
- Path smoothness: We use jerk cost as a measurement for path smoothness. It is defined as $\sum_{start}^{goal}$ [j(t)^2] Δ t. j(t) is the jerk in the trajectory.[8] A jerk means change of force. The larger the jerk, the bigger the shock which might cause fatigue and failure in mechanical structures. Therefore, a small jerk cost is desirable.
- Path length: The distance that the end effector travels.

4.6. Test Results

4.6.1. Straight line scenario

When a straight motion in cartesian space has to be followed. MoveIt set points, called waypoints, on the straight cartesian path that the robot has to follow. Then by interpolation through these waypoints, MoveIt can generate a cartesian straight motion. The occurrence of singularities during a straight motion can cause different interpolation results . This is due to the inverse kinematic solver that is used. It is known, that TRAC-IK is better in handling situations where singularities occur.

4.6.1.1. Test group 1

In this section, we will test the interpolation between start pose and goal pose in Cartesian space.

Set up 1:

Search resolution [m]	Time out [s]	Maximum attempts	Start pose	Goal pose
0.005	0.01	10	(0, 0.2, 0.5) (1, 0, 0, 0)	(1.18, 0.2, 0.5) (1, 0, 0, 0)

Results:

Test 1	TRAC-IK	KDL
Rate path is followed	98.54%	60.57%
Number of waypoints	129	32
Planning time [s]	0.055677	0.021438
Trajectory length [m]	1.180089	0.249469
Smoothness [rad^2/s^5]	18597.74	17137.47

Table 3, Results of straight motion scenario

Set up 2:

Search resolution [m]	Time out [s]	Maximum attempts	Start pose	Goal pose
0.005	0.01	10	(0.7, -0.7, 0.4) (0.993, 0, 0, 0.122)	(-1, -0.3, 0.4) (0.993, 0, 0, 0.122)

Results:

Test 1	TRAC-IK	KDL
Rate path is followed	100%	100%
Planning time [s]	0.0500	0.0439
Trajectory length [m]	1.75	1.75
Smoothness [rad^2/s^5]	25418	18315

table 4, Results of point to point scenario

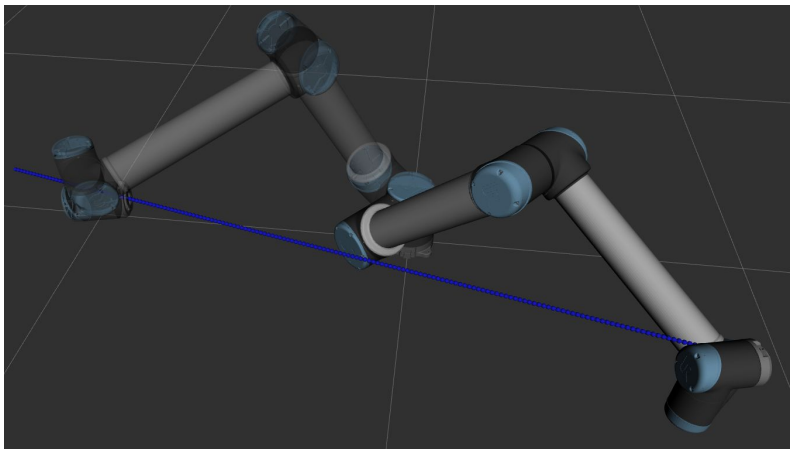

Figure 10: Visualisation of the straight line that is followed by the robot end-effector

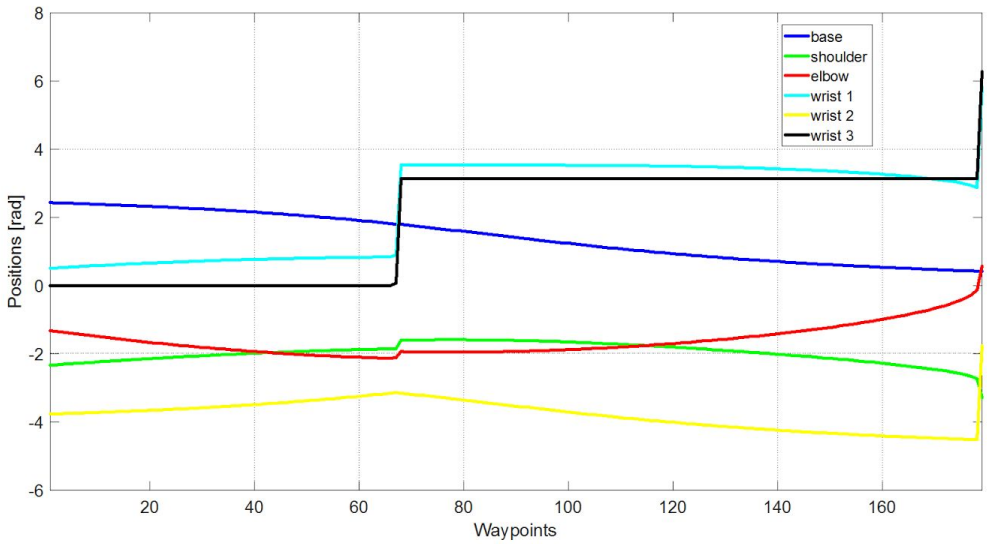

Figure 11: Position - trac_ik. We can find a very large jump around 70th waypoint

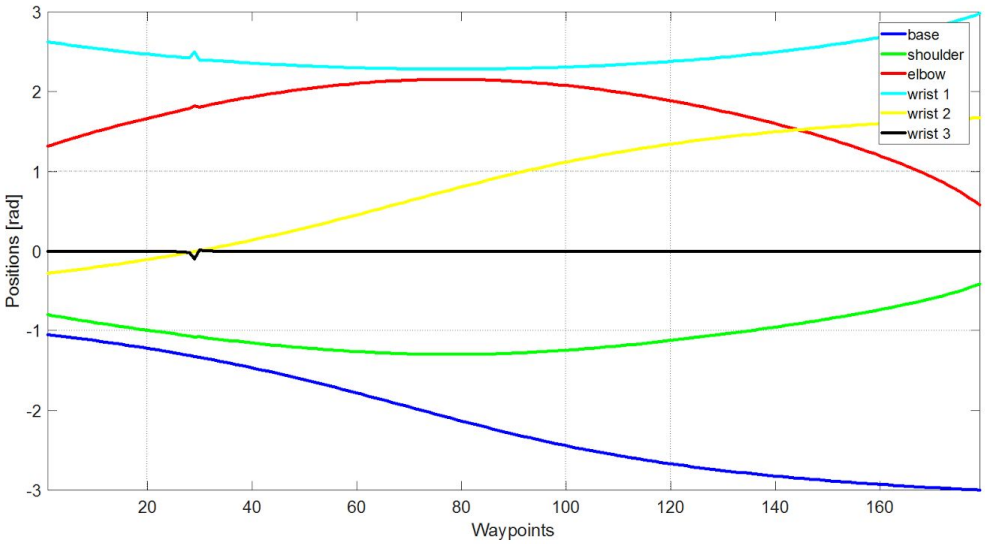

Figure 12: Position - KDL A smaller jump at the 30th waypoint

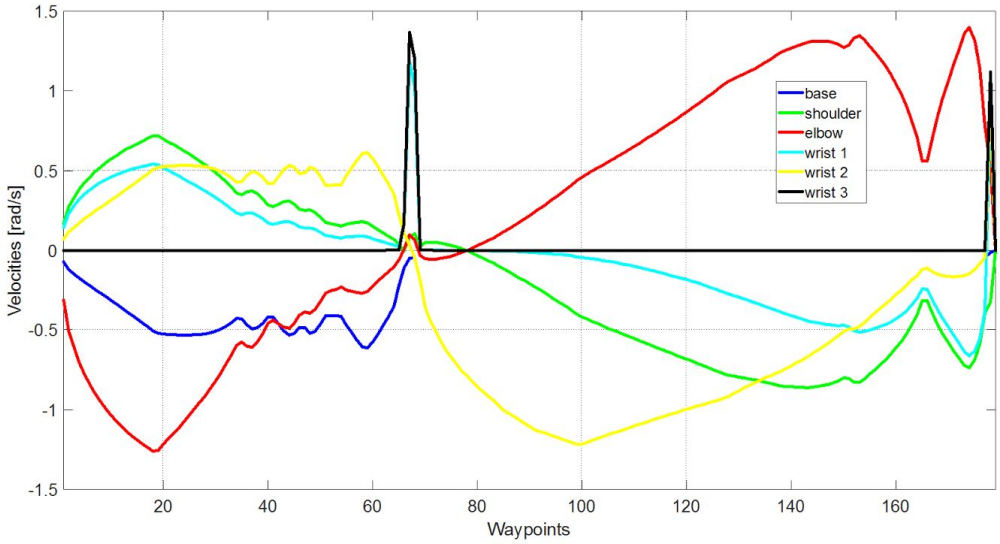

Figure 13: Velocity - trac_ik Notice the peak at the jump point

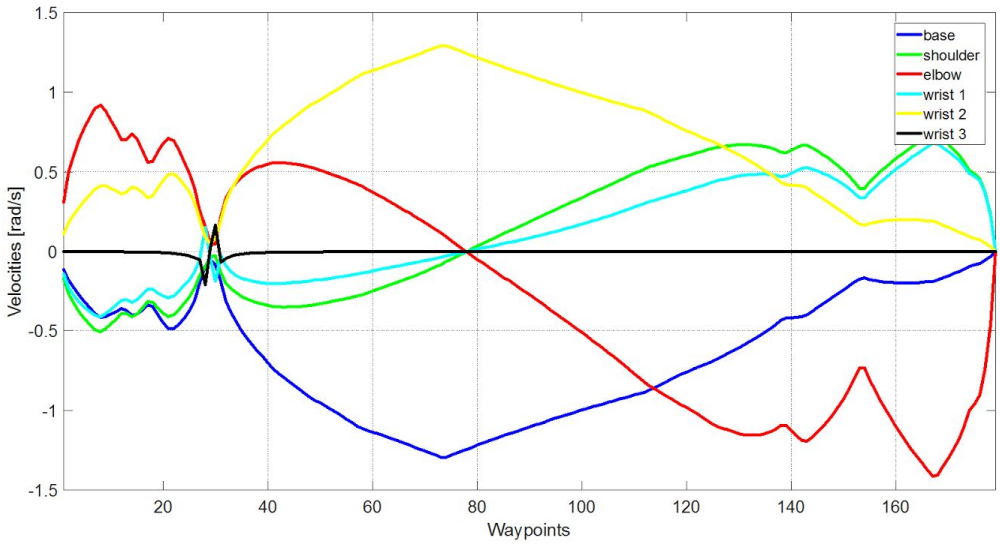

Figure 14: Velocity - KDL Although there is only one small jump, the velocity is high

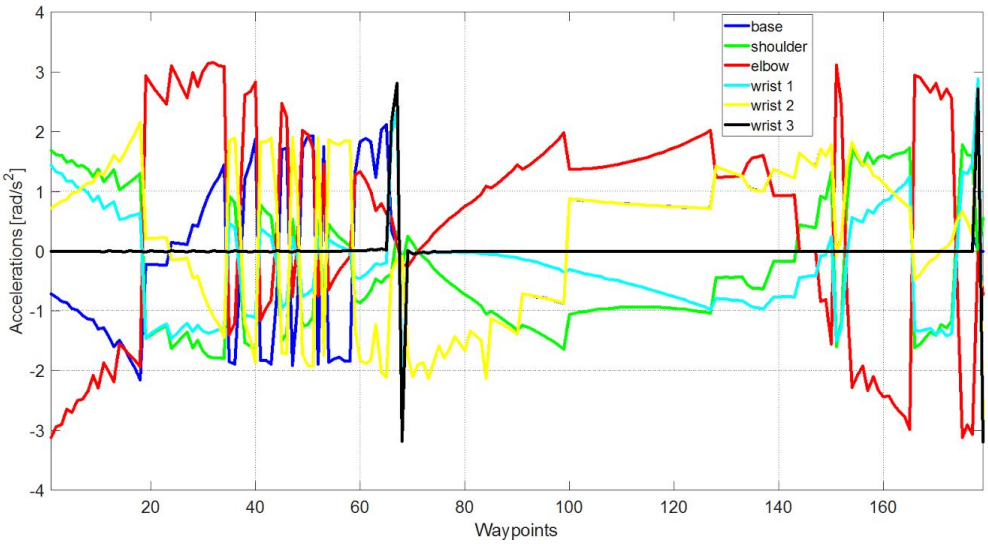

Figure 15: Acceleration - trac_ik The smoothness of this trajectory is very bad

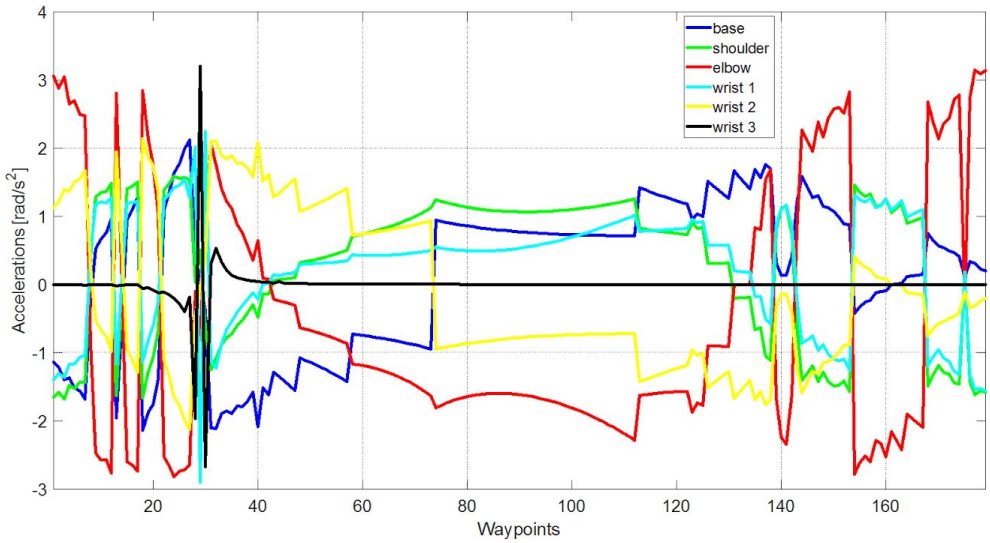

Figure 16: Acceleration - KDL The smoothness of this trajectory is very bad

4.6.1.2 Test group 2

In this section, we use the sample based planners in OMPL to do the same task.

