ROBT 403 Laboratory Report 4

Abdirakhman Onabek
Robotics Engineering, SEDS
Nazarbayev University
Astana, Kazakhstan
abdirakhman.onabek@nu.edu.kz

Daniil Filimonov
Robotics Engineering, SEDS
Nazarbayev University
Astana, Kazakhstan
daniil.filimonov@nu.edu.kz

Kir Smolyarchuk
Robotics Engineering, SEDS
Nazarbayev University
Astana, Kazakhstan
kir.smolyarchuk@nu.edu.kz

I. Introduction

This work presents the configuration and customization of a MoveIt package tailored for a snake robot powered by Dynamixel motors, with a focus on enhancing motion planning capabilities. MoveIt is a widely used robotic framework within ROS that facilitates motion planning, manipulation, and kinematic analysis, essential for achieving accurate control of robotic joints and end-effectors. The configuration process begins with defining the robot's URDF model, establishing the geometric and kinematic properties required for MoveIt to interpret the snake robot's structure [1]. By creating custom trajectory controllers, this setup enables MoveIt to coordinate the robot's joints smoothly, ensuring seamless, controlled movement [2]. This approach allows the robot to execute complex, precise paths, an important feature for applications demanding adaptability in unstructured environments. This work underscores the importance of MoveIt in bridging robotic hardware with software controls, highlighting its utility in managing complex robotic motion with precision and flexibility.

A. Notation and Terminology

To maintain clarity, the following notation and terminology are used throughout this work:

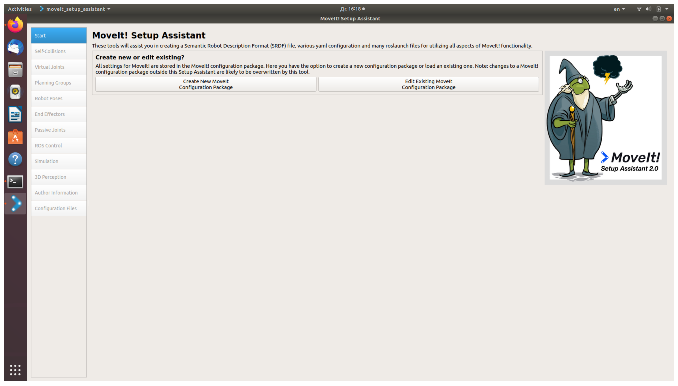
- $\theta_i(t)$: Joint angle of joint i on the snake robot.
- l_i: Length of link i of snake robot.
- (x, y): Cartesian coordinates of the end-effector.
- θ: Orientation of the end-effector.
- Via Points: Intermediate points that the snake robot's segments must pass through along the trajectory.
- URDF (Unified Robot Description Format): A standardized XML format used to describe the robot's model, including its joints and links.
- MoveIt Package: A ROS framework for motion planning and control, tailored here to manage the kinematics and trajectory planning of the snake robot.
- Joint Trajectory Controller: A controller that handles the smooth movement of individual joints based on segmented trajectory plans.
- Inverse Kinematics (IK): The process of calculating joint angles that achieve a specific position and orientation for each segment of the snake robot.
- Workspace: The set of reachable positions and orientations that the robot's end-effector or segments can attain within its physical and kinematic limits.
- t: Time variable for trajectory planning.
- t_f: Final time for a given trajectory segment.
- Dynamixel Motor: A modular actuator commonly used in robotics, providing precise control of joint angles and enabling flexible movement in the snake robot.

II. MoveIt Setup in Setup Assistant

A. Setting Up the MoveIt Setup Assistant for the Snake Robot

The MoveIt Setup Assistant allows us to configure MoveIt for our custom snake robot equipped with Dynamixel motors. This section details the configuration process, providing step-by-step instructions. Here we skip the steps like end-effectors, passive joints, virtual joints, simulation and ROS control.

1) **Create a New MoveIt Configuration Package**
In the MoveIt Setup Assistant, select "Create New MoveIt Configuration Package." This option will allow you to configure a new package specifically for the snake robot.



MoveIt! Setup Assistant

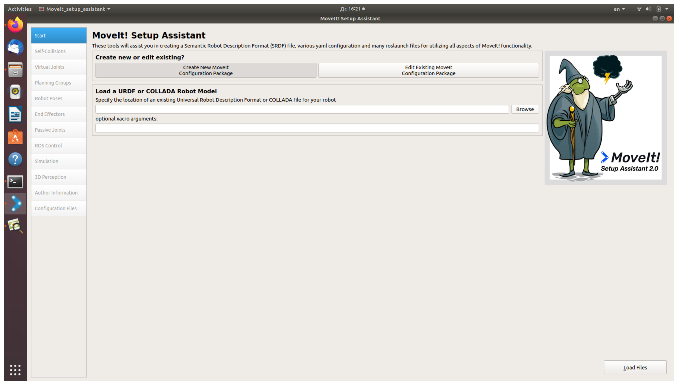
These tools will assist you in creating a Semantic Robot Description Format (SRDF) file, various yaml configuration and many roslaunch files for utilizing all aspects of MoveIt! functionality.

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.

Fig. 1. Create a New MoveIt Configuration Package

2) **Select the URDF File**
   - In the "URDF" tab, load the URDF file for your robot. Navigate to the location of your snake robot's URDF file, typically located in the `robot_description` folder of your workspace.
   - Click "Load Files" to import the URDF model into the assistant.



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.

