Robotic Arm Control based on ROS via EtherCat

Ju Wu

June 2021

Contents

1 Problem Statements

In this semester project, I controlled the robotic arm modeled by computer aided design software SOLIDWORK&CATIA based on Robotic Operation System (ROS). I first converted the model in CAD into unified robot description format (urdf) file via configuring parent&child links and their coordination positions and orientations. The oriental constraints of each coordination can be settled in the urdf file in advance alongside the conversion or they can be set alongside MoveIt inverse kinematics solver and Cartesian path planner. I designed the function to output a series of joint states computed by inverse kinematics solver given the desired pose of end-effector of the robotic arm in a given rate and designed the function to generate interpolated Cartesian path with the given waypoints. Then I studied the motors placed in the joints of the robotic arm, micro-controllers to regulate the motors and the EtherCat communication protocol between Nvidia Jetson Xavier Development Kit (Jetson) the micro-controllers. Finally I propose the promising tests and improvement directions.

2 Objectives

In this part, I set the objectives of the project as follows.

- modelling and visualization of the robotic arm in ROS: XML language is used to describe the model of robotic arm and then the urdf model is imported into rviz for visualization.
- Inverse kinematics solver: provided desired pose of the end-effector of the robotic arm, the inverse kinematics solver function is designed to compute the corresponding path from pose A to B in joint space with specified interval and space&joints constraints, and both the real and simulated controllers can take the path to drive the gripper of the robotic arm to reach the desired pose.
- Cartesian trajectory generator: given way-points with detailed positions, velocities, accelerations and efforts and specifications like time constraints and limits of rotational angle against certain axis, the generated *trajectory_msgs::JointTrajectory* is obtained to guide the robotic arm move in the space with the desired interpolated path.
- motor control with extant hardware components and control schemes: I configure, calibrate and test the the maxon brushed DC motor with EPOS4 3-axis micro-controller in CSP and CSV modes in EPOS Studio and then let the reference input of the control schemes get the computed desired joint states in a certain interval.

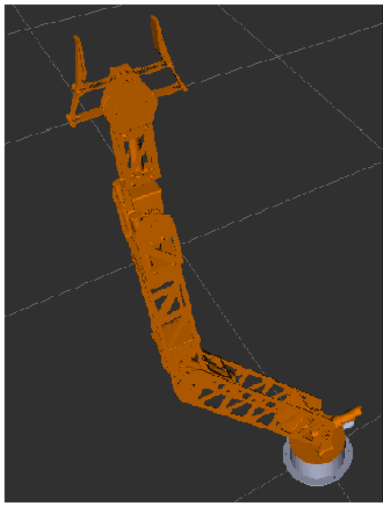
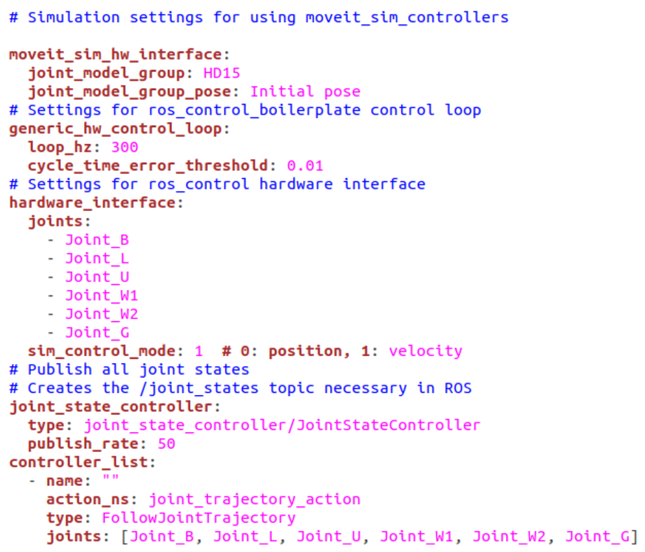
	
