Restore-L MoveIt! Documentation

Tariq Zahroof
Stanford University
Stanford, CA 94305
tzahroof@stanford.edu

Abstract

The Restore-L Mission by NASA seeks to service old satellites using a highly equipped, robotic servicing spacecraft. The Restore-L Spacecraft will, during its mission, rendezvous with, grasp, service and refuel many government-owned satellites. This project developed a motion planning tool to safely autonomously navigate a robotic manipulator to a tooling location while avoiding environmental obstacles and clutter. Here, we analyze the effects of the BFMT+Shortcut technique to navigate the robotic arm. This paper gives documentation for use and further development of the MoveIt!-based tools devised by ASL.*

1. Instructions

1.1. Installation

The following instructions should be run from the terminal.

1. Install ROS Kinetic:

   `http://wiki.ros.org/kinetic/Installation/Ubuntu`

2. Install catkin tools for Kinetic:

   `http://catkin-tools.readthedocs.io/en/latest/installing.html`

   (a) Set up a catkin workspace, but use **catkin build** instead of **catkin_make**:

   `http://wiki.ros.org/catkin/Tutorials/create_a_workspace`

   i. **catkin build** is a more convenient tool for building catkin packages. It lists the build process in a more readable fashion, and also lists the build errors as well. To switch tools from **catkin_make** to **catkin build** or vice-versa, the devel and source files need to be deleted before the workspace can be recompiled.

3. Install MoveIt! Kinetic from Source (and the prerequisites):

   `http://moveit.ros.org/install/source/`

   (a) Clone the up-to-date OMPL package (`https://github.com/ompl/ompl`) into the source directory: `~/catkin_ws/src`.

   (b) Apply the following commit changes to the `moveit` package from: `https://github.com/ros-planning/moveit/pull/903/commits/d6c095b3be30b3c3312079ef173a7686320463c8`. The commit will change `moveit_planners/ompl/ompl_interface`.

   (c) Build the directory by running **catkin build** from the catkin workspace directory head (i.e. `~/catkin_ws`.

4. Install the MoveIt! tutorials (for reference) and the FRANCA arm (`panda_moveit_config`):

   `http://docs.ros.org/kinetic/api/moveit_tutorials/html/doc/getting_started/getting_started.html`

   (a) Scroll down to the "Download the Example Code" section and follow the rest of the tutorial.

5. Install ROS-industrial core package (this is due to the dependencies of the motoman package). Clone `https://github.com/ros-industrial/industrial_core` into `~/catkin_ws/src`

   (a) Build the directory with **catkin build** in `~/catkin_ws`.

6. Install the ROS-industrial MoveIt! package. Clone `https://github.com/ros-industrial/industrial_moveit` into the `~/catkin_ws/src` folder.

(a) Delete the `industrial_collision_detection`, `constraint_ik`, and `industrial_moveit_benchmarking` folders within the package (`~/catkin_ws/src/industrial_moveit`) to allow the package to build (these folders are antiquated).

(b) Build the directory. (Run **catkin build** from `~/catkin_ws`).

7. Install the motoman package. Clone `https://github.com/tzahroof/motoman` into the `~/catkin_ws/src` folder, and then run **catkin build** from the `~/catkin_ws`.

8. For optional usage and/or perusal, the `restorebot_moveit_desc` and `restorebot_desc` packages can be installed to see the prototype mock-ups. They can be installed from `https://github.com/tzahroof/restorebot_moveit_desc` and `https://github.com/tzahroof/restorebot_desc`, respectively. The `restorebot_moveit_desc` package depends on the `restorebot_desc` package.

Important Note: the catkin file system has issues if existing, already-built folders are replaced with similarly named folders. As such, if you build the directory and then replace a folder with a same-named folder, then you might have to clean the `src` or `devel` folders by invoking the **catkin clean** command in the head of the workspace (i.e. `catkin_ws`). This might, for example, fix any issues of building before the `OMPL` package is replaced. Also, REMEMBER TO SOURCE THE FILES by running the command: **source `~/catkin_ws/devel/setup.bash`**.