Set up:

Start pose	Goal pose	Planner tolerance [m]	Maximum planning time [s]
(0.7, -0.7, 0.4) (0.993, 0, 0, 0.122)	(-1, -0.3, 0.4) (0.993, 0, 0, 0.122)	0.01	1.0

Results:

Test 2	Planner	Total attempts	Success rate %	Trajectory length [m]	Trajectory smoothness [rad^2/(s^5)]	Planning time [s]
Average value	PRMstar	200	100	2.09	266	1.0*
Shortest trajectory	PRMstar	--	--	2.09	266	1.0*

table 5-1, PRMstar results of scenario 1

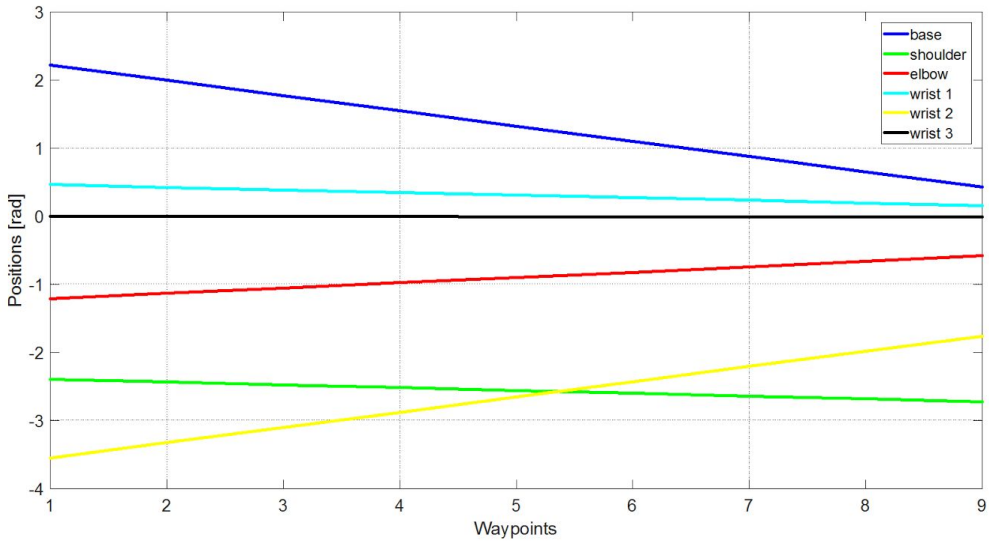

Figure 17: Position of each joints returned from PRMstar. No sudden jump

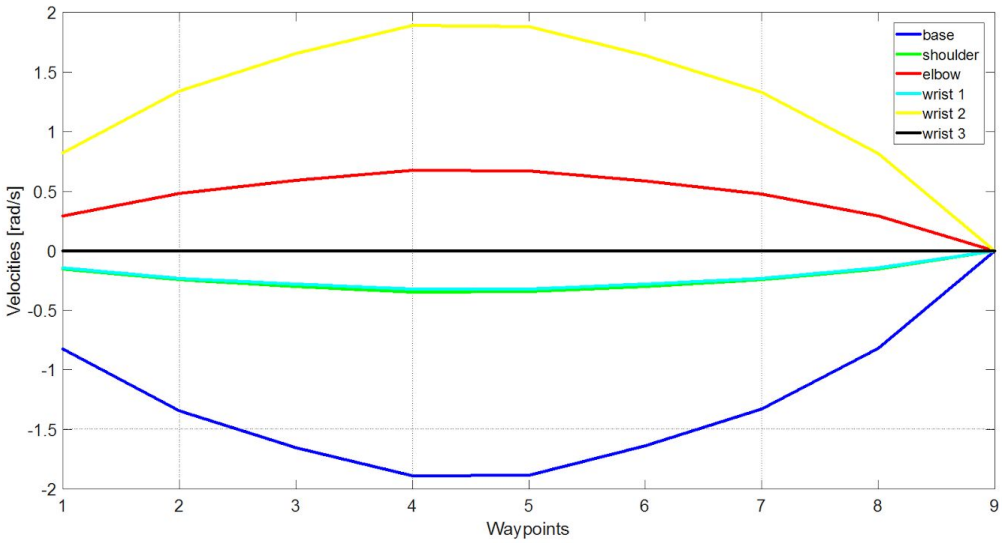

Figure 18: Velocity of each joints returned from PRMstar

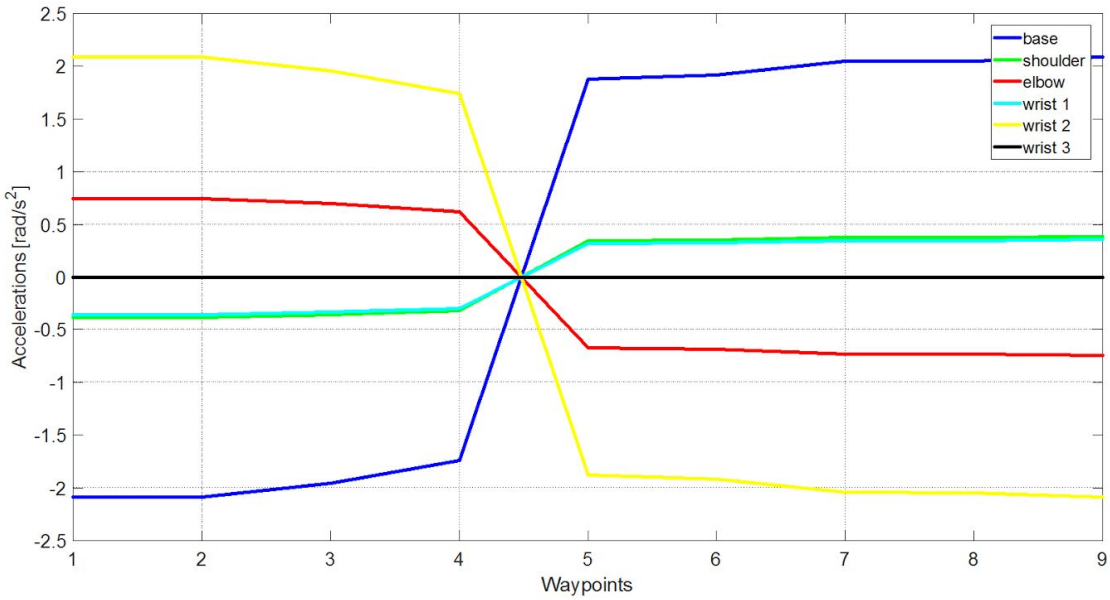

Figure 19: Acceleration of each joint returned from PRMstar. A lot smoother than interpolation approach.

Test 3	Planner	Total attempts	Success rate %	Trajectory length [m]	Trajectory smoothness [rad^2/(s^5)]	planning time [s]
Average value	PRM	200	100	2.09	266	1.0*
Shortest trajectory	PRM	--	--	2.09	266	1.0*

table 5-2, PRM results of scenario 1

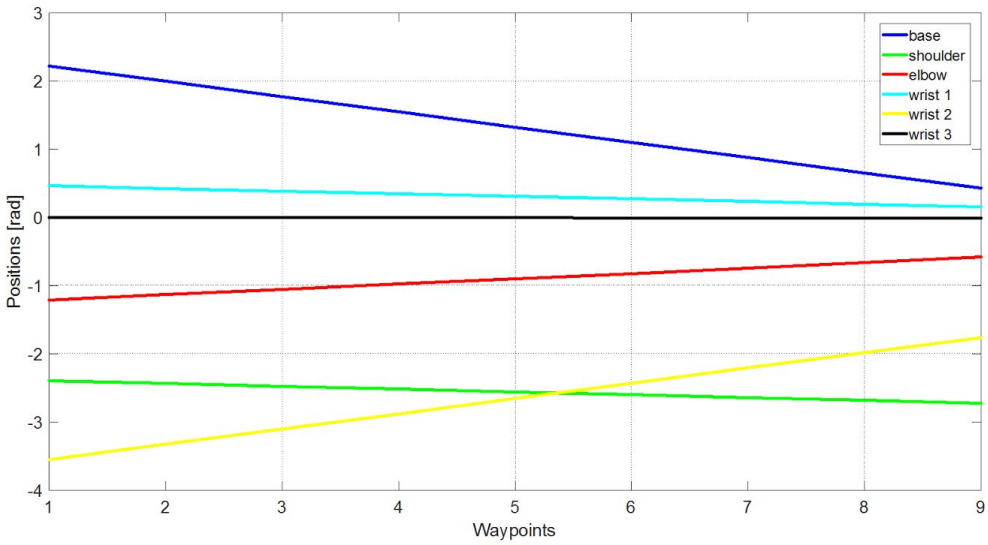

Figure 20: Position of each joints returned from PRM. Very similar to PRMstar

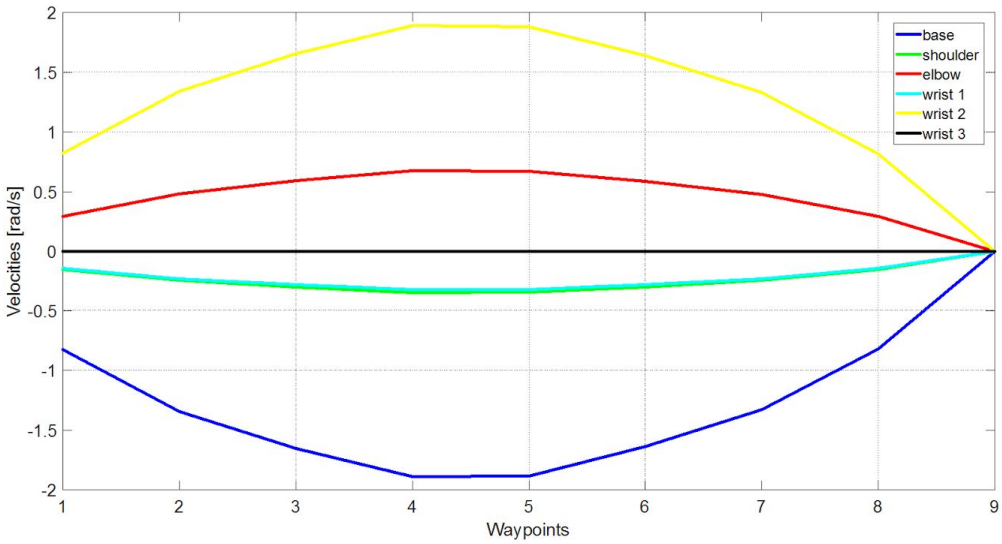

Figure 21: Velocity of each joints returned from PRM. Very similar to PRMstar

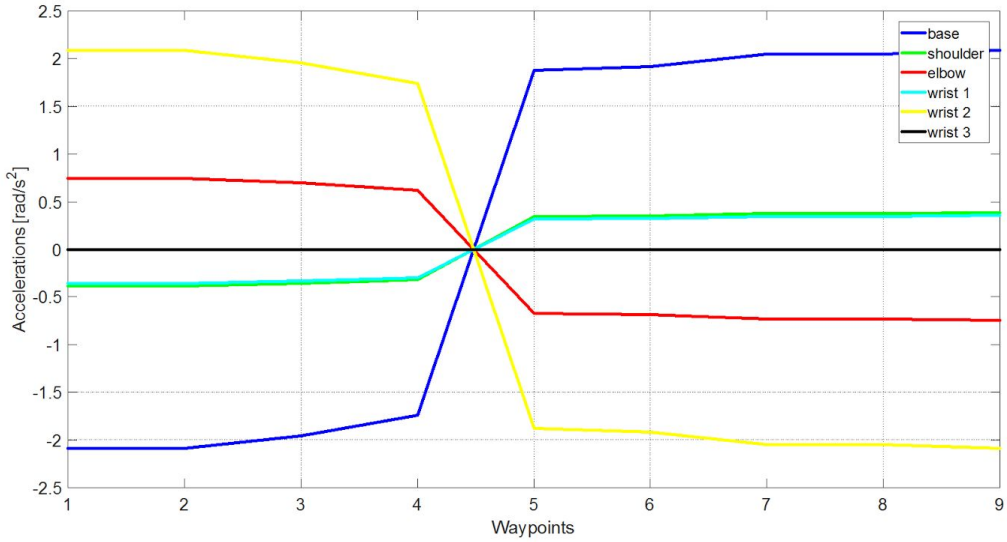

Figure 22: Acceleration of each joint returned from PRM. Very similar to PRMstar

Test 4	Planner	Total attempts	Success rate %	Trajectory length [m]	Trajectory smoothness [rad^2/(s^5)]	Planning time [s]
Average value	RRTConnect	200	100	2.10	1500	0.0233
Shortest trajectory	RRTConnect	--	--	2.09	601	0.0191

table 5-3, RRTConnect results of scenario 1

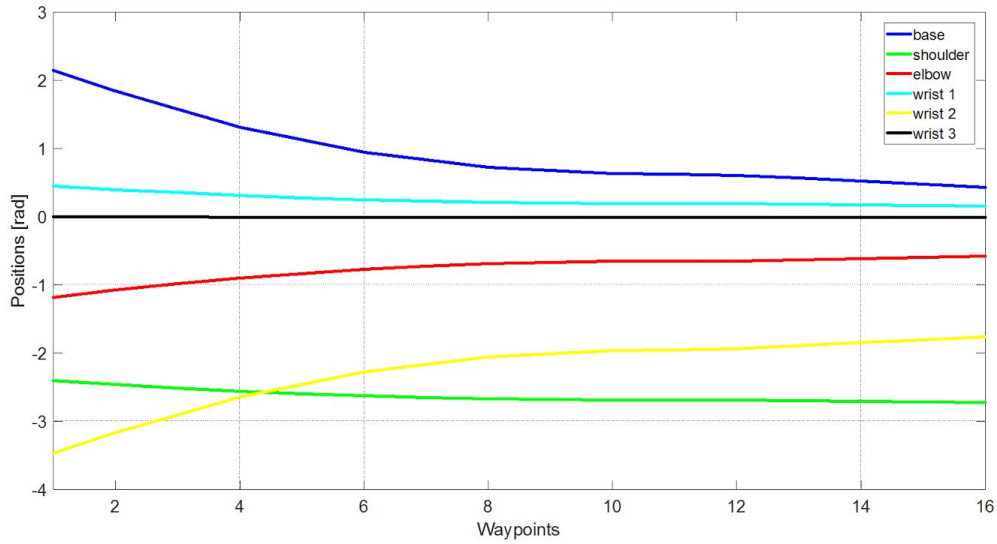

Figure 23: Position of each joints returned from RRTConnect.

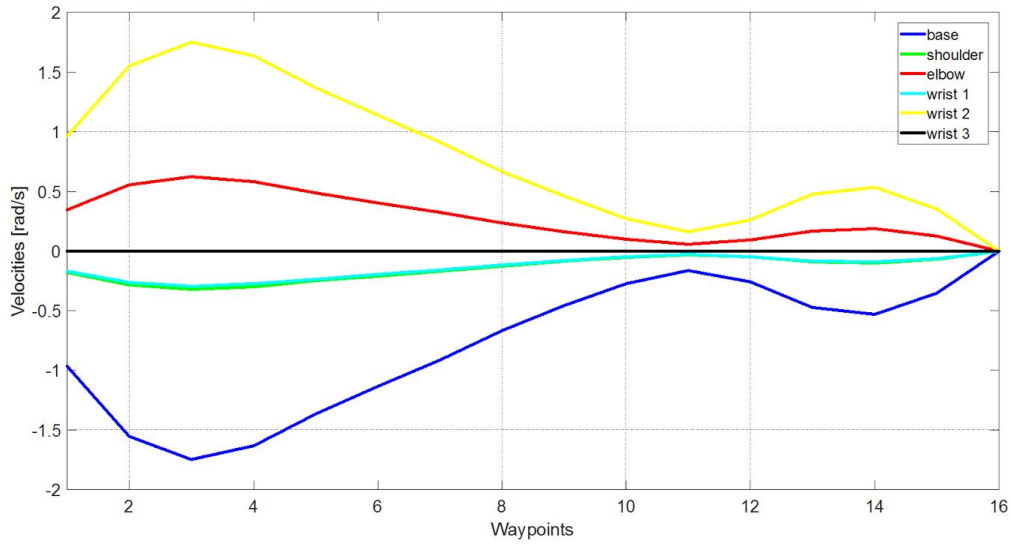

Figure 24: Velocity of each joints returned from RRTConnect. We observe a different pattern compared with PRM and PRMstar.