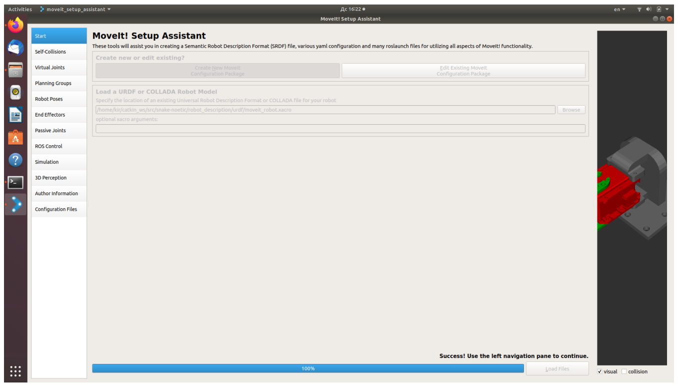
Create new or edit existing?

Load a URDF or COLLADA Robot Model

Specify the location of an existing Universal Robot Description Format or COLLADA file for your robot

optional xacro arguments:

Fig. 2. Choose URDF File



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?

Load a URDF or COLLADA Robot Model

Specify the location of an existing Universal Robot Description Format or COLLADA file for your robot

/home/kir/catkin_ws/src/snake-noetic/robot_description/urdf/moveit_robot.xacro

optional xacro arguments:

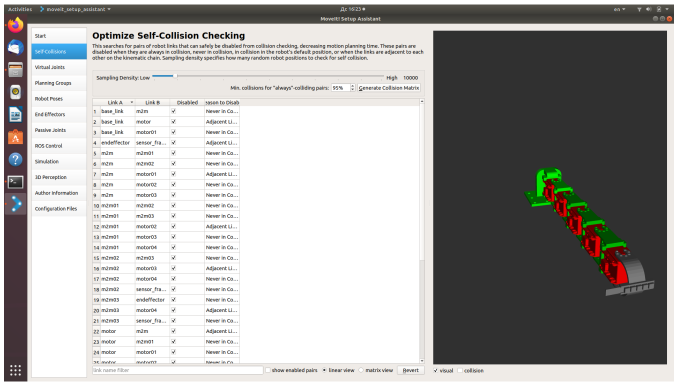
Success! Use the left navigation pane to continue.

100%

Fig. 3. URDF File Uploaded

3) **Generate Self-Collisions**

- In the "Self-Collisions" tab, click "Generate Collision Matrix." This process generates a list of potential self-collisions in the robot's joints and links, allowing MoveIt to avoid configurations that would cause the robot to collide with itself.
- This may take a few minutes. Once completed, you can view and adjust the collisions.



Optimize Self-Collision Checking

This searches for pairs of robot links that can safely be disabled from collision checking, decreasing motion planning time. These pairs are disabled when they are always in collision, never in collision, in collision in the robot's default position, or when the links are adjacent to each other on the kinematic chain. Sampling density specifies how many random robot positions to check for self collision.

Sampling Density: Low — High 10000

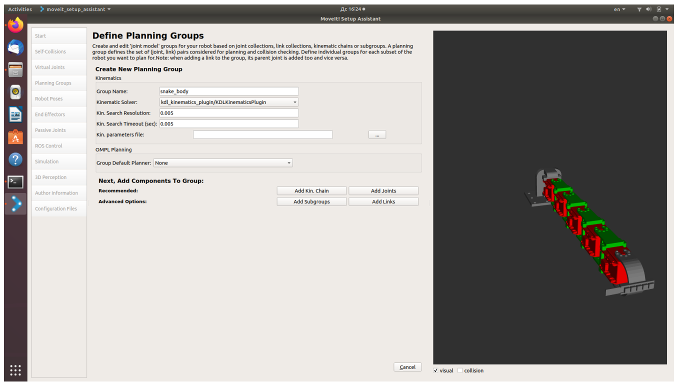
Min. collisions for "always"-colliding pairs: 95%

	Link A	Link B	Disabled	eason to Disab
1	base_link	m2m	✓	Never in Co...
2	base_link	motor	✓	Adjacent Li...
3	base_link	motor01	✓	Never in Co...
4	endeffector	sensor_fra...	✓	Adjacent Li...
5	m2m	m2m01	✓	Never in Co...
6	m2m	m2m02	✓	Never in Co...
7	m2m	motor01	✓	Adjacent Li...
8	m2m	motor02	✓	Never in Co...
9	m2m	motor03	✓	Never in Co...
10	m2m01	m2m02	✓	Never in Co...
11	m2m01	m2m03	✓	Never in Co...
12	m2m01	motor02	✓	Adjacent Li...
13	m2m01	motor03	✓	Never in Co...
14	m2m01	motor04	✓	Never in Co...
15	m2m02	m2m03	✓	Never in Co...
16	m2m02	motor03	✓	Adjacent Li...
17	m2m02	motor04	✓	Never in Co...
18	m2m02	sensor_fra...	✓	Never in Co...
19	m2m03	endeffector	✓	Never in Co...
20	m2m03	motor04	✓	Adjacent Li...
21	m2m03	sensor_fra...	✓	Never in Co...
22	motor	m2m	✓	Adjacent Li...
23	motor	m2m01	✓	Never in Co...
24	motor	motor01	✓	Never in Co...
25	motor	motor02	✓	Never in Co...

Fig. 4. Generate Collistion Matrix

4) **Define Planning Groups**

- Go to the "Planning Groups" tab to create groups of joints that will be planned together.
- In this setup, you can choose to create one primary group (e.g., `snake_body`) or multiple groups based on different sections of the snake robot. Each group should correspond to logical segments of the robot.
- To add a new planning group, click "Add Group," enter a name (e.g., `snake_body`), and select the joints that belong to this group. You may also define multiple groups if the configuration requires it. (Fig 5,6,7)



Define Planning Groups

Create and edit 'joint model' groups for your robot based on joint collections, link collections, kinematic chains or subgroups. A planning group defines the set of (joint, link) pairs considered for planning and collision checking. Define individual groups for each subset of the robot you want to plan for.Note: when adding a link to the group, its parent joint is added too and vice versa.

