MoveIt! Tips and Tricks

Author: David Nie, Mitchell Torok

Contact: z5303303@ad.unsw.edu.au

Last Updated: 15th Oct, 2025

Disclaimer:

This guide is written by David Nie, a fifth year undergraduate student who happens to have a number of experience with ROS2 and MoveIt! through coursework and thesis.

The guide is not definitive, it is not an official guide by any means, so please take in what's written with a grain of salt.

Before we start, what you should know:

1. There is a reason why `MoveIt!` is not an industry standard, a reason why `MoveIt!` is mainly used in research to the author's knowledge. `MoveIt!` is a great at many things, but one downside that almost everyone agree with is that "It's a pain to work with, a lot of the time there is no guarentee, and when something goes wrong, there is no `printf()` for you to debug with."
2. Your mindset is important. Many people went down the rabbit hole of optimising the planner's performance without knowing how deep he/she needs to go to get the level of performance desired. To combat this, I want to share a quote that I heard from the Engineering Lead in Reach Robotics last year:

   > I asked him: "What planner did you use for motion planning?"
   >
   > He said something like: "Oh, we found something that works well for our use case, and we just sticked to that."

   I realised that it's a bit abstract, but the takeaway here is that there is no "one solution fit all", experiment and find one that suits your application is the golden rule.

   Before you start doing work for motion planning, ask yourself the following:

   → How long is an "acceptable" cycle time

   → What are the poses I need to move to, is any of them close to singularity?

   → Can I simplify movement sequence by using a custom-endeffector instead?

Head Demo Mitch's Remark

> "The moveJ and moveL commands you used previously in MTRN4230 are simple, low-level path planners. While they work for basic motion, they do not account for dynamic environments or the geometry of the robot itself. When using a custom end-effector or operating in a responsive environment, these commands can easily generate joint configurations where the robot collides with itself or becomes trapped.
>
> MoveIt, on the other hand, provides a comprehensive motion planning framework that considers the robot's kinematics, the size and shape of the end-effector, and obstacles in dynamic environments. When set up and verified, this allows for safe, collision-free trajectories and more robust planning in complex tasks."

Intro

Although I said earlier that there is no solution fit all, there are "solutions that fit most", and universal methods you can apply to optimise planner performance. These are:

→ **Constraints and Collision Planes**

  → Constraints include `position constraint` (e.g. plane, box, line), `joint constraint` and `orientation constraint`

  → Collision Planes restrict the area the planner is allowed to explore

→ **Planner Selection**

  → Planners like `RRTConnect` and `TRRT` are more likely to generate a more optimised path than other planners such as the default one `LBKPIECE` used in `ur_moveit_config` package leveraging `MoveIt! + OMPL`

→ **Alternative `moveit_config` package**

  → Dream of having a planner that perform similarly to one used in MTRN4230? Find your answer here.

Constraints and Collision Planes

Note:

→ Head demo Mitch and I both had positive experience with `orientation_constraint`

→ I would personally **add joint constraints as a last resort**, the tighter it is, the longer it will take for the planner to find a valid path. If you use a suitable planner + collision plane combo, it's generally faster in terms of cycle time than using a poorly-fitted-planner + same collision plane + joint constraints combination.

→ People often think adding joint constraints help the planner narrow the scope, hence leading to faster planning + more optimised path

  → In my experience, it's not the case. Adding joint constraints simply serve as an `assert()` at the end, telling the function `if !justify_constraints(curr_traj) continue; else return curr_traj`

→ However, collision planes and orientation constraints seem to have little negative effects on performance

→ In my previous `ScrewdrivingBot` project, my teammates used a `line_constraint` to lower the screw to the screwhole, and there wasn't much of a drop in performance

→ https://www.youtube.com/shorts/ufF1myLWomA

→ No link to the source code as I don't think the style was particularly inspiring

Collision Planes

→ These are pretty self explanatory, and you should have them almost all the time.