robotic arm model rendered in CATIA	robotic arm visualized in rviz

Table 1: Visualization of two models of the robotic arm

3 Simulators

In this section, I first load the urdf model of robotic arm into ROS and visualize it with rviz as shown in Table. 1. To simulate the motions of the model in ROS with *joint_state_publisher* and *joint_state_publisher_gui* and prepare for motion planning, I'll assume that you already have ROS melodic installed on Ubuntu 18.04 and setup on your machine. The robotic arm is 6-DoF and decomposed into [bask_link, Link_B, Link_G, Link_L, Link_U, Link_W_1, Link_W_2] from bottom to top, the joint of link is regarded as the interconnected position of this link and its parent link so that the joint space is [Joint_B, Joint_G, Joint_L, Joint_U, Joint_W_1, Joint_W_2]. And it is essential to create an additional virtual link called *world* connected with *base_link*, otherwise the model can not appear on rviz normally. And the following commands are necessary the first time.

- source /opt/ros/melodic/setup.bash.
- sudo apt-get install ros-melodic-moveit.
- sudo apt-get install ros-melodic-moveit-visual-tools.
- sudo apt-get install ros-melodic-rviz-visual-tools.
- git clone -b melodic-devel https://github.com/ros-industrial-consortium/descartes.git
- sudo apt-get install ros-melodic-industrial-core



```
# Simulation settings for using moveit_sim_controllers

moveit_sim_hw_interface:
  joint_model_group: HD15
  joint_model_group_pose: Initial pose
# Settings for ros_control_boilerplate control loop
generic_hw_control_loop:
  loop_hz: 300
  cycle_time_error_threshold: 0.01
# Settings for ros_control hardware interface
hardware_interface:
  joints:
    - Joint_B
    - Joint_L
    - Joint_U
    - Joint_W1
    - Joint_W2
    - Joint_G
  sim_control_mode: 1  # 0: position, 1: velocity
# Publish all joint states
# Creates the /joint_states topic necessary in ROS
joint_state_controller:
  type: joint_state_controller/JointStateController
  publish_rate: 50
controller_list:
  - name: ""
    action_ns: joint_trajectory_action
    type: FollowJointTrajectory
    joints: [Joint_B, Joint_L, Joint_U, Joint_W1, Joint_W2, Joint_G]
```

Figure 1: Configuration of the model and simulation settings

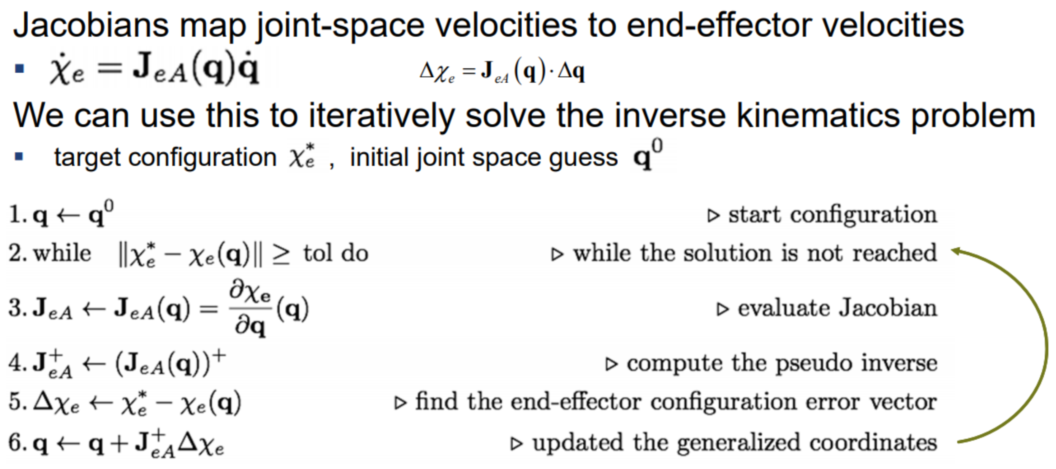
3.1 Motion Planning

In this section, I begin from a simple inverse kinematics solver to compute the joint states trajectory from the initial pose to the desired pose and then demonstrate Cartesian path generators in virtue of interpolation given the way-points and the transformation from Cartesian path to joint space trajectory.