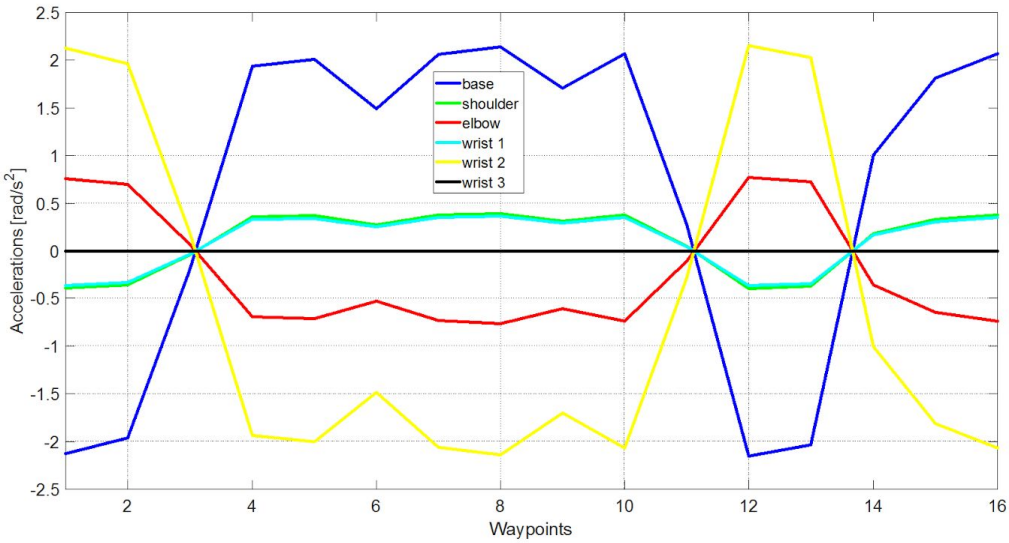

Figure 25: Acceleration of each joint returned from RRTConnect. It is less smoother compared with PRM and PRMstar

Test 5	Planner	Total attempts	Success rate %	Trajectory length [m]	Trajectory smoothness [rad^2/(s^5)]	Planning time [s]
Average value	RRT	200	100	2.09	280	0.0262
Shortest trajectory	RRT	--	--	2.09	266	0.0256

table 5-4, RRT results of scenario 1

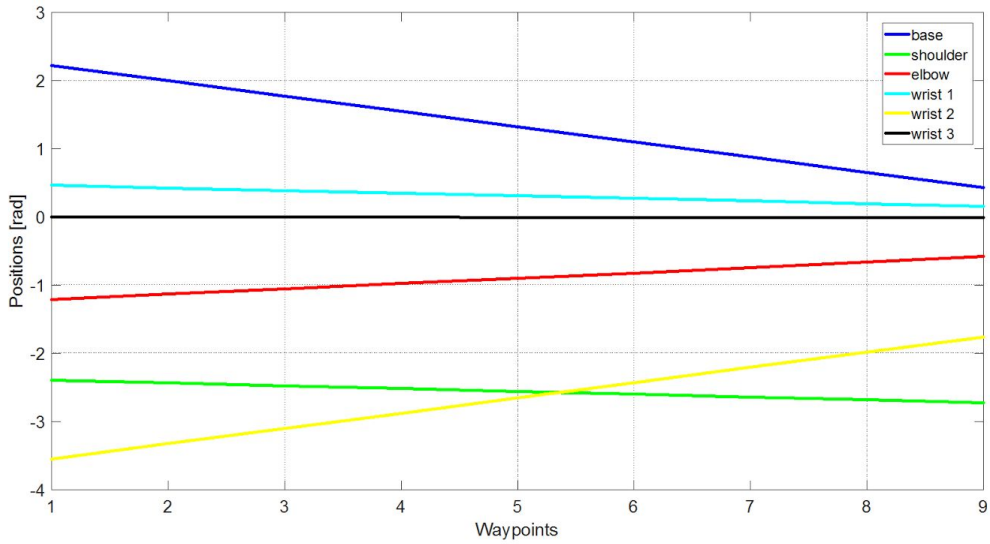

Figure 26: Position of each joints returned from RRT

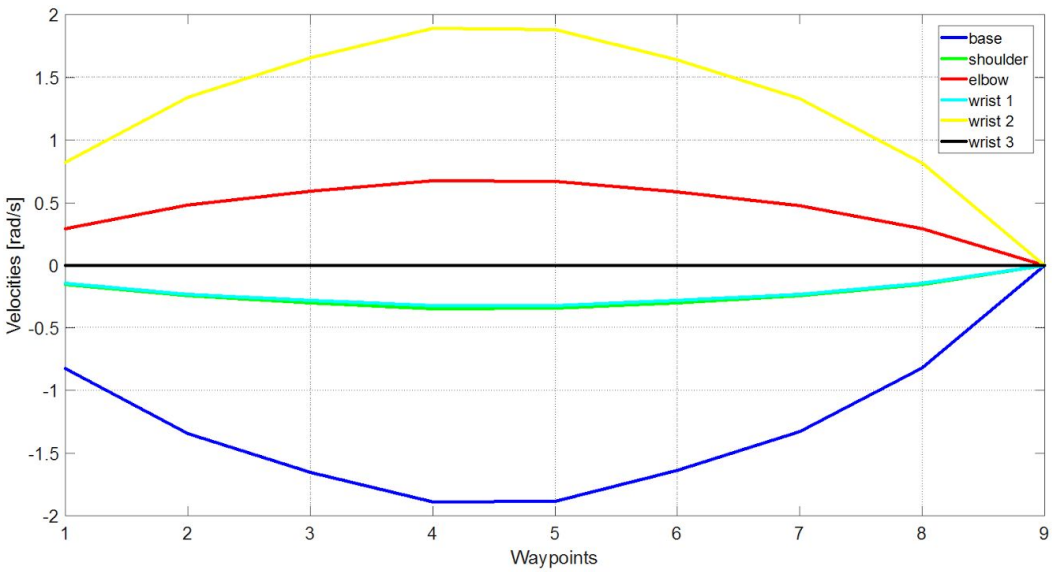

Figure 27: Velocity of each joints returned from RRT. Notice the similarity between RRT and PRM/PRMstar and it is different with RRTConnect

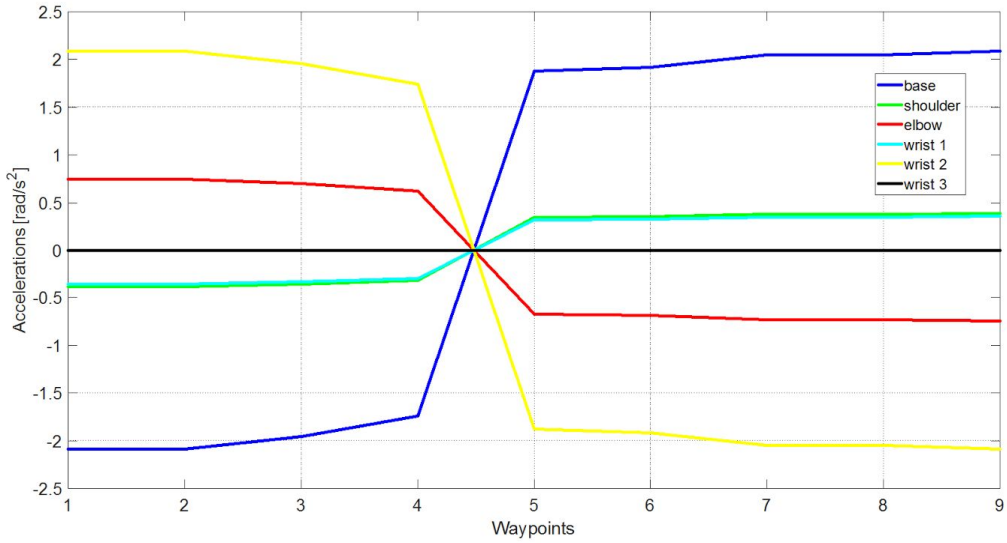

Figure 28: Acceleration of each joint returned from RRT. Different with RRTConnect but similar with PRM/PRMstar

Test 6	Planner	Total attempts	Success rate %	Trajectory length [m]	Trajectory smoothness [rad^2/(s^5)]	Planning time [s]
Average value	KPIECE	200	100	2.10	1559	0.0441
Shortest trajectory	KPIECE	--	--	2.09	266	0.0254

table 5-5, KPIECE results of scenario 1

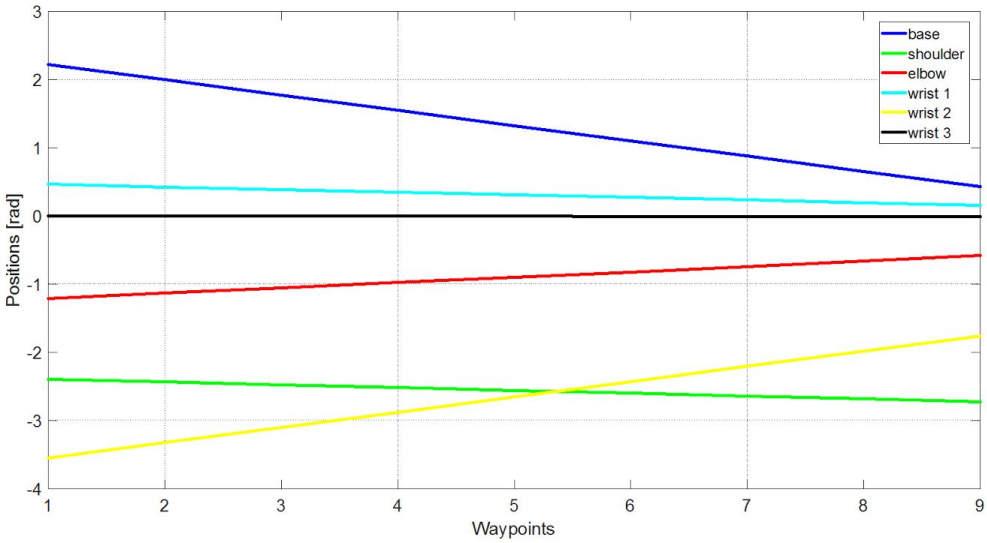

Figure 29: Position of each joints returned from KPIECE

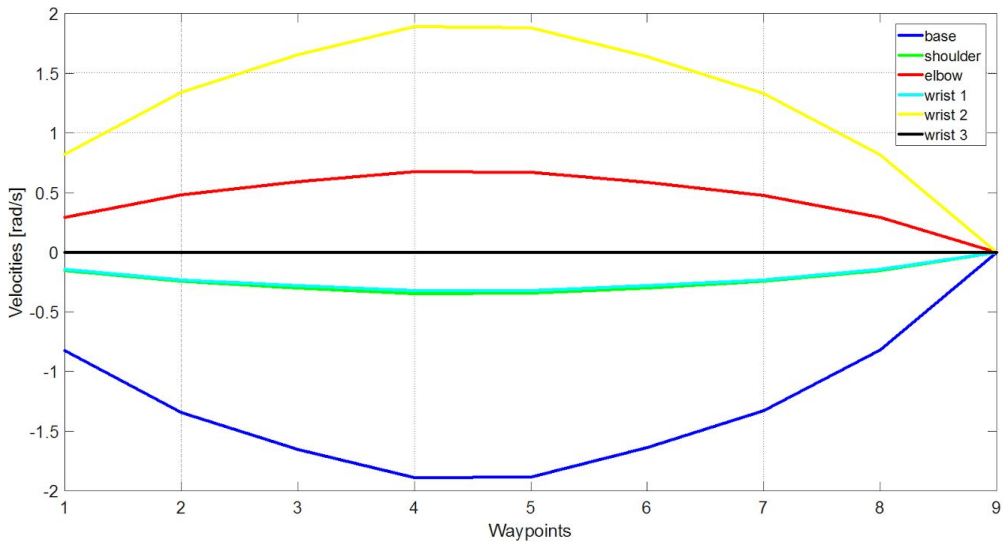

Figure 30: Velocity of each joints returned from KPIECE. Similar with PRM/PRMstar/RRT

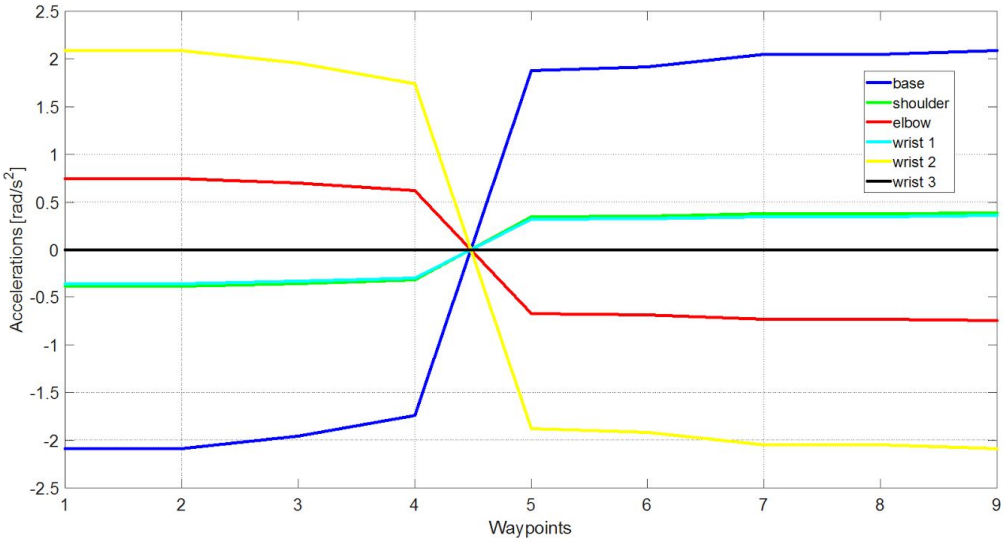

Figure 31: Acceleration of each joint returned from KPIECE.Similar with RRT/PRM/PRMstar

Test 7	Planner	Total attempts	Success rate %	Trajectory length [m]	Trajectory smoothness [rad^2/(s^5)]	Planning time [s]
Average value	LBKPIECE	200	77.0	2.10	1463	0.5894
Shortest trajectory	LBKPIECE	--	--	2.09	1514	0.6800

table 5-6, LBKPIECE results of scenario 1

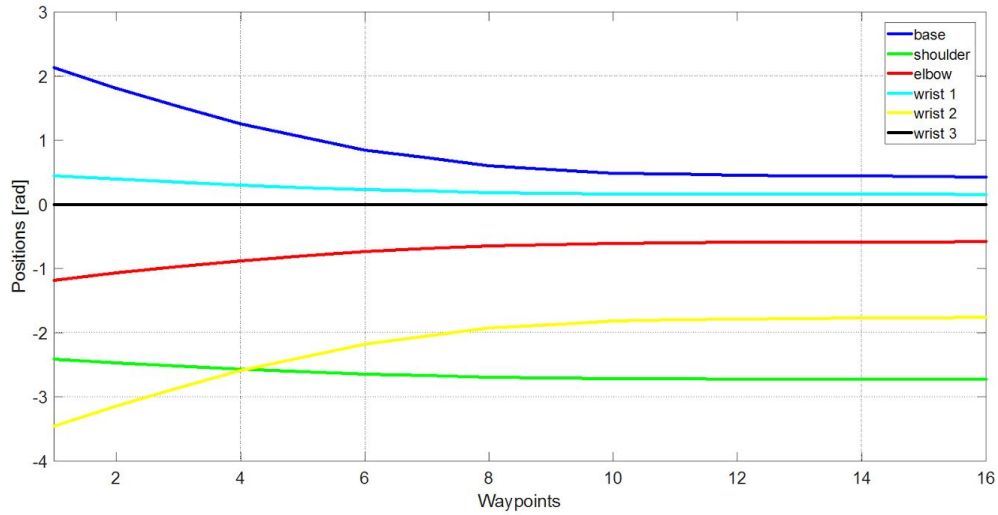

Figure 32: Position of each joints returned from LBKPIECE.