1.2. Tutorials

The following tutorial is necessary to understand what the code is doing:

1. MoveIt! Motion Planning API tutorial:

   `http://docs.ros.org/kinetic/api/moveit_tutorials/html/doc/motion_planning_api/motion_planning_api_tutorial.html`

The following links are helpful for understanding the MoveIt! concepts and API:

1. MoveIt! concepts:

   `http://moveit.ros.org/documentation/concepts/`

2. MoveIt! API:

   `http://docs.ros.org/kinetic/api/moveit_core/html/annotated.html`

2. Running the Code

There are four example codes demonstrating the tools developed by the lab. **BFMT*+Shortcut** is the main tool, and is launched by the first command. The next two, **BFMT*+STOMP** and **BFMT*** are alternative tools to consider. The final example code simulates a pick and place procedure using the BFMT*+Shortcut.

These following codes are launched by:

1. `fmt_shortcut`:

```
$ roslaunch
    motoman_sia20d_moveit_updated
     fmt_shortcut.launch
```

2. `fmt_stomp`:

```
$ roslaunch
    motoman_sia20d_moveit_updated
     fmt_stomp.launch
```

3. `fmt` (launch in order and in separate terminals):

```
$ roslaunch
    motoman_sia20d_moveit_updated
     demo.launch

$ roslaunch
    motoman_sia20d_moveit_updated
     fmt.launch
```

4. `pick_place`:

```
$ roslaunch
    motoman_sia20d_moveit_with_gripper
     pick_place.launch
```

3. Fundamentals

3.1. ROS

ROS is a publisher/subscription service that helps processes communicate. MoveIt! is built on top of ROS, and primarily uses the system for RViz visualization and terminal feedback. However, actual robots, sensors, and other **nodes** can be be implemented with the MoveIt! framework for varied use-cases. A simple analogy for the service is a message board. Publishers post messages to specific topics, while subscribers read the information from the topic.

In its current state, the tool only uses publishers to relay information to the visualization tool. The most important topics are the `/move_group/display_planned_path` and the `/motoman/planning_scene` topics. The RViz simulation needs to be set up to receive planning scene

information (the robot, its state, and the objects in the environment) from `/motoman/planning_scene` and to receive the generated motion plan trajectory from `/move_group/display_planned_path`. Some of the code provided might use slightly different naming conventions for the planning scene and trajectory topic, but are otherwise the same besides semantically.

To test if a ros topic is active, open up a separate terminal window and type `rostopic echo <insert topic name here>`.

Currently, MoveIt! is most strongly supported by ROS Kinetic, and as such the packages have default compatability with ROS Kinetic. Kinetic can be installed onto Ubuntu 16.04 computers.

3.2. RViz

RViz needs to be configured to properly display the robot and obstacles. RViz is NOT a simulator; it simply displays the transformations and motions planned by the MoveIt! code (i.e. there are no physics unless they are manually programmed). For actual simulation, consider looking into **Gazebo**.

When RViz starts up for the first time, it will probably have a blank screen. In the **Displays** tab (usually the left panel; if it does not exist, add it by going to the top bar and clicking `Panels -> Add New Panel -> Displays`. First, set the fixed frame to be that of the base link of the robot. This is determined from the robot's URDF. For the Motoman SIA20d robot, for example, the base link is eloquently referred to as `base_link`. Additionally, one can use a virtual "world" frame by creating a virtual fixed joint (all transformations of the robot joints and links will be done to this main joint). This can be done by modifying the SRDF file.

If MoveIt! installed correctly, **MotionPlanning, PlanningScene, RobotState, and Trajectory** should be available in the window that popped up after the `Add` button was pressed in the left panel. Add the MotionPlanning and Trajectory options. Ensure that the "Planning Scene Topic" is linked to `/motoman/planning_scene` under MotionPlanning. "Trajectory Topic" under Trajectory should be linked to `/move_group/display_planned_path`. Trajectory topic is not particularly needed, but it does have the neat option to loop the generated trajectory animation and to also leave an animation trail. To display text from the MoveIt! code, add **MarkerArray** to the Displays tab and set its Marker topic to `/rviz_visual_tools`. Finally add the **RvizVisualToolsGui** by `Panels -> Add New Panel -> RvizVisualToolsGui`. This allows for button presses to step through the code.