→ by adding collision planes dynamically based on detected obstacles → you now have `dynamic obstacle avoidance`

Example of collision plane setup:

https://github.com/DaviddNie/UR10e_vision_based_fruit_harvesting/blob/main/src/moveit_path_planner/src/moveit_path_planning_server.cpp

```
void setupCollisionObjects() {
  std::string frame_id = "world";
  moveit::planning_interface::PlanningSceneInterface planning_scene_interface;

  planning_scene_interface.applyCollisionObject(generateCollisionObject(2.4, 0.04, 3.0, 0.70, -0.60, 0.5, frame_id, "backWall"));
  planning_scene_interface.applyCollisionObject(generateCollisionObject(0.04, 2.4, 3.0, -0.55, 0.25, 0.8, frame_id, "sideWall"));
  planning_scene_interface.applyCollisionObject(generateCollisionObject(3, 3, 0.01, 0.85, 0.25, 0.05, frame_id, "table"));
  planning_scene_interface.applyCollisionObject(generateCollisionObject(2.4, 2.4, 0.04, 0.85, 0.25, 1.5, frame_id, "ceiling"));
}
```

Constraints:

→ There are three types of common constraints, `position`, `orientation` and `joint`

→ Note that I can't guarentee that I'm applying the best practices, so if you found a better way than what I'm preaching here, I encourage you to document your finding in the forum or your blog and share it with the cohort, we might reward you bonus marks!

Joint Constraint

```
moveit_msgs::msg::Constraints constraints;

moveit_msgs::msg::JointConstraint elbow_constraint;
elbow_constraint.joint_name = "elbow_joint";

// Convert degrees to radians
const double min_angle = 60.0 * M_PI / 180.0;   // 60° in radians (~1.0472)
const double max_angle = 150.0 * M_PI / 180.0;  // 150° in radians (~2.6179)

// Calculate midpoint (105° which is between 60° and 150°)
const double midpoint = (min_angle + max_angle) / 2.0;  // ~1.8326 radians

// Set constraints
elbow_constraint.position = midpoint;
elbow_constraint.tolerance_below = 1.0;  // ~0.7854 radians (45°)
elbow_constraint.tolerance_above = max_angle - midpoint;  // ~0.7854 radians (45°)
elbow_constraint.weight = 1.0;

RCLCPP_INFO(node_->get_logger(), "elbow constraint implemented.");

constraints.joint_constraints.push_back(elbow_constraint);

return constraints;
```

Orientation Constraint

→ If your movement sequence involves having the end-effector facing at a certain angle, orientation constraint can often help you with that

```
moveit_msgs::msg::Constraints set_constraint(const std::string& constraint_str) {
    moveit_msgs::msg::Constraints constraints;

    if (str_contains(constraint_str, ORIEN)) {
        constraints.orientation_constraints.push_back(set_orientation_constraint(constraint_str));
    }

    return constraints;
}

moveit_msgs::msg::OrientationConstraint set_orientation_constraint(const std::string& orien_constraint_str) {
    moveit_msgs::msg::OrientationConstraint orientation_constraint;

    // Common setup for all orientations
    orientation_constraint.header.frame_id = move_group_->getPlanningFrame();
    orientation_constraint.link_name = move_group_->getEndEffectorLink();
    orientation_constraint.absolute_x_axis_tolerance = 0.001;
    orientation_constraint.absolute_y_axis_tolerance = 0.001;
    orientation_constraint.absolute_z_axis_tolerance = 0.001;
    orientation_constraint.weight = 1;

    // Set orientation based on input parameter
    tf2::Quaternion q;

    if (str_contains(orien_constraint_str, DOWN)) {
        // Facing downward: Z-axis pointing down (Roll=0, Pitch=π, Yaw=0)
        q.setRPY(0, M_PI, 0);
    }
    else if (str_contains(orien_constraint_str, LEFT)) {
        // Facing left: X-axis pointing left (Roll=0, Pitch=0, Yaw=π/2)
        q.setRPY(0, 0, M_PI_2);
    }
    else if (str_contains(orien_constraint_str, RIGHT)) {
        // Facing right: X-axis pointing right (Roll=0, Pitch=0, Yaw=-π/2)
        q.setRPY(0, 0, -M_PI_2);
    }
    else if (str_contains(orien_constraint_str, UP)) {
        // Facing upward: Z-axis pointing up (Roll=0, Pitch=0, Yaw=0)
        q.setRPY(0, 0, 0);
    }

    orientation_constraint.orientation = tf2::toMsg(q);

    return orientation_constraint;
}
```