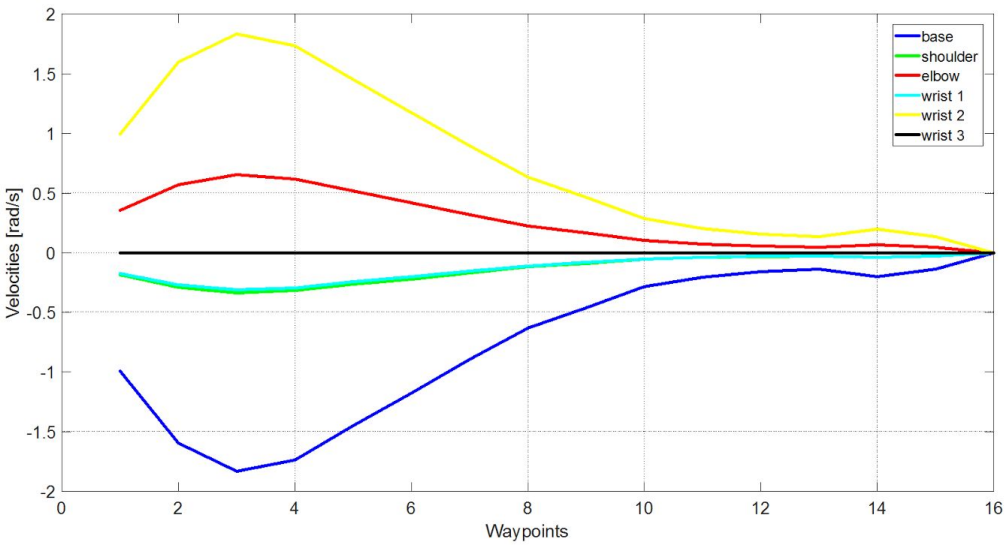

Figure 33: Velocity of each joints returned from LBKPIECE. Different with KPIECE but similar with RRTConncet.

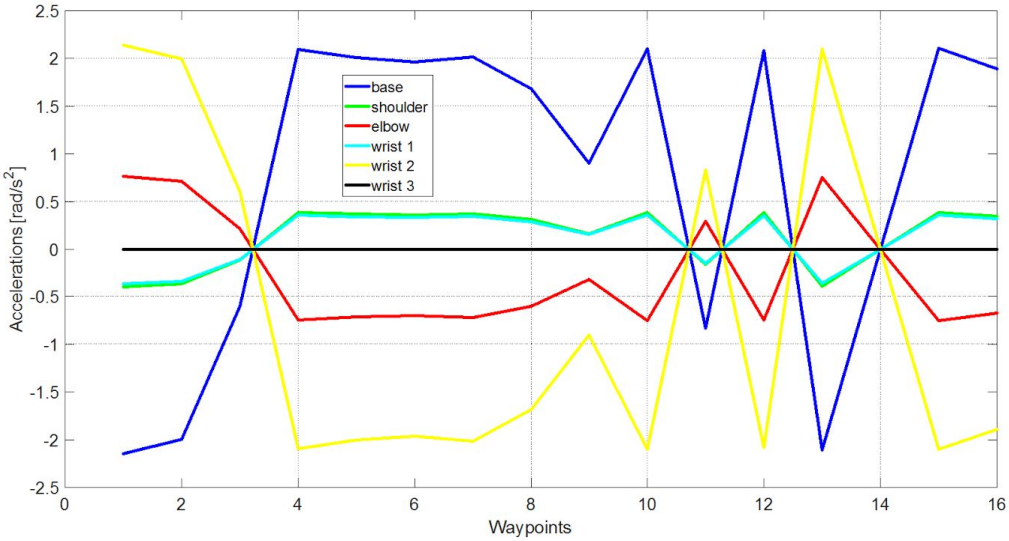

Figure 34: Acceleration of each joint returned from LBKPIECE. Not smooth and looks similar to RRTConnect

Test 8	Planner	Total attempts	Success rate %	Trajectory length [m]	Trajectory smoothness [rad^2/(s^5)]	Planning time [s]
Average value	CHOMP	10*	90.0	3.88	7033	0.0477
Shortest trajectory	CHOMP	--	--	3.24	9053	0.0284

table 5-7, CHOMP results of scenario 1

** We get an core dumped error with total attempts more than 10, sometimes we get such an error even with 5 total attempts. And the Rviz always reports an error. There are some implementation problems in the CHOMP plugin.*

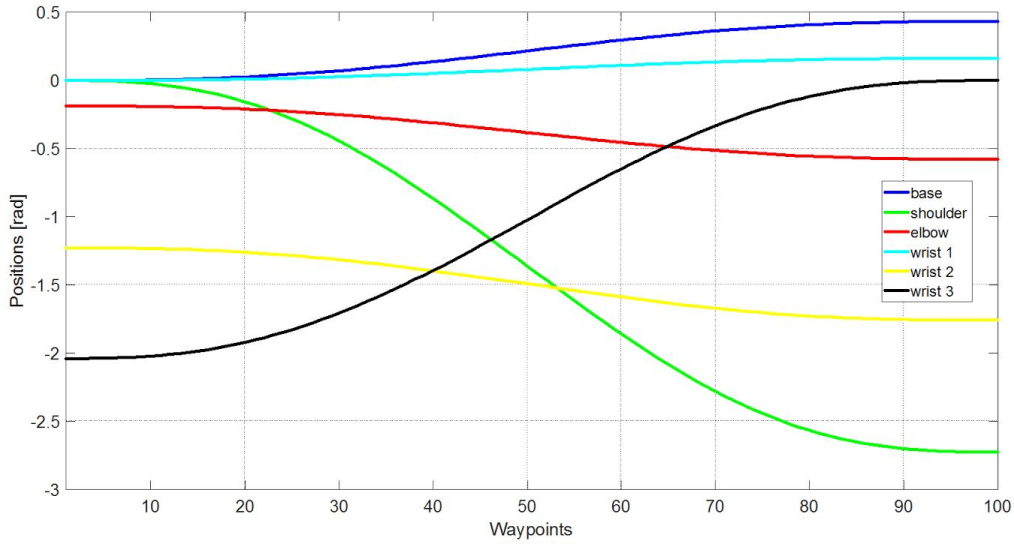

Figure 35: Position of each joints returned from CHOMP

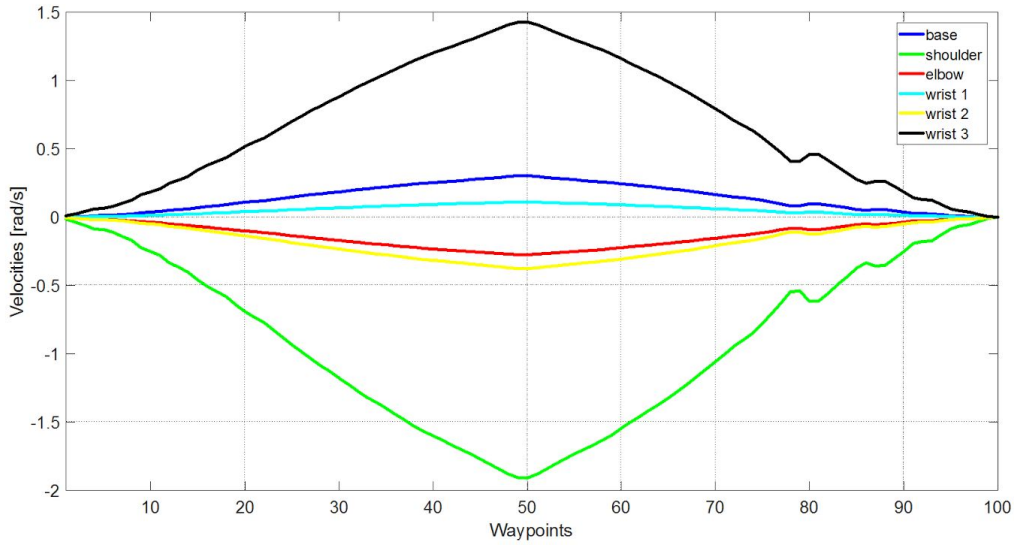

Figure 36: Velocity of each joints returned from CHOMP

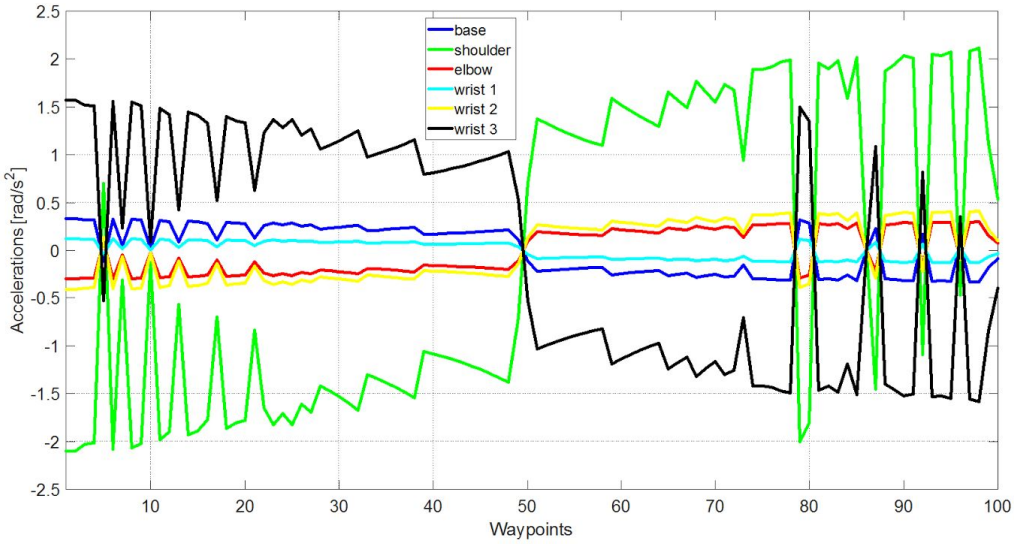

Figure 37: Acceleration of each joints returned from CHOMP. To the author's eyes, CHOMP gives a trajectory pattern very similar to PRM/PRMstar/RRT/KPIECE. But there are many perturbations in the trajectory.

4.6.1.3. Discussion

We can find that although we get the shortest straight line trajectory from the Cartesian space interpolation, the other metrics of interpolation is pretty bad. There is sudden jump in the trajectory; there are a lot of shaking and vibration; the planning time is a little longer than other planners. If we enable a filter that reject a trajectory with jump and ask for re-interpolation, could the results become better? We think there wouldn't be too much improvement. Because which solution will IK solver converge to is not predictable before we actually run the solver. But once we have a jump, we have to tuning the initial values of the IK solver to eliminate the jump. This procedure will cost much time.Therefore, we suggest not to use Cartesian space interpolation even in a simple point to point scenario.

In the second test group, we found an interesting behaviour. Although RRT and RRTConnect (KPIECE and LBKPIECE) use the same search method, they give a very different trajectory. However, RRT, KPIECE and PRM/PRMstar give very similar results while RRTConnect and LBKPIECE's results looks more similar. The common feature which RRT, KPIECE and PRM/PRMstar share is that they are all unidirectional planners. And RRTConnect and LBKPIECE are bidirectional planners.

We also noticed that the smoothness of unidirectional planners' results are much better than the smoothness of bi-directional planner's. The reason is that every edge between two nodes is a motion primitive. In unidirectional planners all of these primitives are in the same direction (from node i to node i+1). But in bidirectional planners, there are two trees and all the edge in the second tree (grown from the goal state) is in the reverse direction. Besides, there are many times of shifting from one tree to another.

The shortest length these planners returns are more or less the same - about 2.09m.

The trajectory returned by CHOMP looks similar to PRM/PRMstar/RRT/KPIECE. However, there are many perturbations in the trajectory. The reason might be the HMC perturbation used in CHOMP to avoid local optima. But we can't be sure about that. The CHOMP plugin is not robust, we get many bugs from it. So it might be an implementation problem.

4.6.2. "Ping pong" scenario

For sample-based planner in OMPL, we perform two sets of tests using different inverse kinematic solver. The first set uses the KDL and the second set uses trac_ik. The test settings and the test result is listed below. In the figures, the red line indicates the longest trajectory and the blue line is the shortest.