The robot world and scene can be moved around by clicking the **"Move Camera"** option at the top bar below the drop-down bar (if said option does not exist, it can be added by the plus sign). Left-click + drag rotates the camera, while Right-click + drag zooms in or out of the scene. The scene can be interacted with the interact option. The **"Select"** option provides the x,y,z,r,y,p information about the robot, provided that the **Selection Panel** has been added to the GUI.

The above provided a quick overlook of the RViz environment. Further information can be found in this tutorial: `http://docs.ros.org/kinetic/api/moveit_tutorials/html/doc/quickstart_in_rviz/quickstart_in_rviz_tutorial.html`.

3.3. The Catkin File System

The reader should have followed the catkin tutorial to create a proper catkin workspace. Additionally, the reader should have installed the catkin pip tools.

ROS uses the catkin system to provide easy file navigation and node launching. A specific folder is designated the workspace folder, and all ROS packages should be placed in its **src** folder. Everytime a significant change happens to a package (code changed, new package added, new dependencies added, etc), the package must be built again from the head of the workspace folder to have the system acknowledge these changes. For example, if one were to create their workspace folder in `~/catkin_ws`, one would navigate to `~/catkin_ws` and use the command **catkin build** (the python command) in the terminal to build the changes. Any problems during the build process will be reported here. Afterwards, to have the changes be recognized by the terminal, the user has to source the `setup.bash` file in the devel folder. Using the previous example, the user would navigate to `~/catkin_ws/devel` and input the command **source setup.bash**. Packages would be added to the workspace in the `~/catkin_ws/src` folder.

System or important packages are often saved in the `/opt/ros/kinetic` folder. This makes it harder to make non-sanctioned changes to these files. Files are usually installed here when installing user (not developer) packages.

A quick tip would be to edit the `.bashrc` file to automatically source the setup file whenever the terminal is opened (subsequent changes during the terminal session need to be re-sourced). Simply add the command (using the given example): `source ~/catkin_ws/devel/setup.bash`.

With the Catkin File System in place, we now have access to two following, important commands: **roscd** and **roslaunch**. `roscd` is similar to entering `cd` into the terminal; it navigates the user to the package from whichever directory they were located. For example, to navigate to the motoman package, type in `roscd motoman`.

The user can also create **launch** files, which are essentially text documents that detail (and ef-

fectively launch) ROS nodes. These files can be launched easily by `roslaunch`. For example, to launch the `pick_place.launch` code from the Motoman package, simply type `roslaunch motoman_sia20d_with_gripper_moveit pick_place.launch`.

The catkin system can be fairly daunting to make changes, but luckily almost all of the proper dependencies and setup have already been made. The two key files to change are the `CMakeLists.txt` and `package.xml` files. To summarize, `CMakeLists.txt`indicates where the package's files and subdirectories are located, and additionally indicates which other packages/files our package needs. `package.xml` indicates which packages are needed to build and run the package. More information can be found here, `http://wiki.ros.org/catkin/CMakeLists.txt`, and here, `http://wiki.ros.org/catkin/package.xml`. **C++** files are not default acknowledged by ROS to be nodes; they need to be added to the `CMakeLists.txt` file (and thus rebuilt every time the code is changed). Luckily, text files (such as .launch, .yaml, etc) do not need to be added.

For more information, please look here: `http://wiki.ros.org/catkin/Tutorials`

4. MoveIt! files, URDFs, and SRDFs

Robots are created in text files called **URDF** files. (Unified Robot Description Format). These files use links and joints to represent a robot. Joints link two links together, and are the main thing controlled by the MoveIt! code. Links can be represented by shapes or STL files. Links are located and oriented in perspective of their parent link. URDFs include information like joint types, joint limits, link/joint proximity location, effort, etc. Xacro is a special tool that has the ability to create URDF files. Xacro files are very similar to URDFs, except that Xacros can use macro commands and math within the file. URDF files can be generated from Xacro files by `rosrun xacro xacro --inorder -o model.urdf model.urdf.xacro`. Packages created by the MoveIt! setup assistant will read the xacro file (thus generating a URDF each time the launch file is executed). Xacros can also be used to combine other Xacro or URDF files to create bigger robots. For example, the `motoman_sia20d_with_gripper_moveit` package's Xacro file combines the Motoman robot Xacro file and the FRANKA arm's gripper.