Create New Planning Group

Kinematics

Group Name: snake_body

Kinematic Solver: kdl_kinematics_plugin/KDLKinematicsPlugin

Kin. Search Resolution: 0.005

Kin. Search Timeout (sec): 0.005

Kin. parameters file:

OMPL Planning

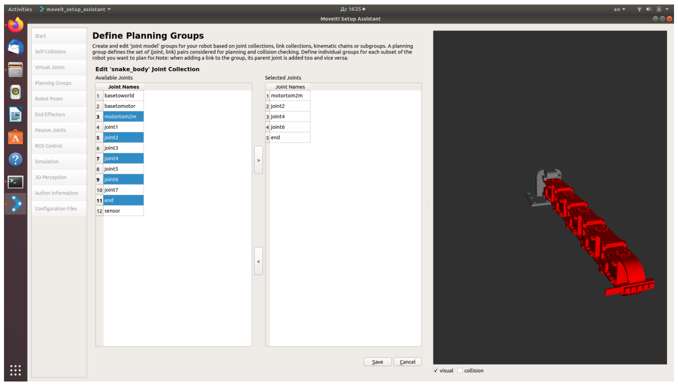
Group Default Planner: None

Next, Add Components To Group:

Recommended:

Advanced Options:

Fig. 5. Group Name and KDL Plugin Select



Define Planning Groups

Create and edit 'joint model' groups for your robot based on joint collections, link collections, kinematic chains or subgroups. A planning group defines the set of (joint, link) pairs considered for planning and collision checking. Define individual groups for each subset of the robot you want to plan for.Note: when adding a link to the group, its parent joint is added too and vice versa.

Edit 'snake_body' Joint Collection

Available Joints

	Joint Names
1	basetoworld
2	basetomotor
3	motortom2m
4	joint1
5	joint2
6	joint3
7	joint4
8	joint5
9	joint6
10	joint7
11	end
12	sensor

Selected Joints

	Joint Names
1	motortom2m
2	joint2
3	joint4
4	joint6
5	end

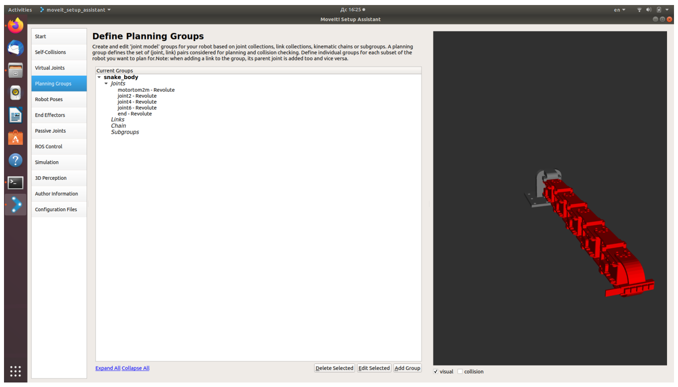
Fig. 6. Add your joints to Group

5) **Define Robot Poses**

- In the "Robot Poses" tab, define any desired default poses for the robot. These poses are useful for initializing the robot in specific configurations. (Fig 8)
- Click "Add Pose," name the pose (e.g., "init"), and set the joint values to represent a neutral starting configuration where all joints are at zero.

6) **Save and Generate Package**

- In the final step, select "Configuration Files" in the MoveIt Setup Assistant, then choose a directory in your workspace to save the configuration package.

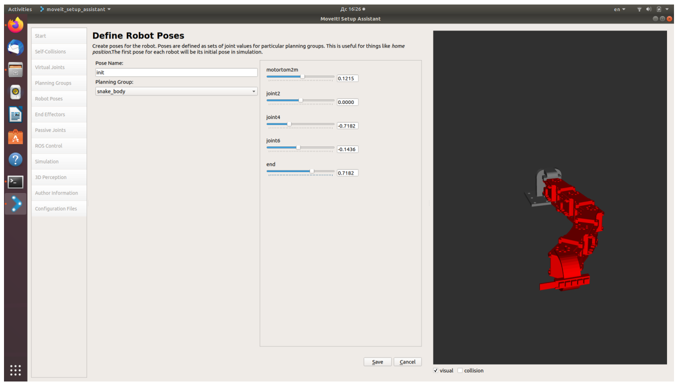


Define Planning Groups

Create and edit 'joint model' groups for your robot based on joint collections, link collections, kinematic chains or subgroups. A planning group defines the set of (joint, link) pairs considered for planning and collision checking. Define individual groups for each subset of the robot you want to plan for.Note: when adding a link to the group, its parent joint is added too and vice versa.

```
Current Groups
▾ snake_body
  ▾ Joints
      motortom2m - Revolute
      joint2 - Revolute
      joint4 - Revolute
      joint6 - Revolute
      end - Revolute
    Links
    Chain
    Subgroups
```

Fig. 7. Succesfully Created Group



Define Robot Poses

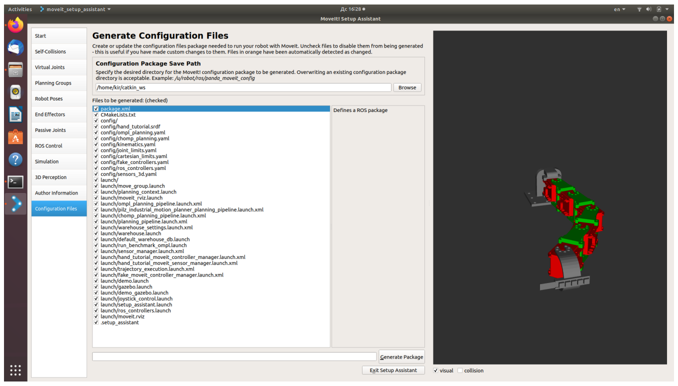
Create poses for the robot. Poses are defined as sets of joint values for particular planning groups. This is useful for things like *home position*.The first pose for each robot will be its initial pose in simulation.