4.6.2.1 Test group 1

Set up:

IK solver	Longest_valid_segment_fraction	Planner tolerance [m]	Maxim planning time [s]
KDL	0.005	0.01	2.0

OS distro	ROS distro	CPU	RAM
Ubuntu 16.04 - (X86) 64bit	kinetic	intel 6700hq	8GB

Results:

Test 1	Planner	Total attempts	Success rate %	Mean Trajectory length [m]	Mean Trajectory smoothness [rad^2/(s^5)]	Mean planning time [s]
From free space to the ball	KPIECE	1000	62.5	4.31	4081	0.925
From the ball to free space	KPIECE	1000	100.0	4.53	4581	0.187

table 6-1, KPIECE with KDL results of scenario 2

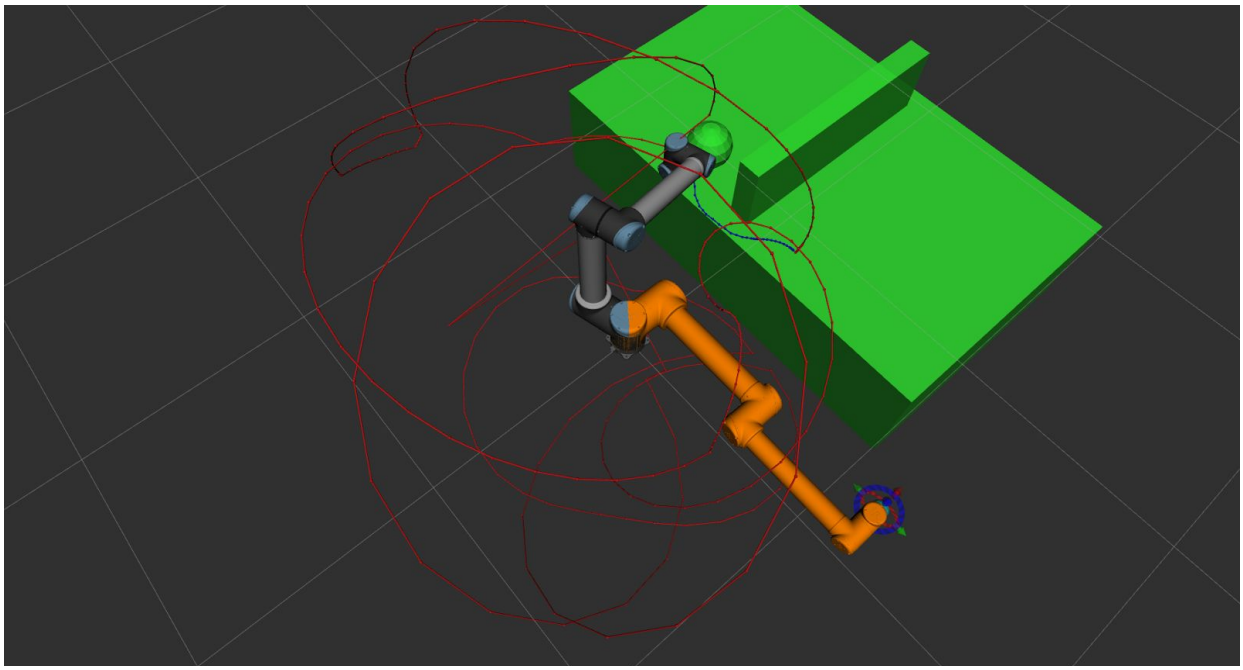

Figure 38: Result from KPIECE. The blue line is the shortest and the red line is the longest

Test 2	Planner	Total attempts	Success rate %	Mean Trajectory length [m]	Mean Trajectory smoothness [rad^2/(s^5)]	Mean planning time [s]
From free space to the ball	LBKPIECE	1000	87.0	8.1	5330	1.206
From the ball to free space	LBKPIECE	1000	85.3	6.38	4888	1.182

table 6-2, LBKPIECE with KDL results of scenario 2

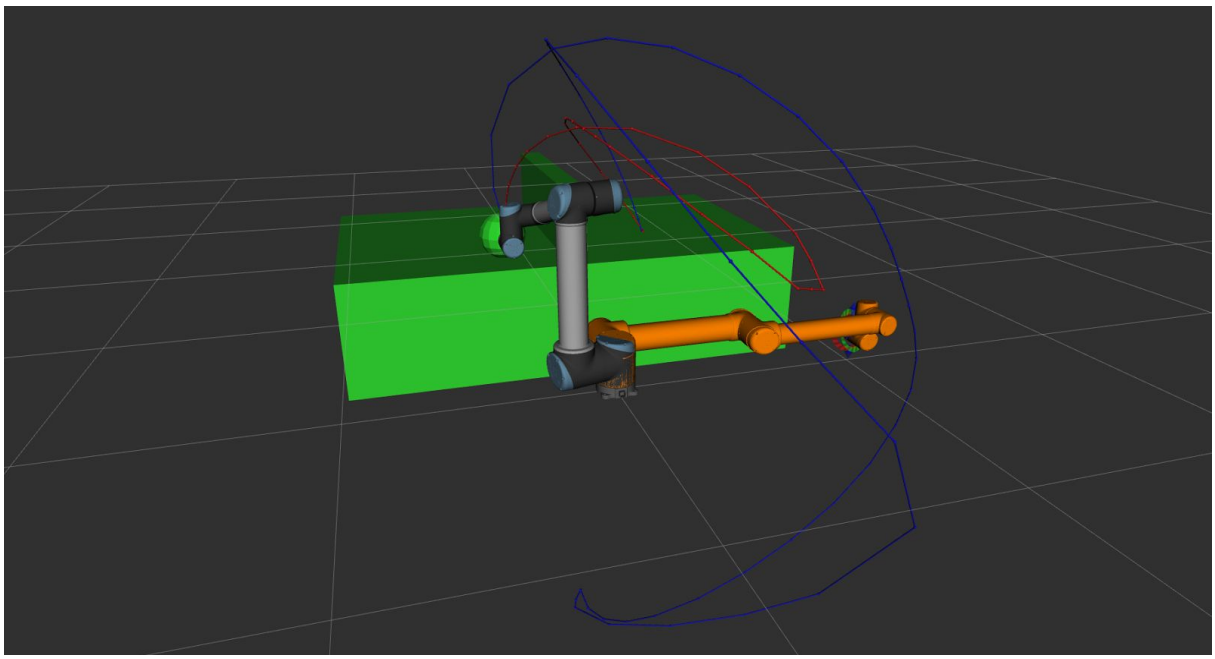

Figure 39: Result from LBKPIECE. The blue line is the shortest and the red line is the longest

Test 3	Planner	Total attempts	Success rate %	Mean Trajectory length[m]	Mean Trajectory smoothness [rad^2/(s^5)]	Mean planning time [s]
From free space to the ball	RRTConnect	1000	99.5	4.62	5606	0.0691
From the ball to free space	RRTConnect	1000	99.9	4.52	5771	0.0500

table 6-3, RRTConnect with KDL results of scenario 2

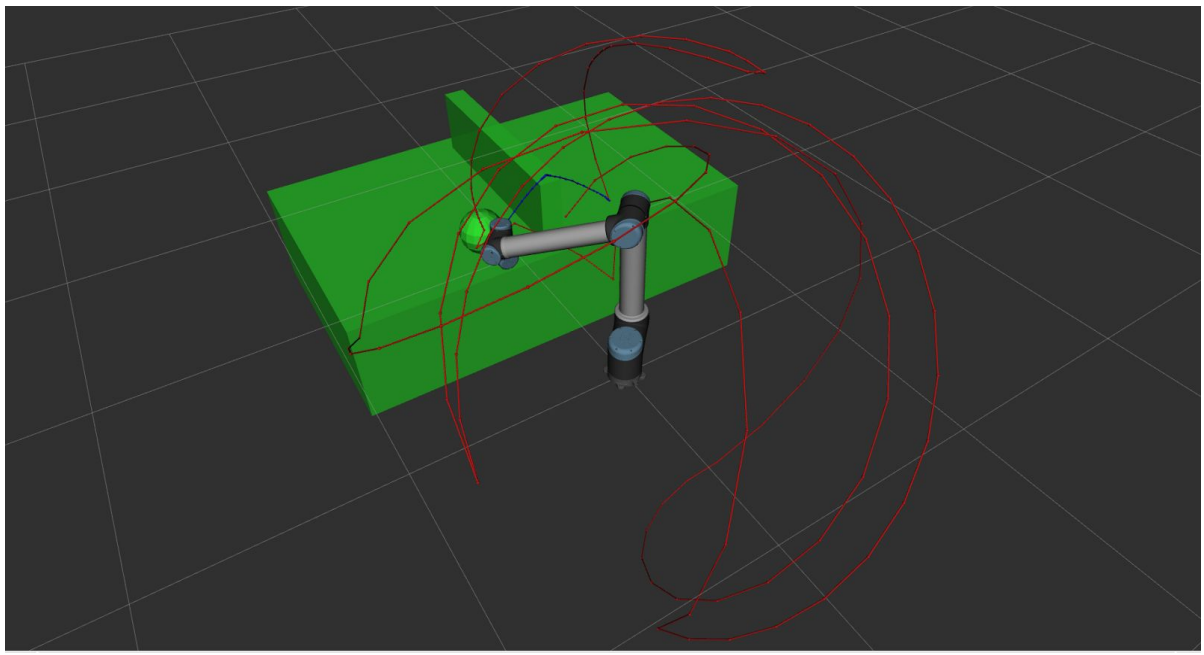

Figure 40: Result from RRTConnect. The blue line is the shortest and the red line is the longest

Test 4	Planner	Total attempts	Success rate %	Mean Trajectory length [m]	Mean Trajectory smoothness [rad^2/(s^5)]	Mean planning time [s]
From free space to the ball	PRM	1000	17.2	5.04	502	0.1*
From the ball to free space	PRM	1000	8.4	4.9	555	0.1*

table 6-4, PRM with KDL results of scenario 2

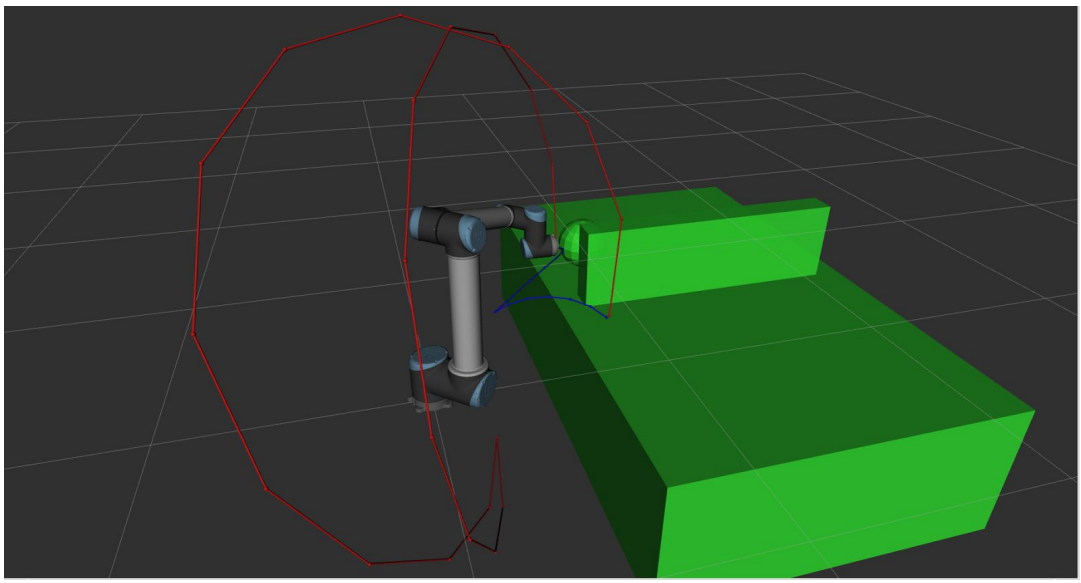

Figure 41: Result from PRM. The blue line is the shortest and the red line is the longest

Test 5	Planner	Total attempts	Success rate %	Mean Trajectory length [m]	Mean Trajectory smoothness [rad^2/(s^5)]	Mean planning time [s]
From free space to the ball	PRMstar	1000	63.6	9.44	554	0.1*
From the ball to free space	PRMstar	1000	73.9	10.31	501	0.1*

table 6-5, PRMstar with KDL results of scenario 2

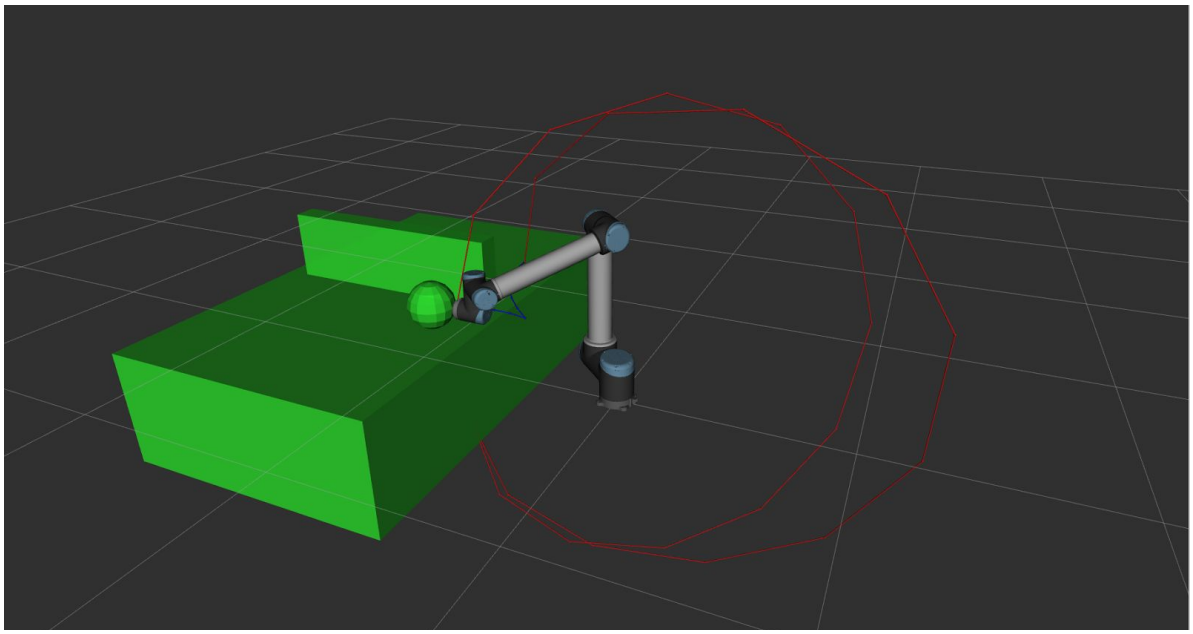

Figure 42: Result from PRMstar. The blue line is the shortest and the red line is the longest

* PRM and PRMstar are multi-query planners. They will run two process at the same time. One is for roadmap and another is for the path search. They will use all the time that is given to them. We set the maximum planning time as 0.1s for these two planners.

RRT and RRTstar performs bad in our test. They can only successfully find one path out of (about) 30 attempts. Explanation about this result is in the discussion section.

4.6.2.2 Test group 2

Set up:

IK solver	Longest_valid_segment_fraction	Planner tolerance/[m]	Maximum planning time/[s]
trac_ik	0.005	0.01	2.0

OS distro	ROS distro	CPU	RAM
Ubuntu 16.04	kinetic	intel 6700hq	8GB

Results:

Test 6	Planner	Total attempts	Success rate %	Mean Trajectory length [m]	Mean Trajectory smoothness [rad^2/(s^5)]	Mean planning time [s]
From free space to the ball	KPIECE	1000	100.0	6.25	3173	0.132
From the ball to free space	KPIECE	1000	99.9	6.89	3892	0.0702

table 7-1, KPIECE with trac_ik results of scenario 2

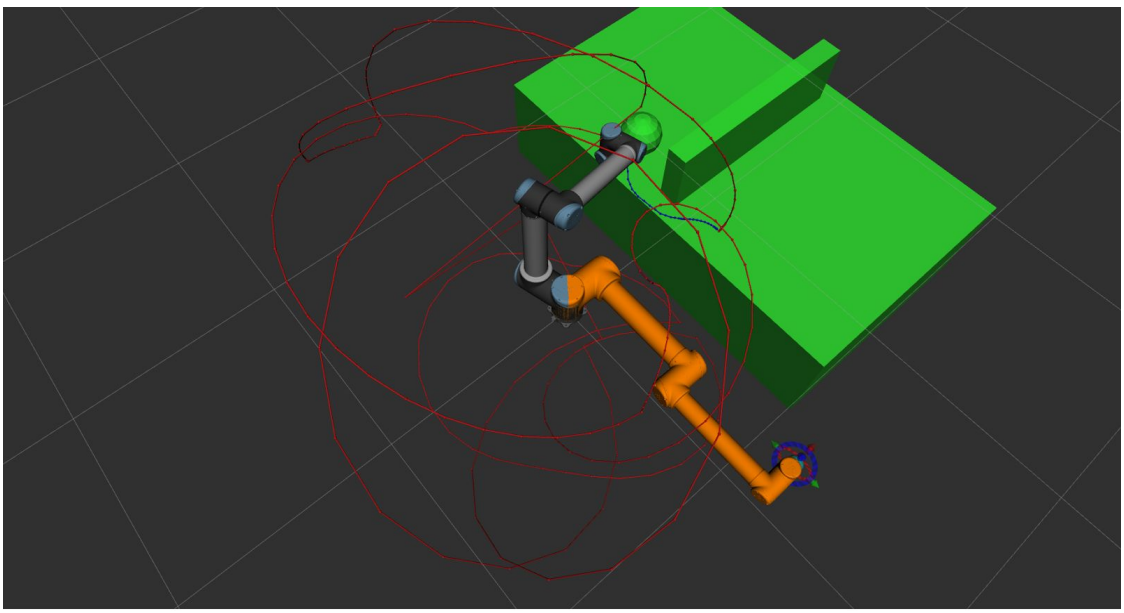

Figure 43: Result from KPIECE. The blue line is the shortest and the red line is the longest