Useful Links:

Moveit Tutorial on Constrained Planning →

https://moveit.picknik.ai/main/doc/how_to_guides/using_ompl_constrained_planning/ompl_constrained_planning.html

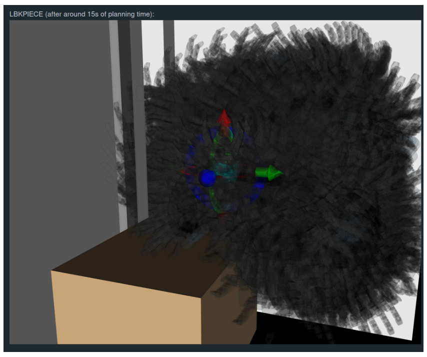
My past project that involves setting constraints →

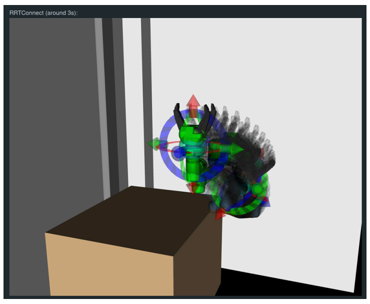
https://github.com/DaviddNie/UR10e_vision_based_fruit_harvesting/blob/main/src/moveit_path_planner/src/moveit_path_planning_server.cpp#L63

Planner Selection

The short takeaway here is: "The default planner is no good".

I want to reference an example from a forum thread (https://github.com/moveit/moveit/issues/197), the LHS showcases all the trajectories computed using the default planner `LBKPIECE`, and the RHS showcases the same by using `RRTConnect` Planner.

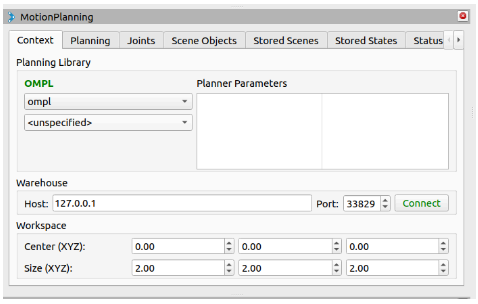

LBKPIECE (after around 15s of planning time):


RRTConnect (around 3s):

Q: well, how do I change the planner than?

A: From `Motionplanning` plugin, you can change the OMPL (Open Motion Path Planning Library) setting from `<unspecified>` to `RRTConnect`

A quick sanity check is to `Crtl+F` in the terminal where the `moveit_config` is launched, and search for `geometric` and you'll find the actual planner used.


MotionPlanning

Context | Planning | Joints | Scene Objects | Stored Scenes | Stored States | Status

Planning Library

OMPL

ompl

<unspecified>

Planner Parameters

Warehouse

Host: 127.0.0.1 Port: 33829 Connect

Workspace

Center (XYZ): 0.00 0.00 0.00

Size (XYZ): 2.00 2.00 2.00

Q: But I don't wanna set it every time I open up `rviz`

A: Easy, in your `moveit_server` that communicate with `moveit` , use the function `setPlannerId`

```
// Initialize MoveGroupInterface with proper parameters
// base_link → tool0
move_group_ = std::make_shared<moveit::planning_interface::MoveGroupInterface>(
  node_,
  "ur_manipulator",
  // ^ this is the name of the movegroup defined by the moveit-config-package
  std::shared_ptr<tf2_ros::Buffer>(),
  rclcpp::Duration::from_seconds(5.0)
);

move_group_->setPlannerId("RRTConnect");
// ^ this is the name of the planner in your `ompl_planning.yaml` file
// from the `moveit_config` package you are using

// If you use `ur_moveit_config` package, the equivelent will be
move_group_->setPlannerId("RRTConnectkConfigDefault");

// Good ones to try
move_group_->setPlannerId("RRTConnectkConfigDefault");
move_group_->setPlannerId("TRRTkConfigDefault");

// The default/fallback one is
move_group_->setPlannerId("LBKPIECEkConfigDefault");
```

A comprehensive list of planners available in `ompl`

https://ompl.kavrakilab.org/planners.html

Note:

→ Not all planners listed are available through `MoveIt! + ompl`

→ Most of the time, it's better to stick with a non-optimising planner that is good at finding an approximately optimised path (e.g. `TRRT`) than an optimising planner, because in my experience, optimised path often takes much longer to compute which negates the benefits of short paths.

