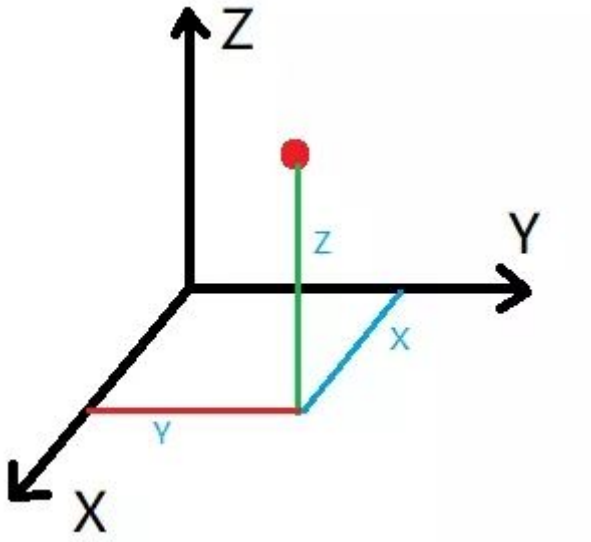
MoveIt Cartesian Path

Preface

We have built MoveIt environments on both Jetson_Nano and Orin series motherboards. Due to the onboard performance of Jetson_Nano, running the MoveIt program on the motherboard will be slow and slow to load. It takes about 3 minutes to complete the loading. Therefore, we recommend that users of Jetson motherboards run the MoveIt program on the configured virtual machine we provide. Orin motherboards can run MoveIt smoothly on the motherboard without running it on a virtual machine. Whether running on a virtual machine or on the motherboard, the startup instructions are the same. The following tutorials will take running on the Orin motherboard as an example.

1. Introduction

Cartesian coordinate system is a general term for rectangular coordinate system and oblique coordinate system. Cartesian path is actually the line connecting any two points in space



2. Functional Description

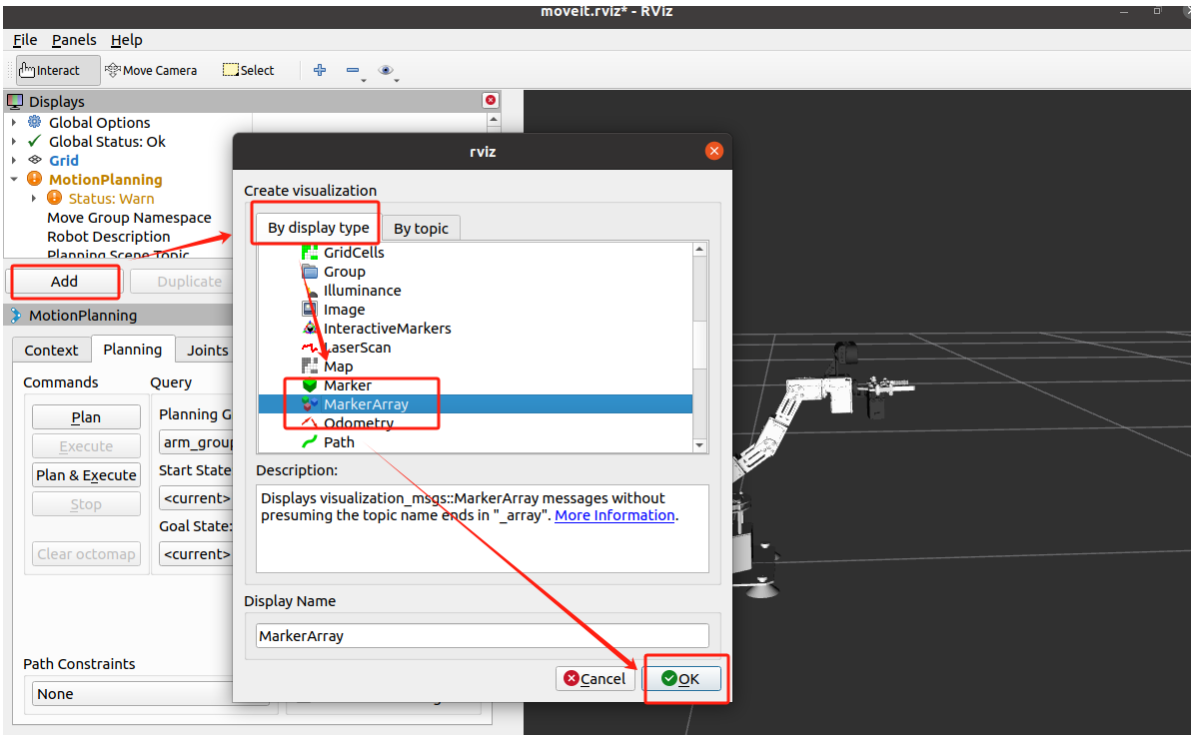
After the program runs, the robot arm will move according to the posture set by the program, and the running path will be displayed in rviz.