3.1.1 Inverse Kinematics

I first show the general numerical method to derive the inverse kinematics given the desired pose of end-effector as shown in Fig. 2. The inverse kinematics problem can be viewed as finding roots of a nonlinear equation: $T(\theta) = X$, many numerical methods exist for finding roots of nonlinear equations. For inverse kinematics problem, the target configuration $X \in SE(3)$ is a homogeneous matrix. We need to modify the standard root finding methods, but the main idea is the same.

The numerical method for inverse kinematics begins with initial joint space guess q^0. Then given the Jacobians map from joint-space velocities to end-effector velocities, if the error between target end-effector configuration and the estimated one larger than the tolerance, the current Jacobians against joint space is evaluated, its pseudo inverse and end-effector configuration error vector have to be computed and the generalized coordinates of joint space are updated in virtue of Newton-Raphson method. A solution will be obtained if the error is within the predefined tolerance. Thus there may be multiple solutions and the algorithm tends to converge to the solution that is the closest to the initial guess.


Jacobians map joint-space velocities to end-effector velocities

- $\dot{\chi}_e = \mathbf{J}_{eA}(\mathbf{q})\dot{\mathbf{q}}$ $\Delta\chi_e = \mathbf{J}_{eA}(\mathbf{q}) \cdot \Delta\mathbf{q}$

We can use this to iteratively solve the inverse kinematics problem

- target configuration χ_e^* , initial joint space guess $\mathbf{q}^0$

```
1. $\mathbf{q} \leftarrow \mathbf{q}^0$                                          ▷ start configuration
2. while $\|\chi_e^* - \chi_e(\mathbf{q})\| \geq$ tol do                       ▷ while the solution is not reached
3. $\mathbf{J}_{eA} \leftarrow \mathbf{J}_{eA}(\mathbf{q}) = \frac{\partial \chi_e}{\partial \mathbf{q}}(\mathbf{q})$    ▷ evaluate Jacobian
4. $\mathbf{J}_{eA}^+ \leftarrow (\mathbf{J}_{eA}(\mathbf{q}))^+$              ▷ compute the pseudo inverse
5. $\Delta\chi_e \leftarrow \chi_e^* - \chi_e(\mathbf{q})$                     ▷ find the end-effector configuration error vector
6. $\mathbf{q} \leftarrow \mathbf{q} + \mathbf{J}_{eA}^+ \Delta\chi_e$          ▷ updated the generalized coordinates
```

Figure 2: Numerical method to derive the desired inverse kinematics

3.1.2 Direct MoveIt Usage

In this part, I configure the inverse kinematic solver for our robotic arm. I visualize the model of robotic arm in *rviz* and employ the *moveit* package to set up kinematics solver as the follows.

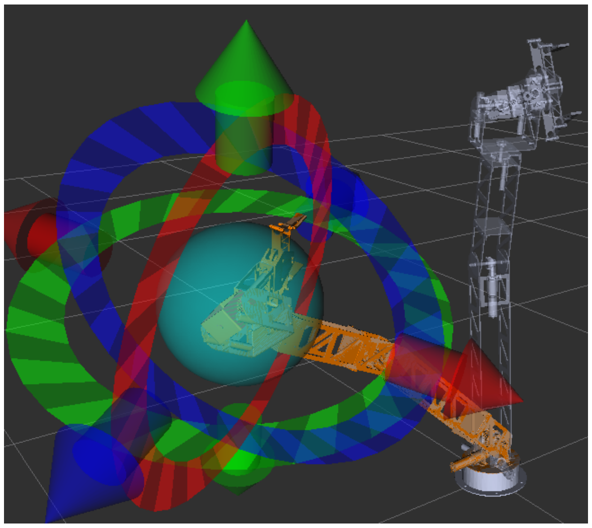
- Kinematics solver: *cached_ik_kinematics_plugin/CachedKDLKinematicsPlugin*
- Kinematics solver search resolution: considering that this specifies the resolution that a solver might use to search over the redundant space for inverse kinematics, e.g. one of the joints in this case is specified as the redundant joint, I select the value as 0.005 for the configuration.
- kinematics solver timeout: considering this is a default timeout specified in seconds for each internal iteration that the inverse kinematics solver may perform, I select the value as $0.005s$.
- OMPL: I select *SBL* motion planning algorithm for the robotic arm.