→ Most optimising planners only operates on a single thread, but there are planners that can leverage multithreading, may want to give those a go.

For those using `ur_moveit_config`

Q: How can I edit high-level planning parameters?

A:

Step 1: Go to your `ur_moveit_config` package directory by running `ros2 pkg prefix <package_name>`

→ In your VM, it's at `~/4231/ros_ur_driver/src/Universal_Robots_ROS2_Driver/ur_moveit_config/`

→ Note that if it's downloaded through binary. you will need to download the source code in order to edit the package, as source code is not included for binary installation.

→ **Pro Tip:** For better encapsulation, a better way would be to make a copy of the `ur_moveit_config` to your project_repo, and edit that stand-alone package instead

Step 2: go to `/config/ompl_planning.yaml`

→ you can now view and edit the planner parameters, make sure to build and source it afterwards

Example `ompl_planning.yaml`:

https://github.com/DaviddNie/UR10e_vision_based_fruit_harvesting/blob/main/src/ur10e_moveit_config_official/config/ompl_planning.yaml

Alternative `moveit_config` package

`IMPORTANT NOTE`: This method to the author's knowledge, does not work well when the flange joint is not pointing downwards, if your project invovles rotating `wrist` to have the `flange` facing parallel to the table, `ur_moveit_config` with `TRRT` is a good starting point.

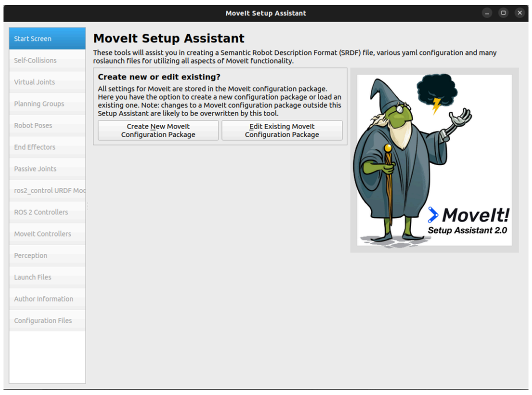
→ Example (Vertical Pick & Place) :

https://www.youtube.com/watch?v=1r7PfkH8pU8

If you are doing some sort of horizontal pick and place, this is the one to go

→ Example (Horizontal Pick & Place):

https://www.youtube.com/watch?v=19EWXa9_rUo



MoveIt Setup Assistant

These tools will assist you in creating a Semantic Robot Description Format (SRDF) file, various yaml configuration and many roslaunch files for utilizing all aspects of MoveIt functionality.

Create new or edit existing?

All settings for MoveIt are stored in the MoveIt configuration package. Here you have the option to create a new configuration package or load an existing one. Note: changes to a MoveIt configuration package outside this Setup Assistant are likely to be overwritten by this tool.

Instructions

→ Option 1: Follow the instructions on

https://docs.universal-robots.com/Universal_Robots_ROS_Documentation/doc/ur_tutorials/my_robot_cell/doc/build_moveit_config.html

Note

→ You **can't** simply copy my package at the following link because it's tailored for `ur10e`
https://github.com/DaviddNie/UR10e_vision_based_fruit_harvesting/tree/main/src/ur10e_moveit_config

→ Before you select your `.xacro` file in Step 1, you have to source where you stored the `.xacro` file in the terminal that launches the `setup_assistant` to provide context, otherwise the program will crash

→ `setup_assistant` does not generate a `ompl_planning.yaml` to my knowledge, and I couldn't figure out a simple way to edit/tune planning paramters (again, if you find a way, let us know in the forum and we will probably give you bonus marks)