Test 7	Planner	Total attempts	Success rate %	Mean Trajectory length [m]	Mean Trajectory smoothness [m^2/(s^5)]	Mean planning time [s]
From free space to the ball	LBKPIECE	1000	88.0	5.51	4215	1.190
From the ball to free space	LBKPIECE	1000	93.3	5.27	4722	1.137

table 7-2, LBKPIECE with trac_ik results of scenario 2

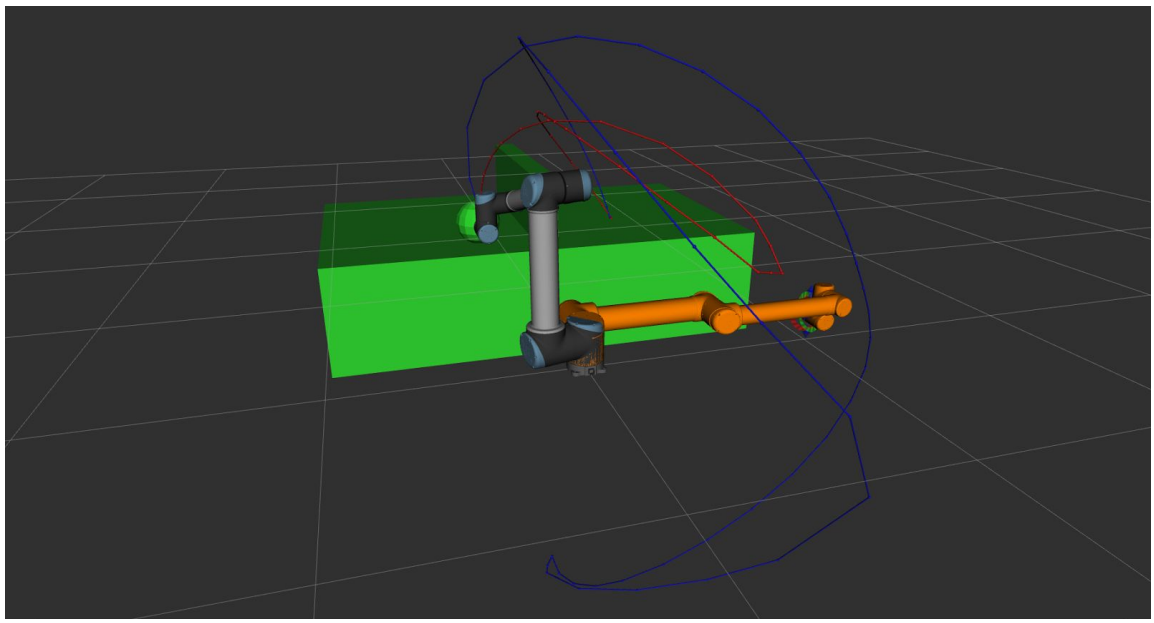

Figure 44: Result from LBKPIECE. The red line is the shortest and the blue line is the longest

Test 8	Planner	Total attempts	Success rate %	Mean Trajectory length [m]	Mean Trajectory smoothness [m^2/(s^5)]	Mean planning time [s]
From free space to the ball	RRTConnect	1000	99.9	4.37	4097	0.0487
From the ball to free space	RRTConnect	1000	100.0	4.07	4185	0.0436

table 7-3, RRTConnect with trac_ik results of scenario 2

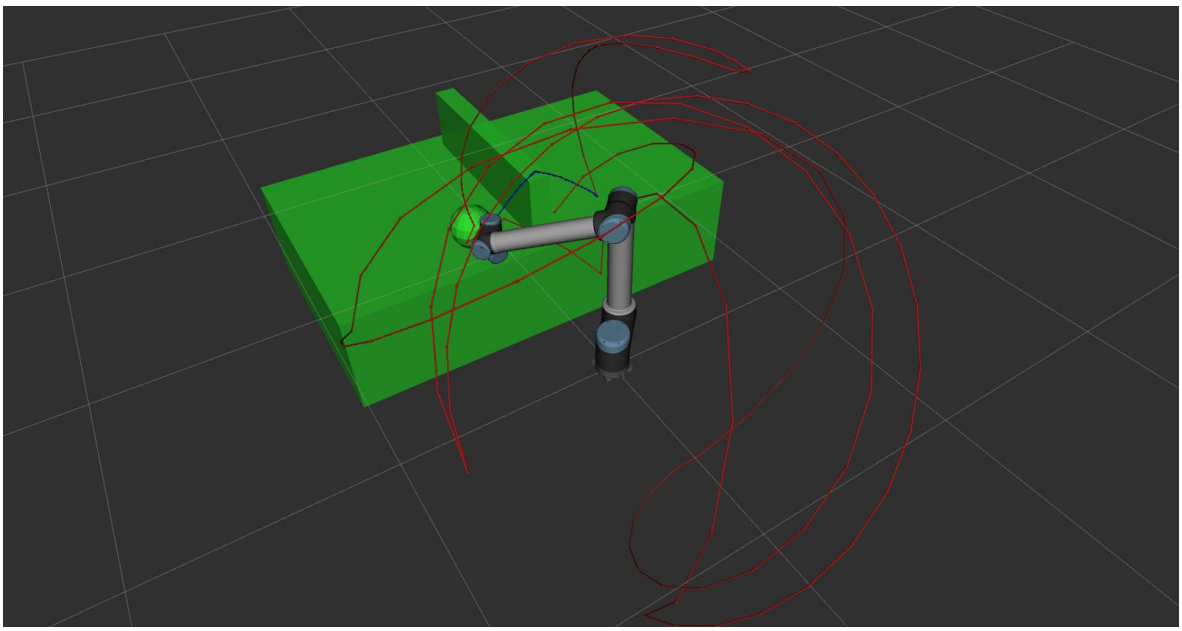

Figure 45: Result from RRTConnect. The blue line is the shortest and the red line is the longest

Test 9	Planner	Total attempts	Successful rate %	Mean Trajectory length [m]	Mean Trajectory smoothness [m^2/(s^5)]	Mean planning time [s]
From free space to the ball	PRM	1000	78.6	9.39	538	0.1*
From the ball to free space	PRM	1000	80.3	9.51	535	0.1*

table 7-4, PRM with trac_ik results of scenario 2

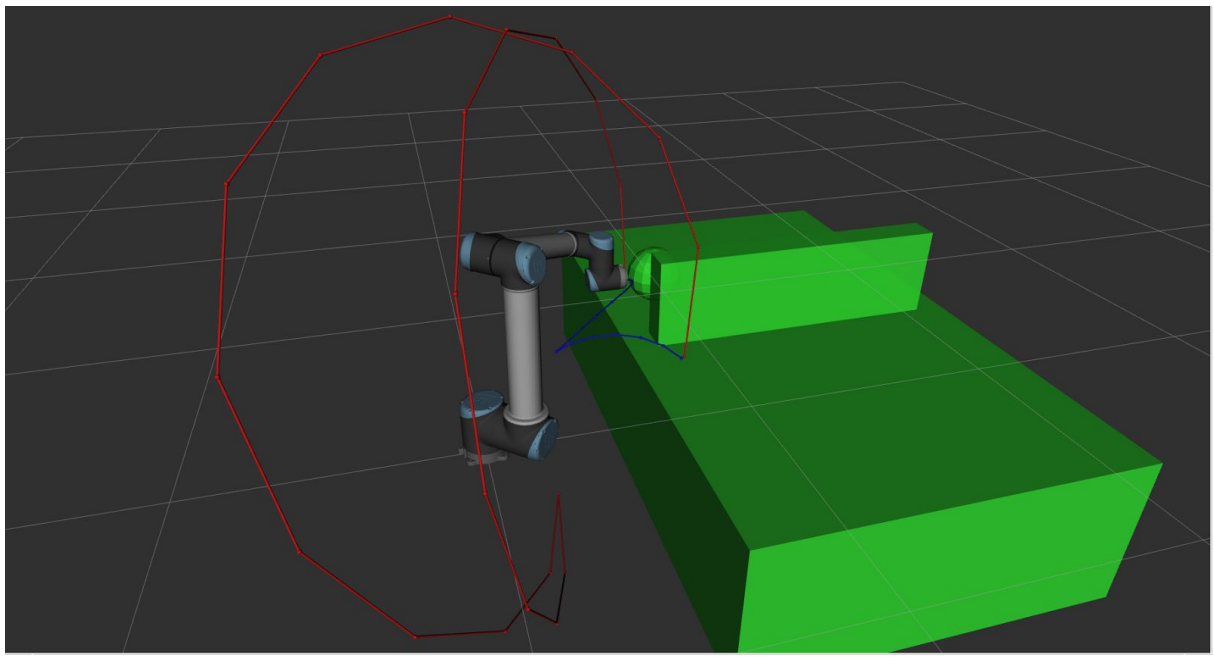

Figure 46: Result from PRM. The blue line is the shortest and the red line is the longest

Test 10	Planner	Total attempts	Mean Success rate %	Mean Trajectory length [m]	Mean Trajectory smoothness [m^2/(s^5)]	Mean planning time [s]
From free space to the ball	PRMstar	1000	67.3	11.06	602	0.1*
From the ball to free space	PRMstar	1000	80.8	10.54	558	0.1*

table 7-5, PRMstar with trac_ik results of scenario 2

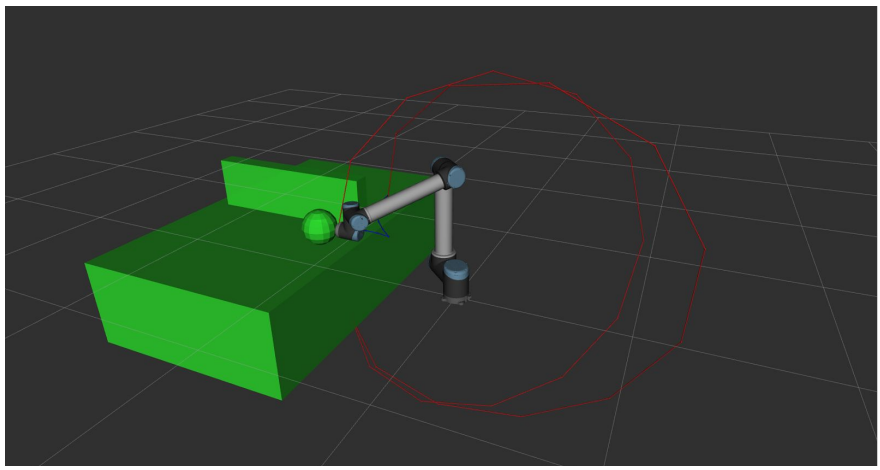

Figure 47: Result from PRMstar. The blue line is the shortest and the red line is the longest

* PRM and PRMstar are multi-query planners. They will run two processes at the same time. One is for roadmap and another is for the path search. They will use all the time that is given to them. Therefore, we set the maximum planning time as 0.1s for these two planners.

4.6.2.3 Discussion of the results

We could find that planning “from the ball to the free space” is easier than planning “from free space to ball”. This could be explained as follows:

We know that in multi-query sample based algorithms, we add new samples, including the goal state, to the search graph one by one and check for collisions. In OMPL, the goal is treated as a common state[5] which means it won’t be inserted into the graph frequently. Therefore the planner may not be able to sample a state that happens to be the goal and successfully find a collision free edge that link the goal state to a nearby state, in which case OMPL cannot find a trajectory to the goal state. Because free space around the ball is narrower, it’s harder to successfully find a collision free path to the ball compared with finding a path to free space.

In single query algorithms, RRT for example, the planner will grow a tree randomly from the start pose. And every new node of this tree is checked to see if it is the goal state. In OMPL, the checking is done by calling a function *isSatisfied()* which returns true if it is the goal. Because of the narrower free space, it’s less likely to sample the goal state.
That’s why we found the planning time is shorter and the success rate is higher in “from the ball to free space” case. This is an implementation issue of OMPL, one way to solve this problem is to give the goal state a higher weighting factor during sampling so that the goal state is frequently added to the graph and tested [11].

The two test sets that use different inverse kinematic solvers give very different results. The second set that use trac_ik perform better in almost all planners. We know that Moveit will only call inverse kinematic solver once to calculate joint space values of the target pose. Therefore, it’s strange to find that planners which use trac_ik outperform those using KDL so much. Through step by step test, we found that in this use case KDL tends to converge to a solution that is in contact with the table. And this problem gets worse when the goal is near the ball (There are collisions with the ball too). If there is contact, the inverse kinematic solver has to be called again and again until the solution is not in contact. Because of the relatively slow speed, KDL won’t leave too much time to the planner and the chance to find a path is low.

In the description of test setups, there is a term called the “longest_valid_segment_fraction”. This term influences the performance a lot. Once two nodes are connected, planners in OMPL will use a collision detector to check if there is any contact. This is done in a discrete way. The edge between two nodes is cut into small segments whose longest length is defined as “longest_valid_segment_fraction”. And these points instead of the edge will be checked. If this value is too large, the collision detector might ignore some small or narrow objects such as a corner of a table. However, if this value is too large, the collision checking will be too slow. There is a detailed test results and discussion at [7]. The default value which is set by universal robot is 0.05. This is too large. 0.005 is a better choice in our use case.

The RRT planner is as fast as RRTConnect if it could find a path. However, in this scenario where an obstacle is between the goal and start pose, RRT can’t grow its tree effectively. The original RRT will only grow its tree randomly from the start pose and since there is an obstacle in between, the chance of successfully develop the tree to the goal pose is low. This problem could be solved if we develop the tree in a smaller step size. However, this will cost much more time. RRTstar faces the same problem as RRT. Growing two trees as the RRTConnect is a more effective solution. RRTConnect performs best in our test.

PRM and PRMstar are multi-query planners, they will run two processes at the same time. One process is for build the roadmap and another one for searching for a collision free path. They are setted as using all the time that is allowed to refine the roadmap.[9] We set the planning time as 0.1s because we want to find out if they could outperform the RRTConnect. But the result is not very good. There are two main reasons for the bad performance. 1) 0.1 seconds is too short for PRM and PRMstar; 2) as mentioned in [9], the PRM will obtain good result when the obstacles spanning rather large intervals, which is not the case of our ball. Since we are trying to find a planner that is suitable for dynamic environment, the multi-query planners don’t have the advantage of roadmap. There is a noticeable advantage of PRM and PRMstar - they return a much smoother trajectory. This could be contribute to the continuous refinement of the roadmap.

From the test result, we can find an interesting feature. Although some planners have a low successful rate, their average planning time is only about a half of the maximum planning time. It seems all the successful attempts find a trajectory very fast. We think this feature

indicates that if a sample based planner can't sample the right states which give a trajectory, it will take a lot more time to find a trajectory even if the algorithm is probabilistic complete.