Pose Name: init
Planning Group: snake_body

motortom2m	0.1215
joint2	0.0000
joint4	-0.7182
joint6	-0.1436
end	0.7182

Fig. 8. Define specific positions

- Click "Generate Package" to create the necessary configuration files for the MoveIt package. (Fig 9, 10)



Generate Configuration Files

Create or update the configuration files package needed to run your robot with MoveIt. Uncheck files to disable them from being generated - this is useful if you have made custom changes to them. Files in orange have been automatically detected as changed.

Configuration Package Save Path

Specify the desired directory for the MoveIt! configuration package to be generated. Overwriting an existing configuration package directory is acceptable. Example: */u/robot/ros/panda_moveit_config*

/home/kir/catkin_ws

Files to be generated: (checked)

```
✓ package.xml
✓ CMakeLists.txt
✓ config/
✓ config/hand_tutorial.srdf
✓ config/ompl_planning.yaml
✓ config/chomp_planning.yaml
✓ config/kinematics.yaml
✓ config/joint_limits.yaml
✓ config/cartesian_limits.yaml
✓ config/fake_controllers.yaml
✓ config/ros_controllers.yaml
✓ config/sensors_3d.yaml
✓ launch/
✓ launch/move_group.launch
✓ launch/planning_context.launch
✓ launch/moveit_rviz.launch
✓ launch/ompl_planning_pipeline.launch.xml
✓ launch/pilz_industrial_motion_planner_planning_pipeline.launch.xml
✓ launch/chomp_planning_pipeline.launch.xml
✓ launch/planning_pipeline.launch.xml
✓ launch/warehouse_settings.launch.xml
✓ launch/warehouse.launch
✓ launch/default_warehouse_db.launch
✓ launch/run_benchmark_ompl.launch
✓ launch/sensor_manager.launch.xml
✓ launch/hand_tutorial_moveit_controller_manager.launch.xml
✓ launch/hand_tutorial_moveit_sensor_manager.launch.xml
✓ launch/trajectory_execution.launch.xml
✓ launch/fake_moveit_controller_manager.launch.xml
✓ launch/demo.launch
✓ launch/gazebo.launch
✓ launch/demo_gazebo.launch
✓ launch/joystick_control.launch
✓ launch/setup_assistant.launch
✓ launch/ros_controllers.launch
✓ launch/moveit.rviz
✓ .setup_assistant
```

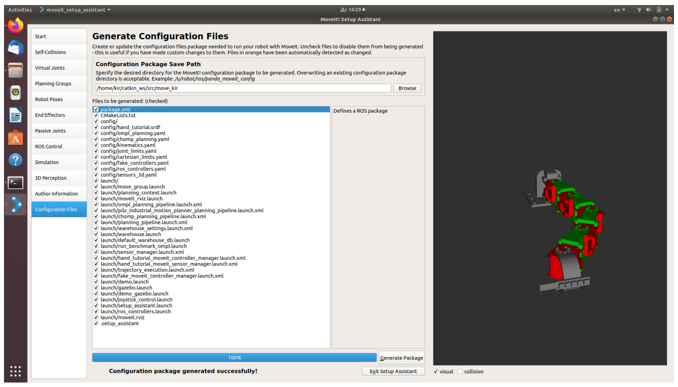
Defines a ROS package

Fig. 9. Save your package

This setup completes the configuration for the MoveIt package. Insert screenshots at each step for clarity, and ensure that each step's specific details align with the instructions provided in the MoveIt tutorial and lab notes.

III. Move the End-Effector "end" by 1.4

The code in Fig. 11 was used to implement cartesian control and move the end-effector by 1.4 units.



Generate Configuration Files

Create or update the configuration files package needed to run your robot with MoveIt. Uncheck files to disable them from being generated - this is useful if you have made custom changes to them. Files in orange have been automatically detected as changed.

Configuration Package Save Path

Specify the desired directory for the MoveIt! configuration package to be generated. Overwriting an existing configuration package directory is acceptable. Example: */u/robot/ros/panda_moveit_config*