Further readings

The key forum thread that answered a lot of my questions

https://github.com/moveit/moveit/issues/197

Universal Robot's Guide on MoveIt! Setup Assistant

https://docs.universal-robots.com/Universal_Robots_ROS_Documentation/doc/ur_tutorials/my_robot_cell/doc/build_moveit_config.html

MoveIt!'s Official Guide on MoveIt! Setup Assistant

https://moveit.picknik.ai/main/doc/examples/setup_assistant/setup_assistant_tutorial.html

All the planners available in MoveIt!

https://ompl.kavrakilab.org/planners.html

An example planner config, for customising the package

https://github.com/moveit/moveit_resources/tree/master/prbt_moveit_config

+==============================

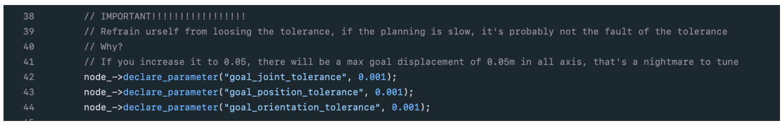
Other common pitfalls

1. Increasing planning time ≠ more optimised trajectory

~~`node_→declare_parameter("planning_time", 20.0);`~~

For non-optimised planners, visualise it like a spider web, once one of the lines touches the destination, **BANG**, it returns that trajectory

2. Do not loosen goal tolerance!!! (See comment)



```
// IMPORTANT!!!!!!!!!!!!!!!!!
// Refrain urself from loosing the tolerance, if the planning is slow, it's probably not the fault of the tolerance
// Why?
// If you increase it to 0.05, there will be a max goal displacement of 0.05m in all axis, that's a nightmare to tune
node_->declare_parameter("goal_joint_tolerance", 0.001);
node_->declare_parameter("goal_position_tolerance", 0.001);
node_->declare_parameter("goal_orientation_tolerance", 0.001);
```

3. If you want to stop `wrist_3 / (joint_6) / flange` from rotating, slapping a joint_constraint is not gonna fix it

As explained before, constraints are not taken into account during the planning stage. Doing so will likely result in `Failed to execute. Constraint on wrist_3 is violated` in all planning attempts.

Q: How do I fix it then?

→ I fixed it by using a different moveit config shown above, but that doesn't mean there isn't another way to fix it, find out yourself.

4. Setting joint targets does not making planning behaviour more predictable

→ Reason being the trajectory from A → B is still generated by the planner

```
bool set_joint_target(const std::vector<double>& positions) {
  try {
      auto joint_targets = create_joint_map_from_positions(positions);
      move_group_→setJointValueTarget(joint_targets);
      return true;
  } catch (const std::exception& e) {
      RCLCPP_ERROR(node_→get_logger(), "Failed to set joint target: %s", e.what());
      return false;
  }
}
```

Q: Is there a fix then?

A: Yes there is. if you frequently go between fixed points, I believe you can cache the tracjectory generated from A to B, something like:

```
if input in cache:
   return cache.find(input)
else:
    traj = generate_traj(input)

    if (valid_path && not_in_cache):
        cache[input] = traj

    return traj
```

5. Planning near singularities?

→ It's hard, the default behaviour of `moveit+ompl` when planning near singularities is to take the longer path, which depends on the planner used, can increase the cycle time anywhere from `15s` to `2min`.

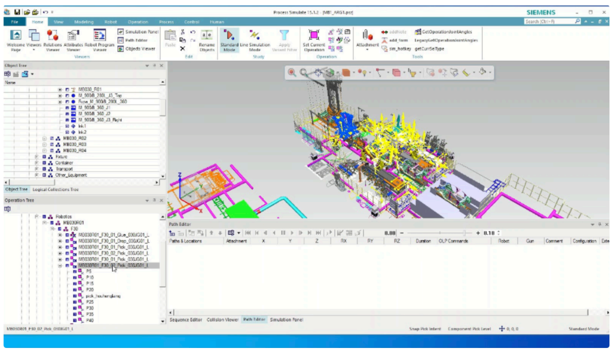
→ In my experience, `TRRT + ur_moveit_config` is the way to go