3. Start

First, start MoveIt. Enter the following command in the terminal to start it.

```
roslaunch dofbot_pro_config demo.launch
```

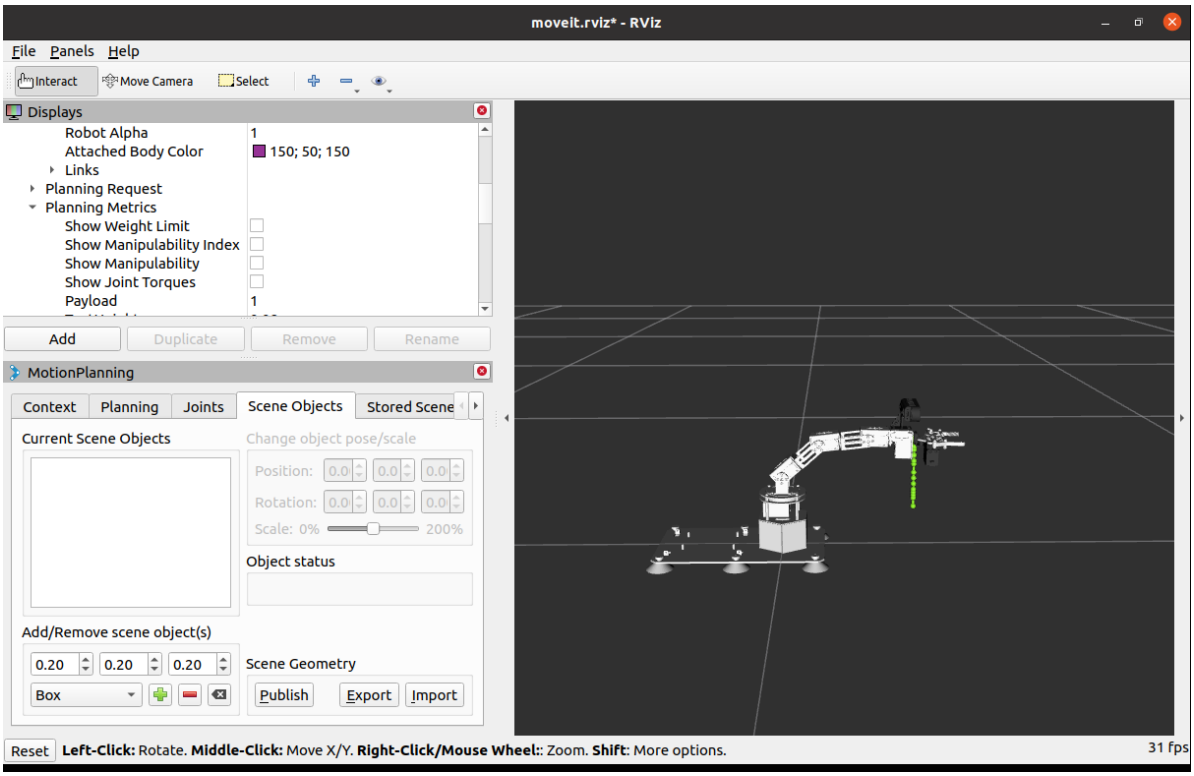
After MoveIt is successfully started, set up rviz and add the [MarkerArray] plug-in, as shown in the figure below.



Enter /rviz_visual_tools in [Marker Topic]. The setting is now complete. Press [Ctrl s] to save the rviz file.