4.6.3. Shelf scenario

4.6.3.1 Test group 1

Set up:

IK solver	Longest_valid_segment_fraction	Planner tolerance/[m]	Maxim planning time/[s]
trac_ik	0.005	0.01	2.0

OS distro	ROS distro	CPU	RAM
Ubuntu 16.04	kinetic	intel 6700hq	8GB

Results:

Test 1	Planner	Total attempts	Successful rate/%	Trajectory length/[m]	Trajectory smoothness	planning time/[s]
	KPIECE	200	54.0	7.89	3897	4.508

table 8-1, KPIECE with trac_ik results of scenario 3

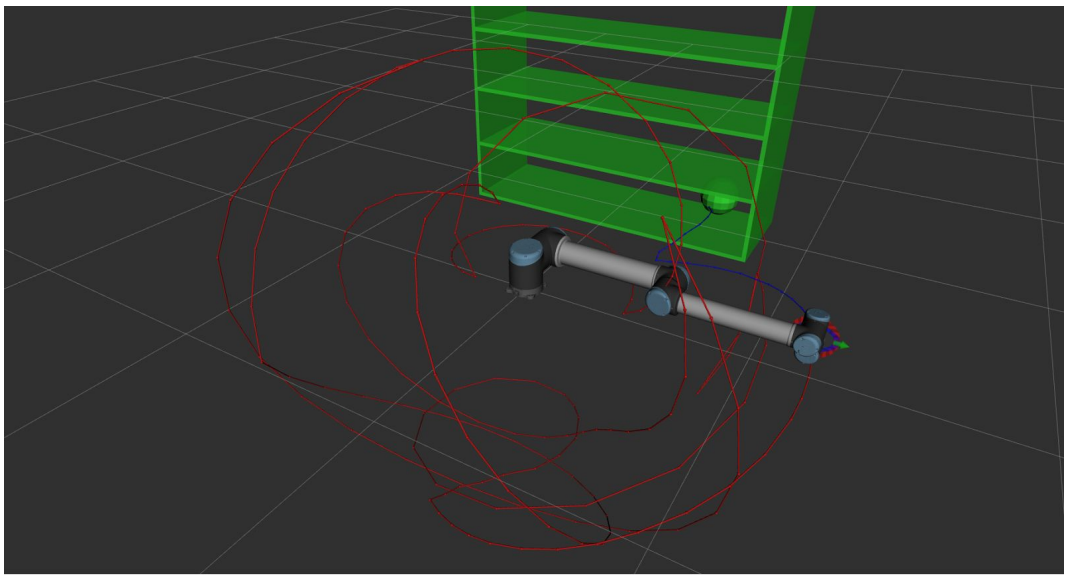

Figure 48: Result from KPIECE. The blue line is the shortest and the red line is the longest

Test 2	Planner	Total attempts	Successful rate/%	Trajectory length/[m]	Trajectory smoothness	planning time/[s]
	LBKPIECE	200	62.0	7.02	4071	1.189

table 8-2, LBKPIECE with trac_ik results of scenario 3

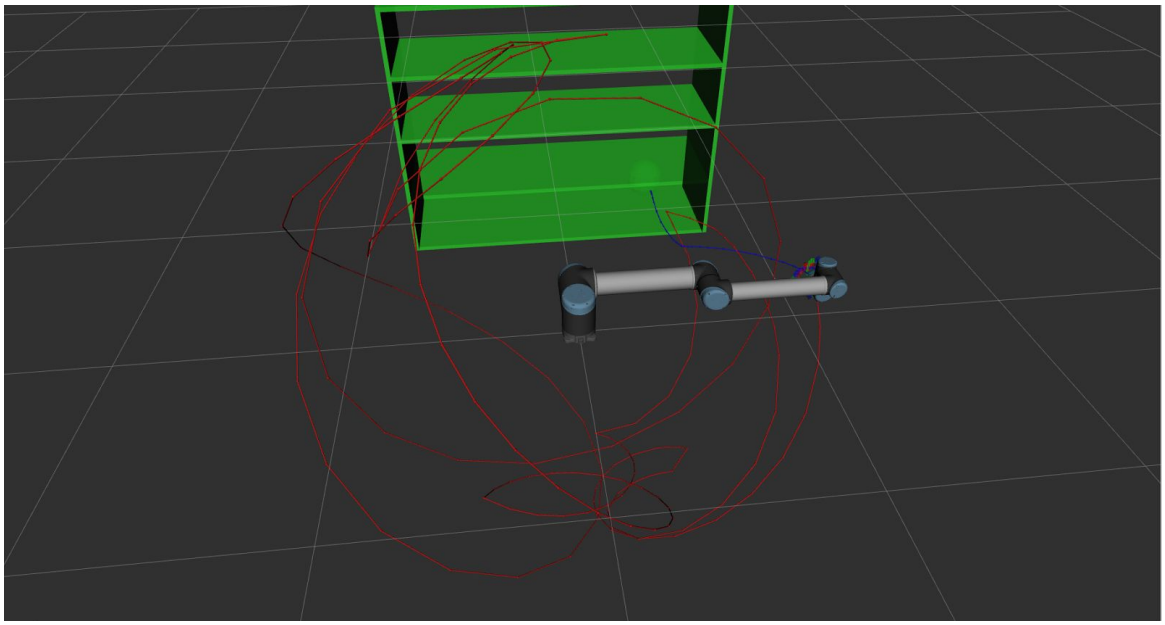

Figure 49: Result from LBKPIECE. The blue line is the shortest and the red line is the longest

Test 3	Planner	Total attempts	Successful rate/%	Trajectory length/[m]	Trajectory smoothness	planning time/[s]
	RRTConnect	200	89.5	7.56	3850	1.788

table 8-3, RRTConnect with trac_ik results of scenario 3

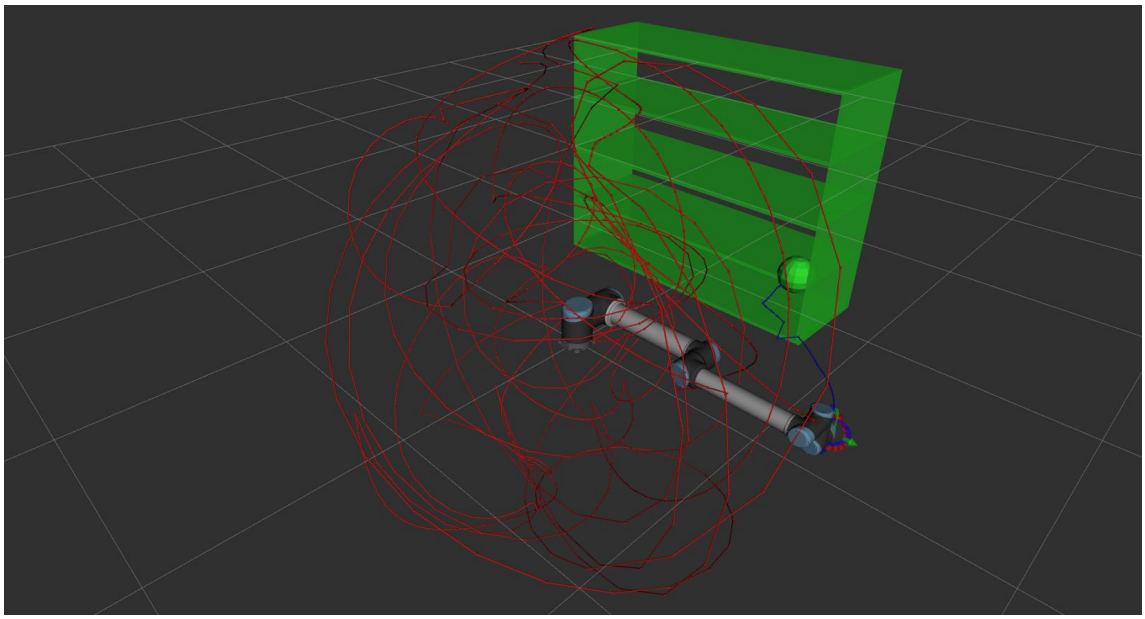

Figure 50: Result from RRTConnect. The blue line is the shortest and the red line is the longest

Test 4	Planner	Total attempts	Successful rate/%	Trajectory length/[m]	Trajectory smoothness	planning time/[s]
	PRM	200	47.0	8.07	1127	5.0*

table 8-4, PRM with trac_ik results of scenario 3

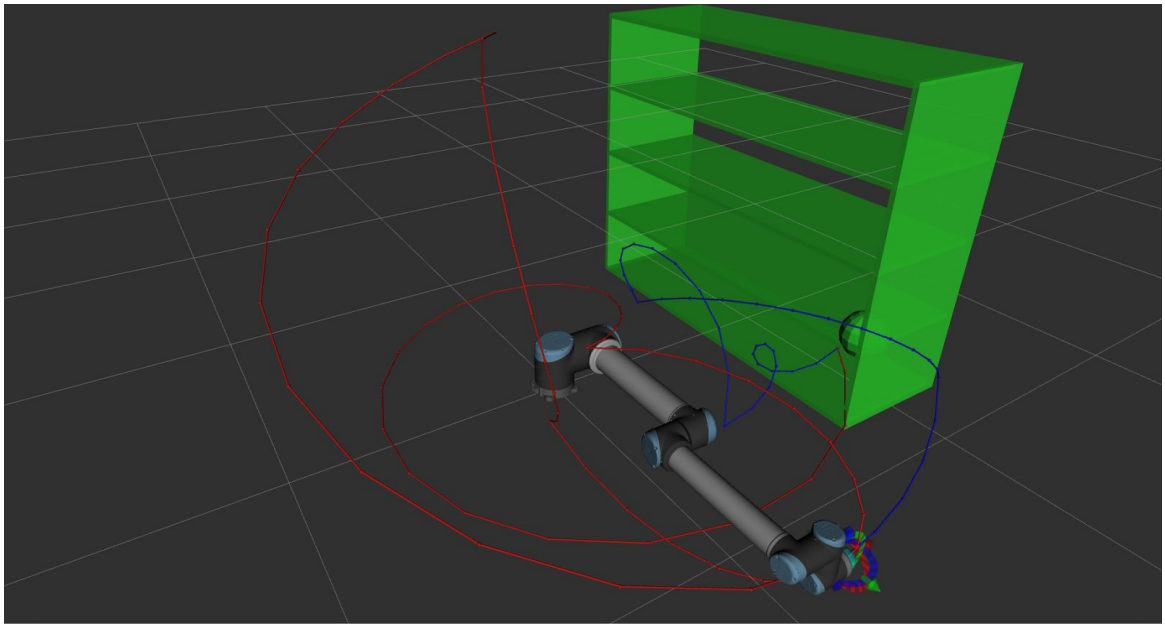

Figure 51: Result from PRM. The blue line is the shortest and the red line is the longest

Test 5	Planner	Total attempts	Successful rate/%	Trajectory length/[m]	Trajectory smoothness	planning time/[s]
	PRMstar	200	47.5	7.15	1019	5.0*

table 8-5, PRMstar with trac_ik results of scenario 3

* The planning time for multi-query is setted manually.

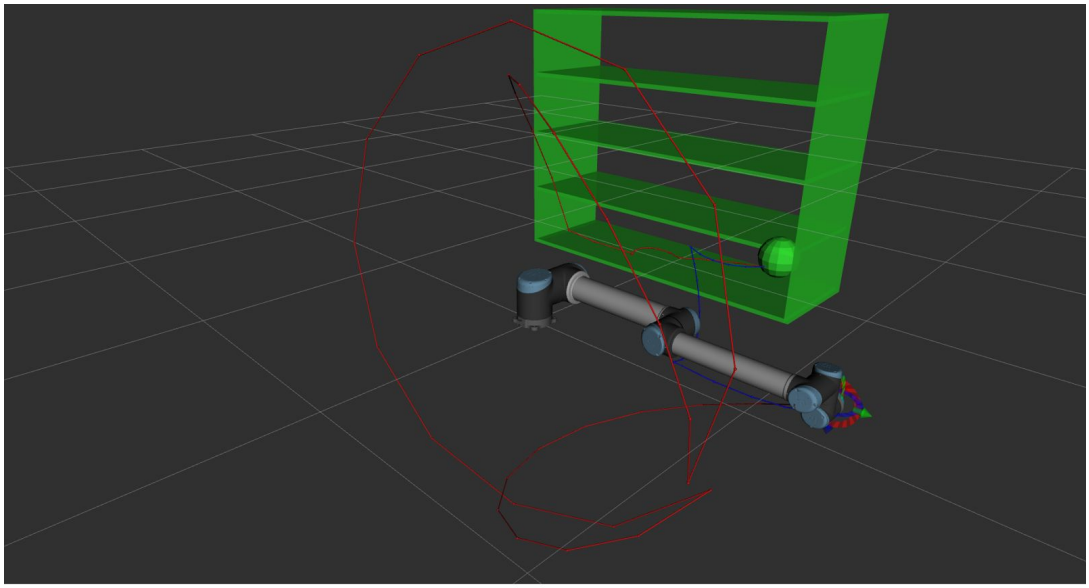

Figure 52: Result from PRMstar. The blue line is the shortest and the red line is the longest

4.6.3.2 Test group 2

In this test, we will check how the success rate changes with maximum planning time.

Set up:

IK solver	Longest_valid_segment_fraction	Planner tolerance/[m]	Planner	Number of attempt for different maximum planning time
trac_ik	0.005	0.01	RRTConnect	200

OS distro	ROS distro	CPU	RAM
Ubuntu 16.04	kinetic	intel 6700hq	8GB

Result:

Max planning time [s]	success rate [%]	mean trajectory length [m]	mean trajectory smoothness	mean planning time [s]
0.5	28.5	9.24	3622	0.284
1.0	43.0	7.01	3502	0.453
2.0	63.5	7.58	3741	0.647
5.0	82.0	7.29	3833	1.255
10.0	86.5	6.98	3385	1.512
20.0	87.5	7.19	3944	1.694
30.0	90.0	7.24	3781	1.999
50.0	91.0	7.42	3575	2.132

table 9, RRTConnect with trac_ik results of scenario 3

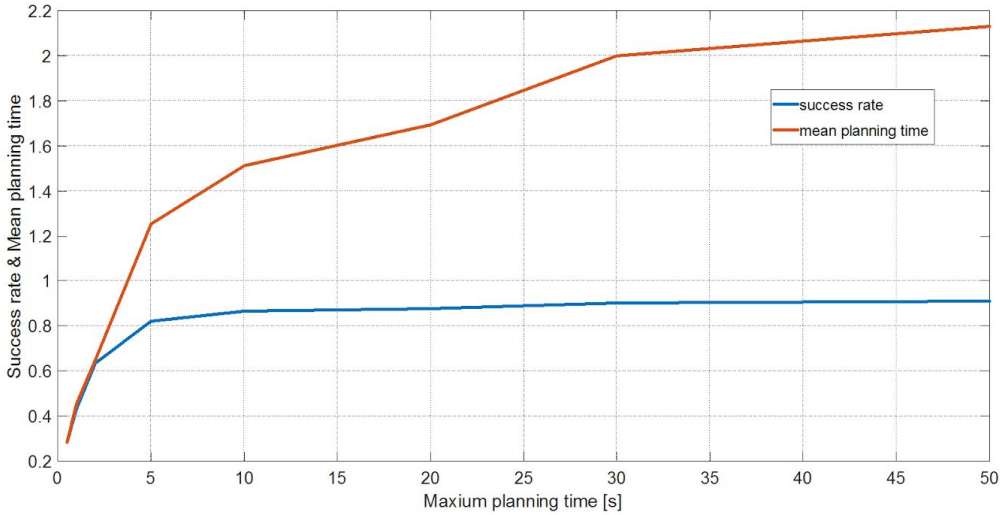

Figure 53: Success rate and mean planning time change with maximum planning time, we can find the success rate increase slowly after 5 seconds.