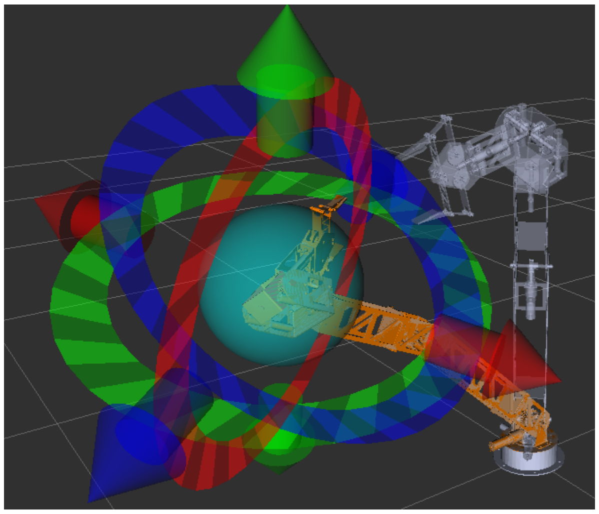
I hereby generate the inverse kinematics and motion planning packages for our robotic arm. To test the packages, I load our robotic arm in *rviz* and set the initial and goal positions as the follows.

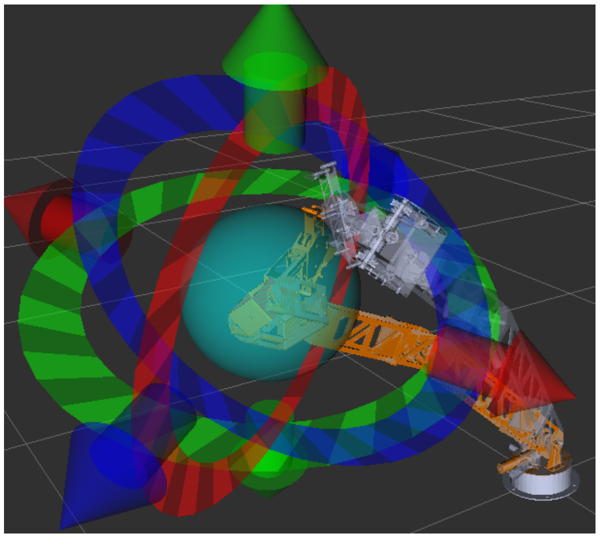
Poses of the robotic arm	Joint positions(rad)					
	Joint_B	Joint_L	Joint_U	Joint_W$_1$	Joint_W$_2$	Joint_G
Initial pose	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
Goal pose	1.61381	-0.80269	0.31239	0.88396	-2.25541	0.69556

Table 2: Joint positions of the initial and goal poses

I demonstrate the snapshot series in which the robotic arm moves from initial pose to the goal pose.