Start Cartesian path node

```
rosrun arm_moveit_demo 04_cartesian
```



4. Core code analysis

Code path: `/home/jetson/dofbot_pro_ws/src/arm_moveit_demo/src/rosrun arm_moveit_demo 04_cartesian.cpp`

```
#include <iostream>
```

```cpp
#include "ros/ros.h"
#include <moveit/move_group_interface/move_group_interface.h>
#include <tf/LinearMath/Quaternion.h>
#include <moveit_visual_tools/moveit_visual_tools.h>
using namespace std;
int main(int argc, char **argv) {
    ros::init(argc, argv, "cartesian_plan_cpp");
    ros::NodeHandle n;
    ros::AsyncSpinner spinner(1);
    spinner.start();
    moveit::planning_interface::MoveGroupInterface dofbot("arm_group");
    string frame = dofbot.getPlanningFrame();
    moveit_visual_tools::MoveItVisualTools tool(frame);
    tool.deleteAllMarkers();
    dofbot.allowReplanning(true);
    // Planning time (unit: seconds)
    dofbot.setPlanningTime(50);
    dofbot.setNumPlanningAttempts(10);
    // Set the allowable target angle error
    dofbot.setGoalJointTolerance(0.001);
    dofbot.setGoalPositionTolerance(0.001);//0.01
    dofbot.setGoalOrientationTolerance(0.001);
    dofbot.setGoalTolerance(0.001);
    // Set the maximum allowed speed and acceleration
    dofbot.setMaxVelocityScalingFactor(1.0);
    dofbot.setMaxAccelerationScalingFactor(1.0);
    ROS_INFO("Set Init Pose.");
    // Set the specific location
    dofbot.setNamedTarget("up");
    dofbot.move();
    sleep(0.5);
    vector<double> pose{0, -1.57, -0.74, 0.71, 0};
    dofbot.setJointValueTarget(pose);
    sleep(0.5);
    moveit::planning_interface::MoveGroupInterface::Plan plan;
    dofbot.plan(plan);
    dofbot.execute(plan);

    // Get the current end position of the robot arm
    geometry_msgs::Pose start_pose =
dofbot.getCurrentPose(dofbot.getEndEffectorLink()).pose;
    std::vector<geometry_msgs::Pose> waypoints;
    // Add the initial pose to the waypoint list
    waypoints.push_back(start_pose);
    start_pose.position.z += 0.02;
    waypoints.push_back(start_pose);
    start_pose.position.z += 0.02;
    waypoints.push_back(start_pose);
    start_pose.position.z += 0.02;
    waypoints.push_back(start_pose);
    start_pose.position.z += 0.02;
    waypoints.push_back(start_pose);
    start_pose.position.z += 0.02;
    waypoints.push_back(start_pose);
    start_pose.position.z += 0.02;
    waypoints.push_back(start_pose);
    start_pose.position.z += 0.02;
    waypoints.push_back(start_pose);
    start_pose.position.y -= 0.02;
    waypoints.push_back(start_pose);
```

```
    start_pose.position.y -= 0.02;
    waypoints.push_back(start_pose);
    start_pose.position.y -= 0.02;

    // Path planning in Cartesian space
    moveit_msgs::RobotTrajectory trajectory;
    const double jump_threshold = 0.0;
    const double eef_step = 0.1;//0.1
    double fraction = 0.0;
    int maxtries = 100;   // Maximum number of attempts to plan
    int attempts = 0;     // Number of attempts to plan
    sleep(5.0);
    while (fraction < 1.0 && attempts < maxtries) {
        fraction = dofbot.computeCartesianPath(waypoints, eef_step,
jump_threshold, trajectory);
        //ROS_INFO("fraction: %f", fraction);
        attempts++;
        //if (attempts % 10 == 0) ROS_INFO("Still trying after %d attempts...",
attempts);
    }
    ROS_INFO("fraction: %f", fraction);
    if (fraction > 0.5) {
        ROS_INFO("Path computed successfully. Moving the arm.");
        // Generate motion planning data for the robotic arm
        moveit::planning_interface::MoveGroupInterface::Plan plan;
        plan.trajectory_ = trajectory;
        // Display the track
        tool.publishTrajectoryLine(plan.trajectory_, dofbot.getCurrentState()-
>getJointModelGroup("arm_group"));
        tool.trigger();
        // Execute the movement
        dofbot.execute(plan);
        sleep(1);
    } else {
        ROS_INFO("Path planning failed with only %0.6f success after %d
attempts.", fraction, maxtries);
    }
    return 0;
}
```