SRDF (Semantic Robot Description Format) files are files that extend/provide more information about the robot urdf. These file establishes self-collisions, joint groups (controlling the entire arm versus controlling just the end-effector), passive and virtual joints, and even provides handy group states to access from MoveIt!

The files located in the **config** folder of a package can be changed for tweaking of the algorithm or visualization. `joint_limits.yaml` expresses further velocity and acceleration limits on the robot (the more conservative one from the URDF or the SRDF is considered). `kinematics.yaml` lists the kinematics plugin used for robot chain transformations. The kinematics plugin can be changed based on robot or need (`KDLKinematicsPlugin` is best for 7+ degree-of-freedom robots). `fake_controllers.yaml` describes how RViz should visualize the robot or trajectories (useful for simulations, since RViz cannot simulate the velocities or accelerations). Alternatively, a robot controller can be hooked up by `ros_controllers.yaml` for the execution or possible simulation of a trajectories.

The `ompl_planning.yaml` file is the most important file, detailing the sampling-based planning strategies available for the robot. OMPL stands for Open Motion Planning Library, and it includes generic geometric planners for numerous sampling-based planning strategies. The lab encourages the use of BFMT*, due to its speed and quality of paths, but other options are available as well. Explanation for the customization of the algorithm parameters can be seen here: `http://docs.ros.org/kinetic/api/moveit_tutorials/html/doc/ompl_interface/ompl_interface_tutorial.html`.

Additionally, MoveIt! is compatible with trajectory optimizing algorithms (instead of sampling-based algorithms), such as CHOMP and STOMP. CHOMP and STOMP require their own `.yaml` files. The `fmt_stomp.cpp` code in the `motoman_sia20d_moveit_updated` package uses both the OMPL library and the STOMP library. This requires the STOMP package to be installed and for the `stomp.yaml` file to be added. Please look in depth into the `fmt_stomp.launch` file and `http://docs.ros.org/kinetic/api/moveit_tutorials/html/doc/stomp_planner/stomp_planner_tutorial.html` to see the necessary changes to incorporate both algorithms in the same node.

After the URDF is created and properly working, most people usually use the MoveIt! Setup Assistant to generate the SRDF and other MoveIt! files instead of hand-writing them. The CMakeFiles.txt and the package.xml files are usually edited for MoveIt! API and C++ compatiblility.

5. The MoveIt Algorithms and Code

MoveIt! uses geometric planners to determine paths and then time parameterizes to meet robot specifications. This means that the planners find a path without consideration for the robot's dynamics, and then use time parameterization techniques to ensure that robot velocities and accelerations never exceed the robot's joint limits.

5.1. The Algorithms

The **cost** of a plan is what the algorithm is trying to improve/minimize. The cost is currently defined by the total distance traveled by the joints (i.e. it tries to minimize total movement across all joints, equally weighted). Plans can be selected by adjusting the cost in the algorithm itself (look into `http://ompl.kavrakilab.org/optimizationObjectivesTutorial.html`), or by running multiple plans and keeping the plan that best fits the slightly modified cost criteria (e.g. prioritizing the motion of certain joints). The latter is recommended, due to the former requiring the editing of source code unless the optimization objective is predefined and default included in MoveIt!

Multiple algorithms have been provided in the packages. The lab encourages the use of the `fmt_shortcut.cpp` code, which combines the OMPL BFMT* algorithm (though any other OMPL library can be substituted here) with the shortcut method. The post-processing shortcut method prunes unnecessary motions to reduce the cost of a plan in a couple of hundredths of a second. The **BFMT*+Shortcut** algorithm creates plans in less than a second, allowing for multiple plans to be generated and the best one chosen. The lab recommends the use of Partial Shortcut for the post-processing algorithm.