Snapshot #1


Snapshot #2


Snapshot #3

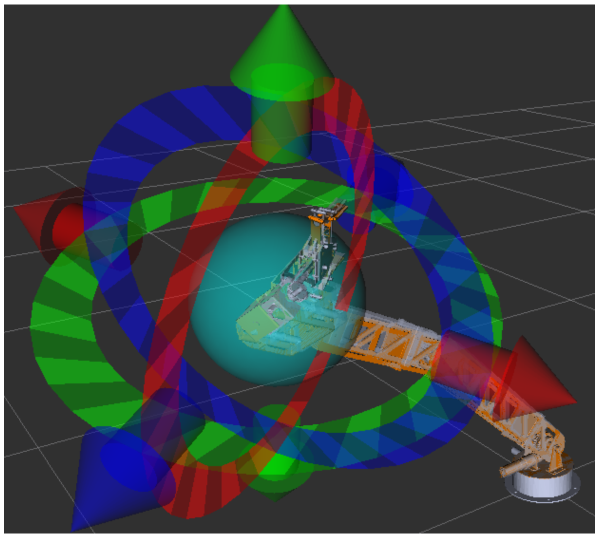

Snapshot #4

Table 3: Snapshot series of the robotic arm motion from the initial to goal pose

The communication relationship is shown in Fig. 4. The */move_group_cmd* is the designed node that subscribes the topic */direction* containing pose of the end-effector and transforms the desired pose as the format of */execute_trajectory/goal*. Then the */move_group* implements the configured inverse kinematics solver and publishes the joint space trajectory in 10Hz to the topic */move_group/fake_controller_joint_states*.

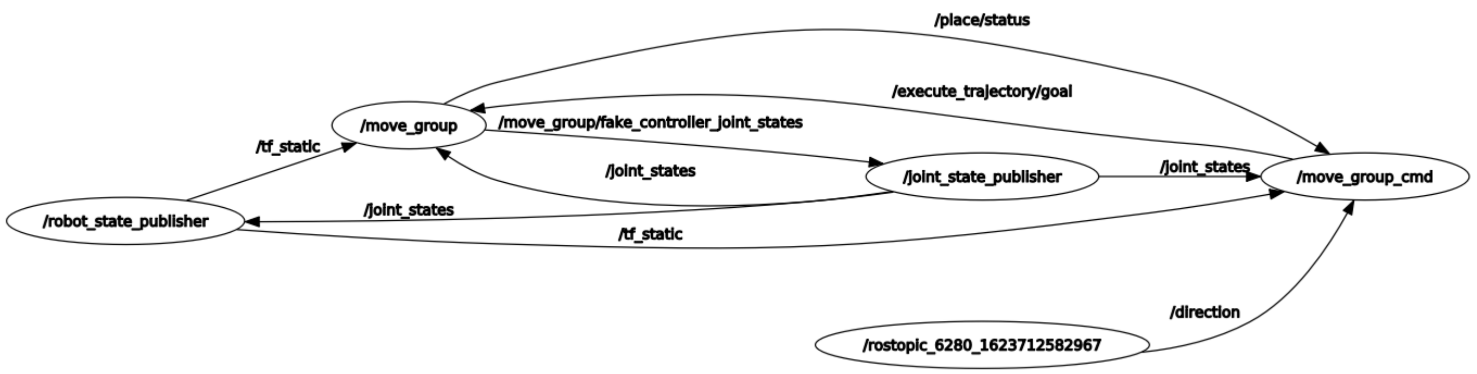


Figure 3: Communication relationship of MoveIt inverse kinematics solver nodes

To satisfy the requirement of control schemes of our motors, the desired commands have to be sent to the topic in a desired constant rate especially synchronized with other components of the system. Considering that the commands for the motors are sent in 10Hz, We need to subscribe the desired joint states in 10Hz as well. To enable the real-time absolute position tracking, the topic */joint_states* is used instead of */move_group/fake_controller_joint_states*. I set the subscriber's rate as 10Hz and increase the queue length to 1000, and then I observed that */joint_states* has the same rate as */move_group/fake_controller_joint_states* and without message loss shown in .