/home/kir/catkin_ws/src/move_kir

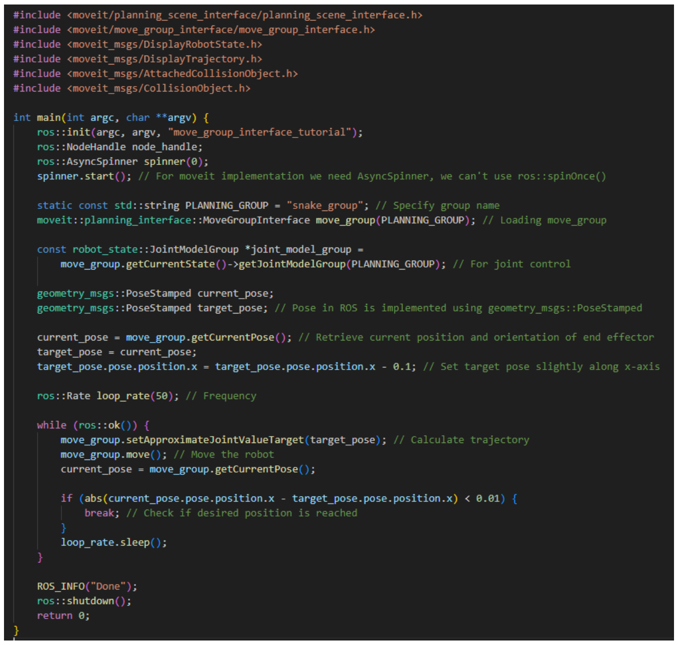
Files to be generated: (checked)

```
✓ package.xml
✓ CMakeLists.txt
✓ config/
✓ config/hand_tutorial.srdf
✓ config/ompl_planning.yaml
✓ config/chomp_planning.yaml
✓ config/kinematics.yaml
✓ config/joint_limits.yaml
✓ config/cartesian_limits.yaml
✓ config/fake_controllers.yaml
✓ config/ros_controllers.yaml
✓ config/sensors_3d.yaml
✓ launch/
✓ launch/move_group.launch
✓ launch/planning_context.launch
✓ launch/moveit_rviz.launch
✓ launch/ompl_planning_pipeline.launch.xml
✓ launch/pilz_industrial_motion_planner_planning_pipeline.launch.xml
✓ launch/chomp_planning_pipeline.launch.xml
✓ launch/planning_pipeline.launch.xml
✓ launch/warehouse_settings.launch.xml
✓ launch/warehouse.launch
✓ launch/default_warehouse_db.launch
✓ launch/run_benchmark_ompl.launch
✓ launch/sensor_manager.launch.xml
✓ launch/hand_tutorial_moveit_controller_manager.launch.xml
✓ launch/hand_tutorial_moveit_sensor_manager.launch.xml
✓ launch/trajectory_execution.launch.xml
✓ launch/fake_moveit_controller_manager.launch.xml
✓ launch/demo.launch
✓ launch/gazebo.launch
✓ launch/demo_gazebo.launch
✓ launch/joystick_control.launch
✓ launch/setup_assistant.launch
✓ launch/ros_controllers.launch
✓ launch/moveit.rviz
✓ .setup_assistant
```

Defines a ROS package

Configuration package generated successfully!

Fig. 10. Successful package generation



```cpp
#include <moveit/planning_scene_interface/planning_scene_interface.h>
#include <moveit/move_group_interface/move_group_interface.h>
#include <moveit_msgs/DisplayRobotState.h>
#include <moveit_msgs/DisplayTrajectory.h>
#include <moveit_msgs/AttachedCollisionObject.h>
#include <moveit_msgs/CollisionObject.h>

int main(int argc, char **argv) {
    ros::init(argc, argv, "move_group_interface_tutorial");
    ros::NodeHandle node_handle;
    ros::AsyncSpinner spinner(0);
    spinner.start(); // For moveit implementation we need AsyncSpinner, we can't use ros::spinOnce()

    static const std::string PLANNING_GROUP = "snake_group"; // Specify group name
    moveit::planning_interface::MoveGroupInterface move_group(PLANNING_GROUP); // Loading move_group

    const robot_state::JointModelGroup *joint_model_group =
        move_group.getCurrentState()->getJointModelGroup(PLANNING_GROUP); // For joint control

    geometry_msgs::PoseStamped current_pose;
    geometry_msgs::PoseStamped target_pose; // Pose in ROS is implemented using geometry_msgs::PoseStamped

    current_pose = move_group.getCurrentPose(); // Retrieve current position and orientation of end effector
    target_pose = current_pose;
    target_pose.pose.position.x = target_pose.pose.position.x - 0.1; // Set target pose slightly along x-axis

    ros::Rate loop_rate(50); // Frequency

    while (ros::ok()) {
        move_group.setApproximateJointValueTarget(target_pose); // Calculate trajectory
        move_group.move(); // Move the robot
        current_pose = move_group.getCurrentPose();

        if (abs(current_pose.pose.position.x - target_pose.pose.position.x) < 0.01) {
            break; // Check if desired position is reached
        }
        loop_rate.sleep();
    }

    ROS_INFO("Done");
    ros::shutdown();
    return 0;
}
```

Fig. 11. Move the end-effector by 1.4

This code is a ROS (Robot Operating System) C++ program utilizing the MoveIt library to control a robot's end effector. It retrieves the end effector's current pose, defines a target position slightly shifted along the x-axis, and moves the robot to this position, checking the end position accuracy.

The program initializes a ROS node to control the robot's movement in a collision-aware manner using MoveIt. The main objectives are to retrieve the current position, set a target pose, calculate an approximate trajectory, and execute the movement until reaching the desired location.

Library and Node Initialization: After including essential ROS and MoveIt libraries, the code initializes the node with a specific name. The AsyncSpinner is employed to allow non-blocking MoveIt commands to execute alongside ROS callbacks, which is essential for smooth robotic operations.

Setting the Planning Group: A specific planning group, "snake-group", is defined to identify which part of the robot (joint set) is being controlled. The MoveGroupInterface object is then instantiated to provide direct access to the MoveIt API, allowing manipulation of this group's joints.

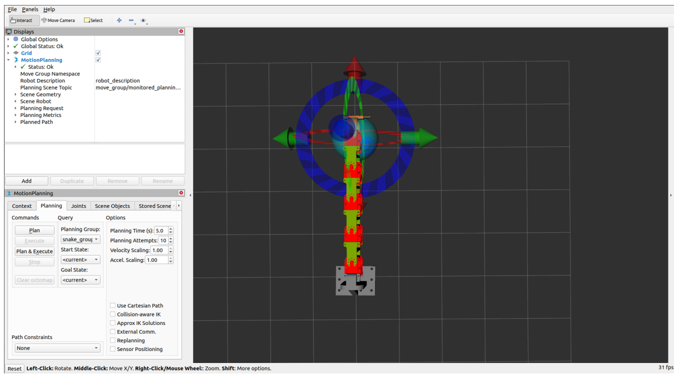
Joint Model Group Access: By retrieving the joint model group for the specified planning group, the code gains access to control the individual joints directly, allowing for additional customization if finer control over joint angles is required.