Another option would be the **BFMT*+STOMP** algorithm. This algorithm uses the path generated by BFMT* to act a seed for the trajectory-optimizing STOMP algorithm. This results in a slightly smoother path, at the cost of significantly more time and the possibility of the post-processing algorithm's failure (though the seed is still recovered). The STOMP algorithm usually takes around 3-6 seconds (after about .7 seconds from BFMT*). The improvement is relatively minuscule compared to the increase in time. That said, STOMP performs well in fairly uncluttered environments and provides slightly more smooth paths.

Finally, the lab also provided the BFMT* algorithm. The shortcut algorithm provides a nicer path for almost no time, but using just the BFMT* algorithm allows for the usage of `MoveGroupInterface` - a MoveIt! class that allows for streamlined user experience at the cost of customization and control. With the pipeline already laid out, the lab recommends using the existing infrastructure for greater control.

5.2. Using the Shortcut Library

All of the shortcut algorithms are stored in the `fmt_shortcut.h` file located in the include folder of the `motoman_sia20d_moveit_updated` folder. To use the appropriate algorithm, create a `shortcut::Shortcut_Planner` object. The following parameters are needed to initialize the object:

1. `shortcut_method`: a string specifying the planner. The options are:
   (a) Regular
   (b) Adaptive
   (c) Partial (Recommended)
   (d) AdaptivePartial
2. `manipulator_group` a string specifying which manipulator group to use for collision-checking
3. `planning_time` the maximum amount of time allocated for planning

The following parameters are only needed for the implementation of Regular and Adaptive Shortcut:

1. `edge_length_discretization` a double for collision-checking purposes. Smaller numbers mean finer collision checking.
2. `waypoint_injection` a double that indicates the frequency to reinject waypoints into the trajectory after the shortcut method for time parameterization purposes.
3. `numberOfAdaptiveReptitions` an int that describes the number of times the shortcut method should be called with the oracle. Relevant only for Adaptive Shortcut.

To use the Shortcut library in another file, simply create an `include` folder within the specified package and create folders within specifying the path from catkin workspace to the package location. For example, for `motoman_sia20d_moveit_updated`, the `include` folder is located in `~/catkin_ws/src/motoman_sia20d_moveit_updated`, and with a `motoman` folder within the include folder and a `motoman_sia20d_moveit_updated` folder within the said `motoman` folder. Then, add the following line to link the include folder to the package: `include_directories(include\ ${catkin_INCLUDE_DIRS}\${Boost_INCLUDE_DIR}\${EIGEN3_INCLUDE_DIRS})`. Now, the `Shortcut_Planner` class can be included within the C++ code (For example, by `#include<motoman/motoman_sia20d_moveit_updated/fmt_shortcut.h>`.

5.3. Basic MoveIt! API

The basic MoveIt! code works as follows:

1. A `PlanningScene` and `RobotModel` are created to monitor the planning environment and the robot's current configuration

2. The `PlannerManager` loads a planner library with the FCL (Flexible Collision Library) collision detector
3. Objects are added to the world through `PlanningScene` and the `"/planning_scene"` topic
4. A `MotionPlanRequest` is created with:
   (a) a joint planning group (which joints to control)
   (b) a start joint state for the robot
   (c) a goal joint state for the robot
   (d) a reference to the `PlanningScene` (which in turn has a reference to the `RobotModel`)
   (e) a maximum planning time
   (f) an initial trajectory (only applicable for STOMP; STOMP seeds its trajectory by `TrajectoryConstraints`)
5. A `MotionPlanResponse` message is generated to detail the plan
6. The `MotionPlanResponse` trajectory is fed to the Shortcut tool
7. The final trajectory is published to `"/display_trajectory"` topic for RViz visualization
8. Any world or object manipulation is broadcasted in the `"/planning_scene"` topic

5.4. The MoveIt! Code

This section will focus on the code for the Motoman Sia20d robot, although a rough prototype of the Restore-L arm on the Spacecraft is provided in the `restorebot_moveit_desc` package.