4.6.3.3 Test group 3

In this test, we try to plan 1000 times with maximum planning time setted as 50.0s. The goal is to find the distribution of planning time and trajectory length.

Set up:

IK solver	Longest_valid_segment_fraction	Planner tolerance [m]	Maxmuim planning time [s]	Number of attempts
trac_ik	0.005	0.01	50.0	1000

OS distro	ROS distro	CPU	RAM
Ubuntu 16.04	kinetic	intel 6700hq	8GB

Result:

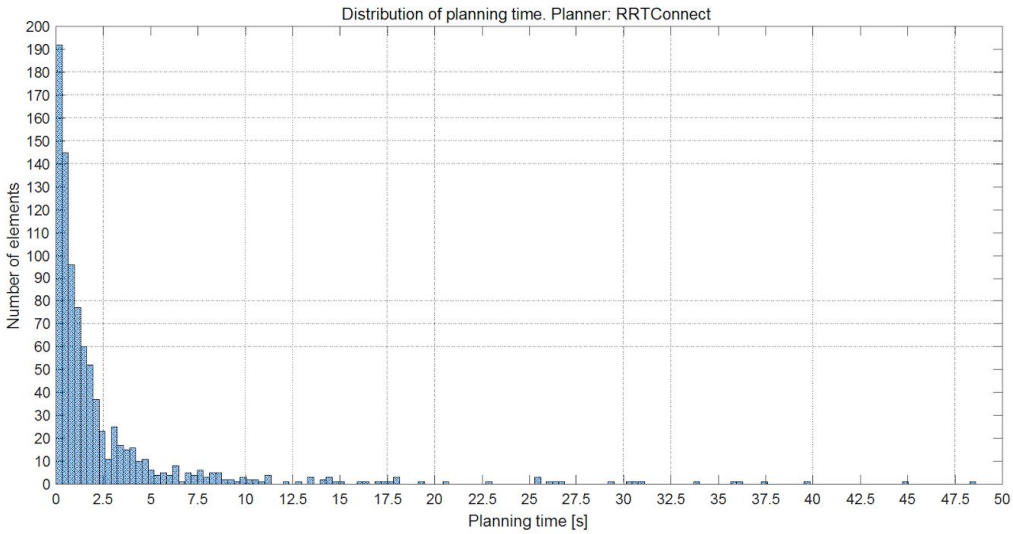

Figure 54: Distribution of planning time. Most successful attempts end with planning time that is shorter than 5 seconds

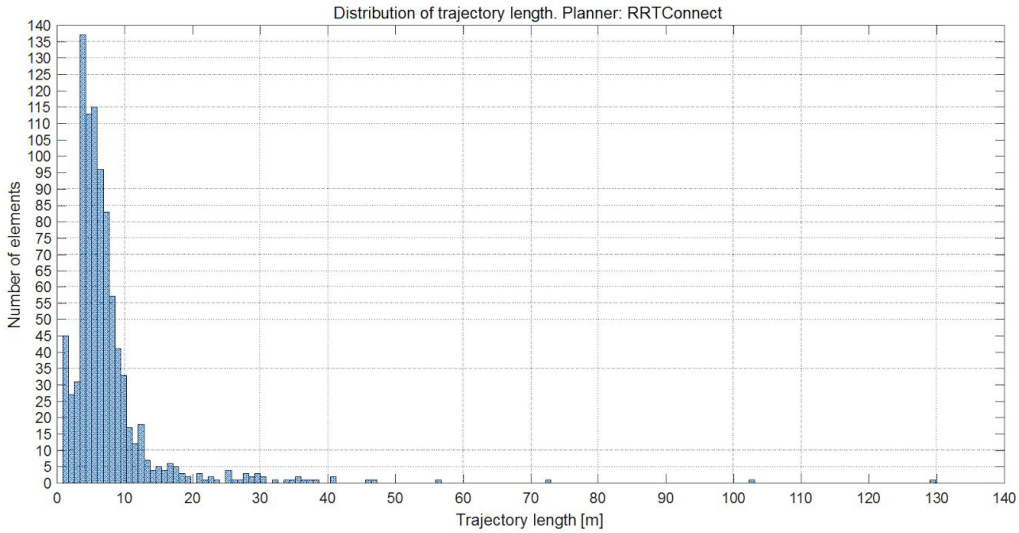

Figure 55: Distribution of trajectory length

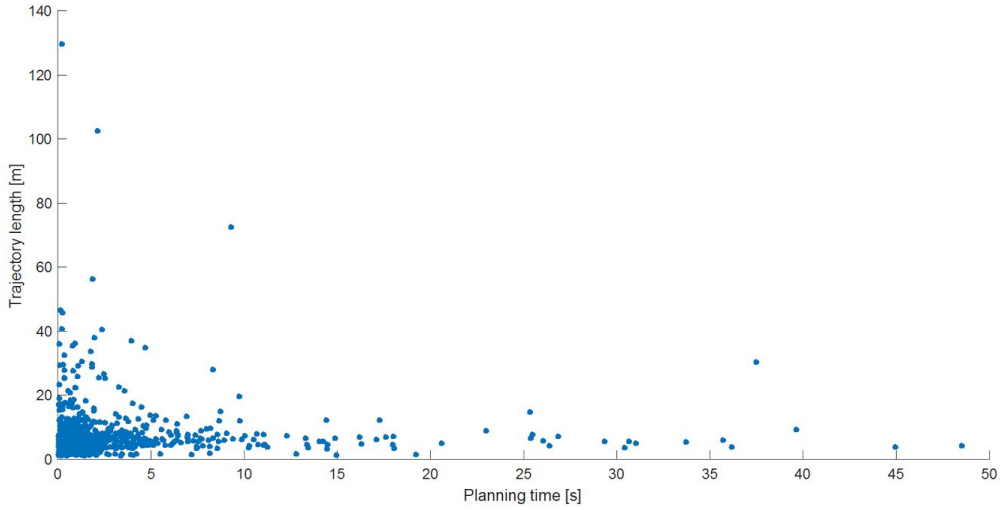

Figure 56: Relationship between planning time and trajectory length

4.6.3.4 Discussion of the results

In this more complex planning scene, all planners have a lower success rate. The planner that has the highest success rate is the RRTConnect. However, as we can see from the pictures above, all planners will sometimes end up with some crazy trajectory and RRTConnect gives the craziest one.

From the results of the second and the third test group, we can find most successful attempts (87.8%) is finished within 5 seconds. And we don't find a strong relationship between planning time and trajectory length.

Combining these features, we suggest to choose the RRTConnect as the default planner in complex environment. We should put the planning attempt inside a while loop which stop after an acceptable trajectory is found. The maximum planning time should be set at 5~10 seconds. The exact value depends on the environment and need some tuning. To prevent crazy trajectory, we need to check the quality of the trajectory and only the qualified trajectory should be accepted.

4.6.4. Inappropriate settings that need to be avoided:

4.6.4.1. Tolerance of planner and inverse kinematic solver.

Tolerance is defined as the norm of the error vector. The tolerance of a planner should be larger than the inverse kinematic solver. Otherwise, the joint space values calculated by a inverse kinematic solver might always cause slightly collision. And the plan generated by the planner couldn't pass the collision checking in this situation.

4.6.4.2 Goal pose is in collision with other objects

Most planners will plan in joint space. When a goal pose (in work space) is set, planners will call IK solvers to calculate a joint space value. However, the inverse kinematic solution is not unique. For a given goal pose, there are multiple IK solutions and it's hard to determine which one will be returned by IK solver. If the IK solver returns a set of joint values that corresponding to a state in collision (The end effector is collision free, but the rest of the arm may have contact with other obstacles) the planner may never give a result.

If we are using the Moveit planning pipeline, the planning request will be checked. When move group found collision at the goal pose, it will call the planning request adapter who will try to change the goal pose through perturbing the joint values by a small amount. If we are sure there is a collision free configuration at the goal pose of end effector, we can manually call the IK solver, check the IK solution and give the move group a collision free joint space goal. The disadvantage of this approach is that we don't know how to make the IK solver converge to another solution. Changing the initial value is one way to do it. But we don't know which direction to change. And I'm not sure if there is a function in trac_ik or KDL that allow us to change the initial value.

4.6.4.3 Monitoring planning scene

In moveit, the planning scene is maintained in another process (the process that runs the move group node), not in our own program. When we create a planning scene object, it's only a snapshot of the real planning scene and it will remain the same even if the real planning scene has been changed. More specifically, when we add a collision object to the planning scene, we publish a message so that move group knows there is a new object. But our own planning scene object is not updated. We have to update the planning scene manually. This is not a problem if we just want to use functions that are provided by the *MoveGroupInterface* class. (These functions use ROS service and/or ROS action, they will send a message to the move group node and all the calculation is done by move group. Move group knows all the change. Our own planning scene doesn't know.) But if we want to use functions that are provided by the *PlanningScene* class (for example, check if a joint configuration is in contact with obstacles using *isStateColliding()*), it will give a wrong answer. There is a member function in *PlanningSceneMonitor* class that returns a planning scene pointer and keeps tracking the planning scene. However, it's not safe and the moveit suggests to avoid it.

5. Conclusion:

5.1. Suggestion on planner selection

From the test result above, we give the following suggestion about planner selection:

1.) In an environment that changes slowly and no narrow free space, a multi-query planner such as PRM or PRMstar could be used. In the implementation, a monitor should be introduced. Its job is to check if the current roadmap contains collision nodes or edges. And a configurator is needed. It should decide what to do with the collision states: to delete them from the roadmap or to rebuild the whole roadmap. We can take the advantage of the smooth trajectory while have an acceptable success rate.
2.) In an environment that changes fast or is narrow, we suggest to use RRTConnect. In the implementation, we need to check the quality of the trajectory to prevent the crazy trajectory.
3.) Even in a simple point to point task with on obstacles, interpolation in Cartesian space is not recommended.
4.) We can't recommend CHOMP or SBPL because 1) at least in our test, they are no better than sample based planners; 2) they are not integrated into ROS very well. Extra development time is required.

5.2. Suggestion on parameter tuning

1.) Maximum planning time. This parameter highly depends on the environment. Therefore, we suggest to create a configurator who keeps the memory about past planning attempts. From the result of 4.6.3.3, the configurator should set the maximum planning time to the value that give a success rate about 80%. The size of the memory, in other words how many past attempts to remember, depends on how fast the environment will change and the frequency of planning requests.
2.) Longest valid segment fraction. This factor depends on the shape of environment. In an environment with many thin object and shape edge, this value should be small so that no collision is ignored. In other cases, this value could be larger, so that we have a faster planner.
3.) Tolerance. The tolerance of planner is decided by the use case. But the tolerance of the IK solver must be smaller than the planner tolerance.

5.3. Future work

The implementation suggestion in previous sections are based on the test results. They haven't been verified yet. More test is needed about these suggestions. There are some problem needed to be solved in the implement. For example, how can the planner know the

quality of a trajectory? In a fixed planning scene, this could be done through compare the trajectory with the past record. If the length, smoothness or other metrics is significantly worse than the record, it might be a bad trajectory. However, in a changing environment or a new environment, this could be a challenge.

We also noticed that even the fastest planner we tested, the RRTConnect, is not fast enough. We need to test more planners to see if there is a faster one.

References

[1] P. Beeson, and B. Ames, “TRAC-IK: An Open-Source Library for Improved Solving of Generic Inverse Kinematics,” In *Proceedings of the IEEE RAS Humanoids Conference*, Seoul, Korea, November 2015.

[2] The Open Motion Planning Library, 2017, viewed 23 May 2017, <http://ompl.kavrakilab.org/>

[3] Kavraki Lab, and Rice University, “Open Motion Planning Library: A Primer”, publisher “Rice”, USA, May 2017.

[4] I. A. Şucan, and L. E. Kavraki, “Kinodynamic Motion Planning by Interior-Exterior Cell Exploration”, in "Algorithmic Foundation of Robotics VIII: Selected Contributions of the Eight International Workshop on the Algorithmic Foundations of Robotics", publisher “Springer Berlin Heidelberg", ISBN “978-3-642-00312-7”, Berlin, Germany, 2010.

[5] I.A. Sucan, M. Moll, and L.E. Kavraki. The open motion planning library. Robotics & Automation Magazine, IEEE, 19(4):72–82, 2012.

[6] M. Kalakrishnan, S. Chitta, E. Theodorou, P. Pastor, and S. Schaal. STOMP: Stochastic trajectory optimization for motion planning. In Robotics and Automation (ICRA), 2011 IEEE International Conference on, pages 4569– 4574. IEEE, 2011.

[7] davetcoleman, *Increase collision checking interval #337*, github, USA, viewed 23 May 2017, <https://github.com/ros-planning/moveit/pull/337>

[8] Flash T, Hogan N. The Coordination Of Arm Movements - An Experimentally Confirmed Mathematical-Model. Journal Of Neuroscience. 1985;5(7):1688–1703.

[9] L.E. Kavraki, P.Švestka, J.-C. Latombe, and M.H. Overmars, Probabilistic roadmaps for path planning in high-dimensional configuration spaces, IEEE Trans. on Robotics and Automation, vol. 12, pp. 566–580, Aug. 1996. DOI: 10.1109/70.508439

[10] Wikipedia contributors, 'Rapidly-exploring random tree', *Wikipedia, The Free Encyclopedia,* 10 April 2017, 13:42 UTC, <https://en.wikipedia.org/w/index.php?title=Rapidly-exploring_random_tree&oldid=774753516> [accessed 23 May 2017]

[11] Kavraki, L. E., & LaValle, S. M. (2008). Handbook of Robotics, Incollection Motion Planning. Berlin/Heidelberg: Springer. doi:10.1007/978-3-540-30301-5_6.

[12] D.Thomas 2014, *Introduction,* viewed 23 May 2017, <http://wiki.ros.org/ROS/Introduction>

[13] E. Fernández (2015). “Learning ROS for Robotics Programming”,

[14] AaronMR 2014, *Concepts,* viewed 23 May 2017, <http://wiki.ros.org/ROS/Concepts>

[15] *MoveIt! Motion Planning Framework* 2017, viewed 23 May 2017, <http://moveit.ros.org/>

[16] *Concepts,* viewed 23 May 2017, <http://moveit.ros.org/documentation/concepts/>

[17] *Tutorial: Using a URDF in Gazebo*, viewed 23 May 2017, <http://gazebosim.org/tutorials/?tut=ros_urdf>

[18] *Semantic Robot Description Format (SRDF) review*, viewed 23 May 2017, <http://wiki.ros.org/srdf/review>

[19] R. Bohlin and L. E. Kavraki, “Path planning using lazy PRM,” in Proc. 2000 IEEE Int. Conf. Robotics Automation, San Francisco, CA, 2000, pp. 521–528.

[20] *Rviz*, viewed 23 May 2017, <http://sdk.rethinkrobotics.com/wiki/Rviz>

[21] *KDL wiki*, viewed 23 May 2017, <http://www.orocos.org/wiki/orocos/kdl-wiki>