Pose Initialization: The current pose of the end effector is obtained, storing both position and orientation in a PoseStamped object. A target pose is then defined, starting as a copy of the current pose to make adjustments easy.

Defining Target Pose: The target pose is created by shifting the x-position of the end effector slightly, indicating that the robot should move a small distance in the x-direction.

Loop-Based Control Execution: A control loop checks if the target pose has been reached by continually comparing the current position to the target. Within the loop:

An approximate trajectory to the target is calculated, and a movement command is issued to start the motion. The current pose is updated after each movement to verify whether the target position has been reached within a specified tolerance. The loop breaks once the end effector is within the set tolerance, ensuring precise positioning. Completion and Shutdown: Once the target position is reached, a completion message is logged, and the ROS node is safely shut down.



Global Options
Global Status: Ok
Grid
MotionPlanning
Status: Ok
Move Group Namespace
Robot Description robot_description
Planning Scene Topic move_group/monitored_plannin...
Scene Geometry
Scene Robot
Planning Request
Planning Metrics
Planned Path

Planning Group: snake_group
Start State: <current>
Goal State: <current>
Planning Time (s): 5.0
Planning Attempts: 10
Velocity Scaling: 1.00
Accel. Scaling: 1.00
Use Cartesian Path
Collision-aware IK
Approx IK Solutions
External Comm.
Replanning
Sensor Positioning
Path Constraints: None

Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel:: Zoom. Shift: More options.

31 fps

Fig. 12. Init Pose in RVIZ

IV. Move the End-Effector in a Rectangle

The code in Fig. 12 was used to implement cartesian control and move the end-effector in a rectangle.

This code expands the functionality of the previous program by moving the robot's end effector in a rectangular path. After initializing the necessary MoveIt interfaces and retrieving the initial pose, it modifies the target pose iteratively to achieve a sequence of movements along each side of the rectangle.

Initialization: The program sets up the initial pose of the end effector and specifies a 50 Hz loop rate to control movement.

Rectangular Movement Sequence: A while loop is used to execute the rectangle pattern once: First Move: The end effector moves 0.5 units along the negative x-axis. Second Move: It then shifts 0.5 units along the positive y-axis. Third Move: Moves another 0.5 units along the negative x-axis. Fourth Move: Moves back along the negative y-axis by 0.5 units. Fifth Move: Completes the rectangle by moving back 0.5 units along the positive x-axis. Target Pose Update and Movement Execution:

Each step updates the target pose values and sets them as the new joint targets for MoveGroupInterface. The movement is executed immediately after setting the target pose, with move-group.move() handling each motion. Completion:

The loop runs through each of the rectangle's four sides before breaking and shutting down ROS. This code works because move-group.move() is not and async function. It will stop the code execution until target is reached.

Demonstration of the rectangle movement is here: https://youtu.be/QhKZhWJ3ck8

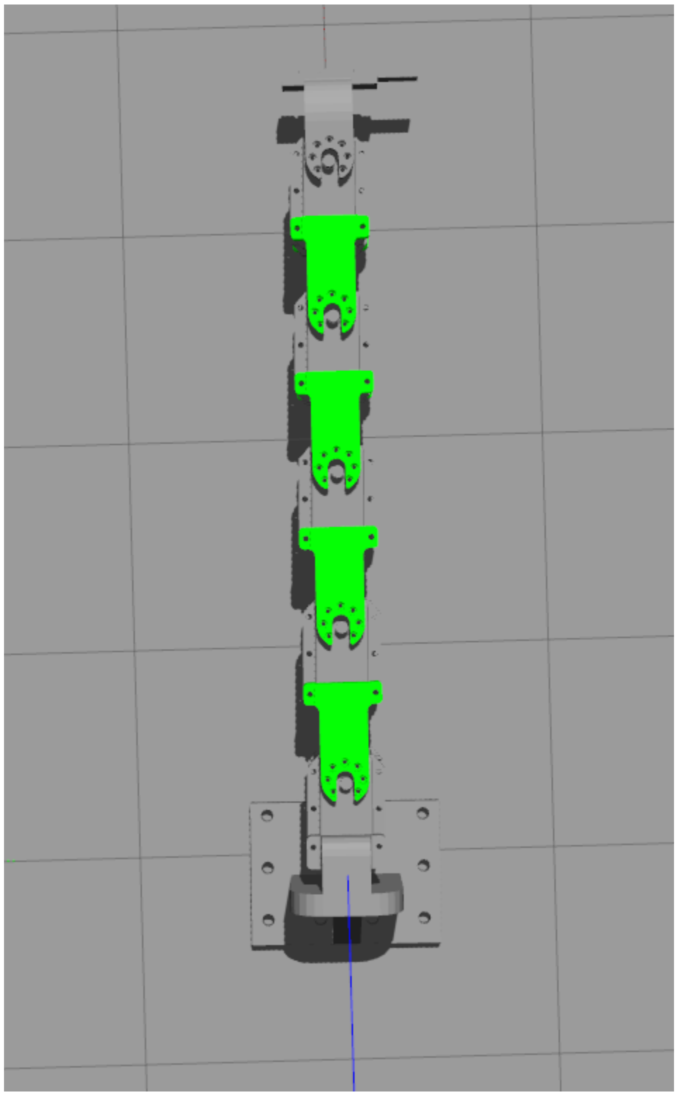


Fig. 13. Init Pose in Gazebo

V. Debugging

Controllers.yaml file was added to config folder with the following code:

```
controller_list:
```