The two useful packages are the `motoman_sia20d_moveit_upgraded` and `motoman_sia20d_with_gripper_moveit` packages. These packages use the URDF from the `motoman_sia20d_support` package (and the `franka_description` for the gripper). This package is forked from the `motoman` package available from `https://github.com/ros-industrial/motoman`, although the package is fairly dated and does not utilize the recent MoveIt! updates. As such, it is recommended to use the repository and packages from the lab, as they have been modified and are up to date.

`fmt_shortcut.launch` launches the BFMT*+Shortcut strategy while `fmt_stomp.launch` launches the BFMT*+STOMP strategy from the `motoman_sia20d_moveit_upgraded` package. These plans use the advanced Motion Planning API referenced `http://docs.ros.org/kinetic/api/moveit_tutorials/html/doc/motion_planning_api/motion_planning_api_tutorial.html`. The advantage of using this API is the greater level of control over the MoveIt! interface, such as allowing for easier post-algorithm evaluation and smoothing. `fmt.launch` simply launches the BFMT* algorithm, but through the `MoveGroupInterface` for easier user usage. One particularly useful aspect of the `MoveGroupInterface` is the inbuilt pipeline for grasping (assuming a robot is connected), although the process is fairly rudimentary. All of the above algorithms have the robot navigate around a fairly large rectangular prism obstacle to a goal location detailed in the SRDF.

The `motoman_sia20d_with_gripper_moveit` package includes a gripper attached to the robot for a pick and place demonstration. The demonstration shows off the BFMT*+Shortcut algorithm to do a complex pick and place action, navigating around obstacles to pick up an object, place said object at a goal location, and then return to home position. Unfortunately, MoveIt!, at the time of writing, does not support visualization of the attached obstacle during robot trajectory visualization. As such, the end-states are visualized to show that the held object was transferred. The process features 7 sub-actions to perform the complex maneuver.

From discussions with NASA Goddard, the primary purpose of the algorithm was determined to navigate the arm close to a goal location autonomously. The `motoman_sia20d_moveit_upgraded` package was created to fulfill this request. The pick and place code is meant to show the possible further usage cases of the algorithm, and additionally provides a streamlined `solve()` method for obtaining a plan given a start state, goal state, and some other general planning essential objects.

The `restorebot_moveit_desc` features beta, unedited code for viewing discretion. The package is provided for optional perusal, particularly if the user is interested in the usage of STOMP and CHOMP.

Important Note: Most MoveIt! classes have two variants: `moveit_msgs::` (for RViz visualization) and `moveit::core::` (for actual planning). Some classes can be easily converted to their counterpart via `"conversions.h"`, but others might require manual reassigning of elements. Check `http://docs.ros.org/jade/api/moveit_core/html/conversions_8h.html` to see the given API methods to convert between certain class types.

5.5. Editing the Launch File

This section details the editing process for the `fmt_shortcut.launch` file. The launch file has various `rosparam` parameters one can tune to easily allow for alteration of the algorithms without recompilation. Any

changes to the C++ file require the entire `catkin_ws` to be rebuilt.

1. `allowed_planning_time`
   (a) represents the maximum amount of time the planner is allowed to take to generate a path
2. `max_Iter`
   (a) represents the number of iterations the planner should be ran (with the shortcut added).
3. `shortcut`
   (a) specifies which shortcut method to use. Can be specified by "Regular", "AdaptivePartial", "Partial", or "AdaptivePartial".
4. `max_EdgeLength_Discretization`
   (a) represents how small each edge length should be for the discretization step of the collision checker. Relevant only for Regular and Adaptive Shortcut. Smaller means finer collision-checking at the cost of time.
5. `max_EdgeLength_Waypoint_Injection`
   (a) represents how small each edge length should be for waypoint injection. Only relevant for Regular and Adaptive Shortcut. Smaller values result in more waypoints.
6. `sbp_plugin`
   (a) specifies which planning library base to use. Default is the OMPL library (`ompl_interface/OMPLPlanner`), and it recommended not to change this value (STOMP and CHOMP, for example, could be used).
7. `environment`
   (a) Configures one of the pre-specified environments. Options are `box`, `sphere`, or `clutter`
8. `adaptive_repetitions`
   (a) Relevant only for Adaptive Shortcut. Specifies how many times the Regular Shortcut algorithm should be called with the oracle applied afterwards (to inject points).