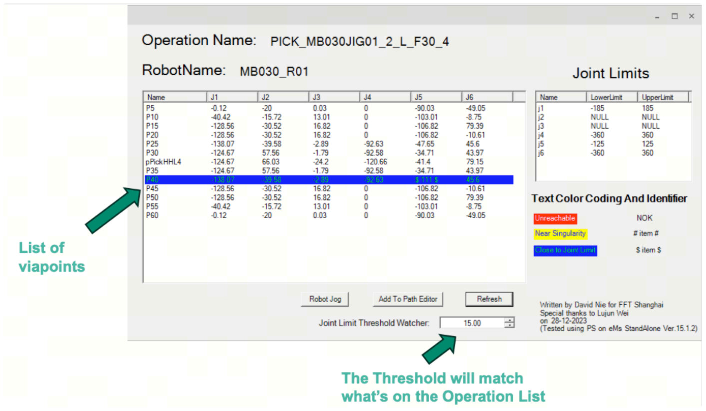
Q: Can you combat this by setting a via-point?

A: The issue is, the planner will likely prefer to take the longer path to get to the via-point as well, which defeats the purpose.

→ **Possible Solutions:**

→ I didn't have time to explore this, but similar to the standard industry approach to the problem, I believe the easiest way would be to switch to the embedded joint controller, in this case through `RTDE` which supports `movel` , `movej` . Reason being despite low-level controllers lack obstacle-avoidance capabilities like MoveIt!, their planning behaviour is independent of the robot's joint configuration.


SIEMENS Process Simulate UI, the yellow arrows are the welding operation nodes



Operation Name: PICK_MB030JIG01_2_L_F30_4

RobotName: MB030_R01

Name	J1	J2	J3	J4	J5	J6
P5	-0.12	-20	0.03	0	-90.03	-49.05
P10	-40.42	-15.72	13.01	0	-103.01	-8.75
P15	-128.56	-30.52	16.82	0	-106.82	79.39
P20	-128.56	-30.52	16.82	0	-106.82	-10.61
P25	-138.07	-39.58	-2.89	-92.63	-47.65	45.6
P30	-124.67	57.56	-1.79	-92.58	-34.71	43.97
pPickHHL4	-124.67	66.03	-24.2	-120.66	-41.4	79.15
P35	-124.67	57.56	-1.79	-92.58	-34.71	43.97
P40	-138.07	-39.58	-2.89	-92.63	$ 111 $	45.6
P45	-128.56	-30.52	16.82	0	-106.82	-10.61
P50	-128.56	-30.52	16.82	0	-106.82	79.39
P55	-40.42	-15.72	13.01	0	-103.01	-8.75
P60	-0.12	-20	0.03	0	-90.03	-49.05

Joint Limits

Name	LowerLimit	UpperLimit
j1	-185	185
j2	NULL	NULL
j3	NULL	NULL
j4	-360	360
j5	-125	125
j6	-360	360

Text Color Coding And Identifier

Unreachable	NOK
Near Singularity	# item #
Close to Joint Limit	$ item $

Robot Jog   Add To Path Editor   Refresh

Joint Limit Threshold Watcher: 15.00

Written by David Nie for FFT Shanghai
Special thanks to Lujun Wei
on 28-12-2023
(Tested using PS on eMs StandAlone Ver.15.1.2)

6. Pro Tip: Write your MoveIt! package in C++

→ My reasoning is that I've tried written it in python, and I encountered a problem that forces me to revert back to C++. I believe some MoveIt! features are only supported in C++, but don't quote me on this.

Head Demo Mitch's Remark

> Moveit does have a python api, it just directly calls the c++ interfaces, we didn't use it cause there is more documentation for the c++ interfaces.
>
> https://moveit.picknik.ai/main/doc/examples/motion_planning_python_api/motion_planning_python_api_tutorial.html

→ Feel free to just copy my `moveit_path_planner` package at the following link, don't reinvent the wheel

https://github.com/DaviddNie/UR10e_vision_based_fruit_harvesting/tree/main/src/moveit_path_planner