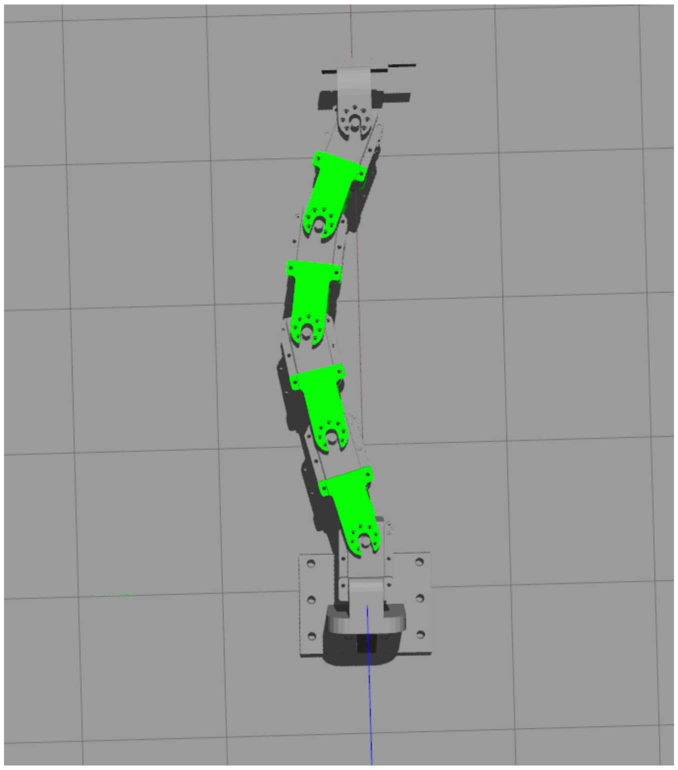
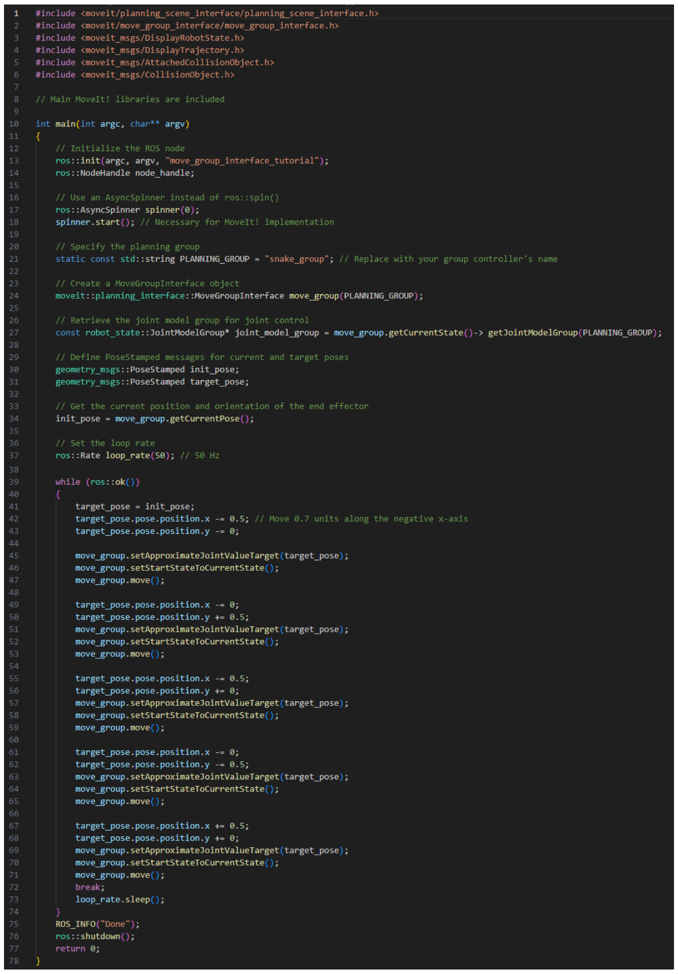


Fig. 14. End-effector moved by 1.4 units



```cpp
#include <moveit/planning_scene_interface/planning_scene_interface.h>
#include <moveit/move_group_interface/move_group_interface.h>
#include <moveit_msgs/DisplayRobotState.h>
#include <moveit_msgs/DisplayTrajectory.h>
#include <moveit_msgs/AttachedCollisionObject.h>
#include <moveit_msgs/CollisionObject.h>

// Main MoveIt! libraries are included

int main(int argc, char** argv)
{
    // Initialize the ROS node
    ros::init(argc, argv, "move_group_interface_tutorial");
    ros::NodeHandle node_handle;

    // Use an AsyncSpinner instead of ros::spin()
    ros::AsyncSpinner spinner(0);
    spinner.start(); // Necessary for MoveIt! implementation

    // Specify the planning group
    static const std::string PLANNING_GROUP = "snake_group"; // Replace with your group controller's name

    // Create a MoveGroupInterface object
    moveit::planning_interface::MoveGroupInterface move_group(PLANNING_GROUP);

    // Retrieve the joint model group for joint control
    const robot_state::JointModelGroup* joint_model_group = move_group.getCurrentState()-> getJointModelGroup(PLANNING_GROUP);

    // Define PoseStamped messages for current and target poses
    geometry_msgs::PoseStamped init_pose;
    geometry_msgs::PoseStamped target_pose;

    // Get the current position and orientation of the end effector
    init_pose = move_group.getCurrentPose();

    // Set the loop rate
    ros::Rate loop_rate(50); // 50 Hz

    while (ros::ok())
    {
        target_pose = init_pose;
        target_pose.pose.position.x -= 0.5; // Move 0.7 units along the negative x-axis
        target_pose.pose.position.y -= 0;

        move_group.setApproximateJointValueTarget(target_pose);
        move_group.setStartStateToCurrentState();
        move_group.move();

        target_pose.pose.position.x -= 0;
        target_pose.pose.position.y += 0.5;
        move_group.setApproximateJointValueTarget(target_pose);
        move_group.setStartStateToCurrentState();
        move_group.move();

        target_pose.pose.position.x -= 0.5;
        target_pose.pose.position.y += 0;
        move_group.setApproximateJointValueTarget(target_pose);
        move_group.setStartStateToCurrentState();
        move_group.move();

        target_pose.pose.position.x -= 0;
        target_pose.pose.position.y -= 0.5;
        move_group.setApproximateJointValueTarget(target_pose);
        move_group.setStartStateToCurrentState();
        move_group.move();

        target_pose.pose.position.x += 0.5;
        target_pose.pose.position.y += 0;
        move_group.setApproximateJointValueTarget(target_pose);
        move_group.setStartStateToCurrentState();
        move_group.move();
        break;
        loop_rate.sleep();
    }
    ROS_INFO("Done");
    ros::shutdown();
    return 0;
}
```