5.6. A More Detailed Look into the Planners

The following is a more detailed look into how the planners work. Knowledge of how the planners is not necessary to use MoveIt!

FMT* (Fast Marching Trees) is a sampling-based algorithm [4]. Due to the difficulty of representing the obstacle region and the collision-free region of the of the robot in n-dimensional space, the algorithm samples robot configurations and uses a black-boxed collision checker to determine whether said samples are in collision with an obstacle. The FMT* algorithm evaluates a new node against the existing tree of connected nodes and connects the new node to the closest tree member (close is determined by the cost definition). If said connection is in collision, the new node (or in other words, that robot configuration) is discarded. The algorithm continues using the samples (default 1000 in `ompl.yaml` file) to connect the start state to the goal region. BFMT* (Bidirectional Fast Marching Trees) uses the same premise of FMT*, but instead has two trees growing from the start state and the goal region, respectively[3]. This allows for faster computation times.

The Shortcut technique is a simple post-processing tool for providing cleaner paths [1]. Given a trajectory, it tries to connect two random nodes within the trajectory with a straight line path. If said path is collision-free, then the nodes are connected by a straight line and the nodes in between are discarded, since a straight line path is the most optimal. This continues for how ever many specified repetitions (default: 30). Adaptive Shortcut uses the shortcut technique, but then afterwards uses an oracle to apply midpoints between the way points [2]. Thus, the shortcut technique can be run again to more closely hug the obstacles.

The Partial Shortcut technique is similar to the regular Shortcut Technique, except that the algorithm only applies to a random DoF [1]. This allows for more optimal paths due to individual DoF optimization, at the cost of slightly increased operation time. The Adaptive Partial Shortcut Technique continues on the premise of the Partial Shortcut Technique, but also weights each DoF based on its variation from the optimal straight line cost from start state to goal [6].

STOMP is trajectory-optimizer, which means that it improves upon a given trajectory to find a locally optimal path[5]. The algorithm uses noise to simulate different joint motions and thus move into trajectory towards a smoother path. The algorithm prioritizes reducing the acceleration, which causes smoother trajectories. Interestingly, STOMP treats obstacles as a soft-constraint (undesirable) versus a hard constraint (forbidden), which has the potential to allow for the traversal through particularly small obstacle regions depending on the weight of the obstacle region.

References

[1] R. Geraerts and M. H. Overmars. Creating high-quality paths for motion planning. *The International Journal of Robotics Research*, 26(8):845–863, 2007.

[2] D. Hsu. Randomized Single-query Motion Planning in Expansive Spaces. *PhD thesis, Dept. of Computer Science, Stanford University, Stanford, CA*, 2000.

[3] E. S. L. J. L. M. J. A. Starek, J. V. Gomez and M. Pavone. An asymptotically-optimal sampling-based algorithm for bi-directional motion planning. *IEEE/RSJ International Conference on Intelligent Robots Systems*, 2015.

[4] L. Janson and M. Pavone. Fast marching trees: a fast marching sampling-based method for optimal motion planning in many dimensions - extended version. *CoRR*, abs/1306.3532, 2013.

[5] M. Kalakrishnan, S. Chitta, E. Theodorou, P. Pastor, and S. Schaal. Stomp: Stochastic trajectory optimization for motion planning. In *2011 IEEE International Conference on Robotics and Automation*, pages 4569–4574, May 2011.

[6] J. Polden, Z. Pan, N. Larkin, and S. van Duin. Adaptive partial shortcuts: Path optimization for industrial robotics. *Journal of Intelligent & Robotic Systems*, 86(1):35–47, Apr 2017.