Fig. 15. Move the end-effector in rectangle

```
    - name: robot/group1_controller
      action_ns: follow_joint_trajectory
      type: FollowJointTrajectory
      default: true
      joints:
      - motortom2m
      - joint2
      - joint4
      - joint6

    - name: robot/group2_controller
      action_ns: follow_joint_trajectory
      type: FollowJointTrajectory
      default: true
      joints:
      - end
```

Also, a new file launch file was written moveit_planning_execution2.launch:

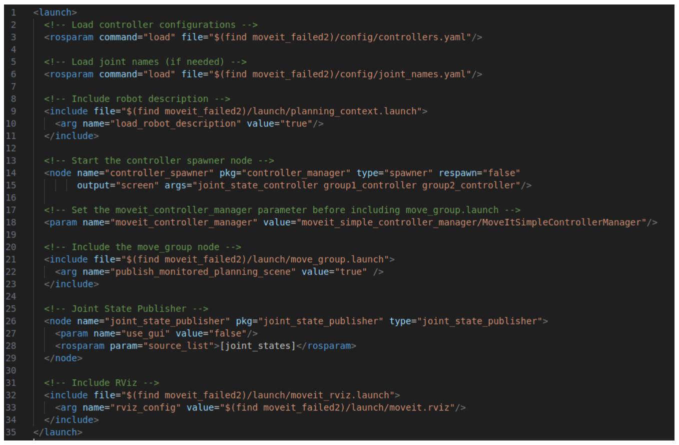
```
controller_list:
    - name: robot/group1_controller
      action_ns: follow_joint_trajectory
      type: FollowJointTrajectory
      default: true
      joints:
      - motortom2m
      - joint2
      - joint4
      - joint6

    - name: robot/group2_controller
      action_ns: follow_joint_trajectory
      type: FollowJointTrajectory
      default: true
      joints:
      - end
```

Also, the launch file in Fig. 16 was written. The one given in slides doesn't work.

VI. Conclusion

In conclusion, this lab provided an in-depth exploration of configuring and controlling a snake robot with Dynamixel motors, emphasizing both MoveIt setup and trajectory execution. Starting with the MoveIt Setup Assistant, we tailored a package that allowed precise control of the robot's kinematics and path planning. The lab progressed through essential configurations, including URDF customization, self-collision generation, and planning group definitions, which together laid the groundwork for effective joint coordination and trajectory control.



```xml
<launch>
  <!-- Load controller configurations -->
  <rosparam command="load" file="$(find moveit_failed2)/config/controllers.yaml"/>

  <!-- Load joint names (if needed) -->
  <rosparam command="load" file="$(find moveit_failed2)/config/joint_names.yaml"/>

  <!-- Include robot description -->
  <include file="$(find moveit_failed2)/launch/planning_context.launch">
    <arg name="load_robot_description" value="true"/>
  </include>

  <!-- Start the controller spawner node -->
  <node name="controller_spawner" pkg="controller_manager" type="spawner" respawn="false"
        output="screen" args="joint_state_controller group1_controller group2_controller"/>

  <!-- Set the moveit_controller_manager parameter before including move_group.launch -->
  <param name="moveit_controller_manager" value="moveit_simple_controller_manager/MoveItSimpleControllerManager"/>

  <!-- Include the move_group node -->
  <include file="$(find moveit_failed2)/launch/move_group.launch">
    <arg name="publish_monitored_planning_scene" value="true" />
  </include>

  <!-- Joint State Publisher -->
  <node name="joint_state_publisher" pkg="joint_state_publisher" type="joint_state_publisher">
    <param name="use_gui" value="false"/>
    <rosparam param="source_list">[joint_states]</rosparam>
  </node>

  <!-- Include RViz -->
  <include file="$(find moveit_failed2)/launch/moveit_rviz.launch">
    <arg name="rviz_config" value="$(find moveit_failed2)/launch/moveit.rviz"/>
  </include>
</launch>
```

Fig. 16. Init Pose in Gazebo

Beyond setup, we implemented joint trajectory controllers to ensure smooth, segmented motion, and tested the robot's capability to follow complex paths through defined via points and inverse kinematics calculations. This process highlights the adaptability and precision of ROS-based control frameworks, enabling the robot to perform advanced movements suited to dynamic environments. Overall, the lab reinforced key concepts in modular robotics and real-time motion planning, demonstrating the practical potential of the snake robot in applications requiring flexibility and high precision.

References

[1] B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo, *Robotics: Modelling, Planning and Control*. Springer Science & Business Media, 2010.

[2] M. W. Spong, S. Hutchinson, and M. Vidyasagar, *Robot Modeling and Control*. John Wiley & Sons, 2006.