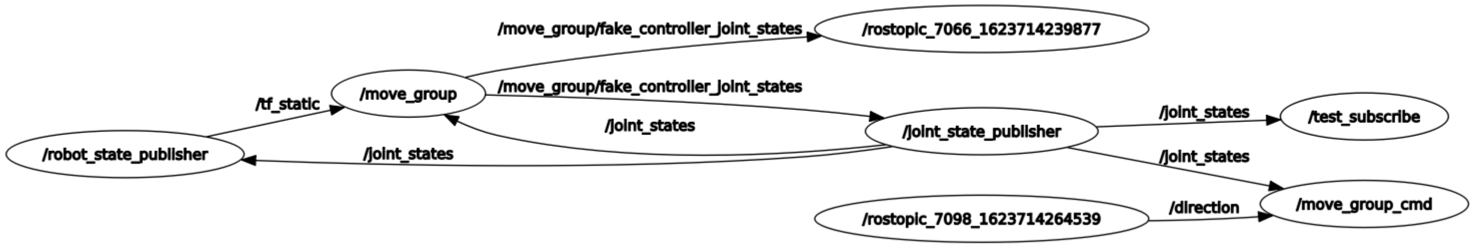
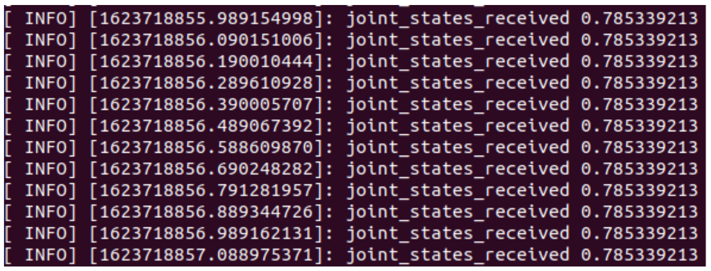


Figure 4: Communication relationship of MoveIt inverse kinematics solver nodes with additional test subscriber



```
[ INFO] [1623718855.989154998]: joint_states_received 0.785339213
[ INFO] [1623718856.090151006]: joint_states_received 0.785339213
[ INFO] [1623718856.190010444]: joint_states_received 0.785339213
[ INFO] [1623718856.289610928]: joint_states_received 0.785339213
[ INFO] [1623718856.390005707]: joint_states_received 0.785339213
[ INFO] [1623718856.489067392]: joint_states_received 0.785339213
[ INFO] [1623718856.588609870]: joint_states_received 0.785339213
[ INFO] [1623718856.690248282]: joint_states_received 0.785339213
[ INFO] [1623718856.791281957]: joint_states_received 0.785339213
[ INFO] [1623718856.889344726]: joint_states_received 0.785339213
[ INFO] [1623718856.989162131]: joint_states_received 0.785339213
[ INFO] [1623718857.088975371]: joint_states_received 0.785339213
```

Figure 5: Positions of Joint_B in /joint_states to verify 10Hz subscription rate

Cartesian Path Planning

In this part, I generate customized Cartesian path as needed and combine it with MoveIt inverse kinematics solver to plan and simulate trajectory in joint space. The functions of Descartes package in ROS are as follows.

- descartes_core: The descartes_core package defines the interfaces for trajectory points, robot_models, and planners. See the docs for more information about the APIs associated with these concepts.
- descartes_planner: The descartes_planner package contains reference planner implementations that can provide solutions to trajectories using different algorithms. More details at descartes_planner.

- descartes_trajectory: The descartes_trajectory package contains reference implementations for common types of trajectory points such as points defining a certain Cartesian pose or a certain joint configuration. More details at descartes_trajectory.

And Descartes, as a path planning library, is often compared to MoveIt. However, it differs from MoveIt in several key ways that can be demonstrated in the following parts:

- Cartesian Planning: While MoveIt plans free space motion (i.e. move from A to B), Descartes plans robot joint motion for semi-constrained Cartesian paths (i.e. ones whose waypoints may have less than a fully specified 6DOF pose).
- Efficient, Repeatable, Scale-able Planning: The paths that result from Descartes appear very similar to human generated paths, but without all the effort. The plans are also repeatable and scale-able to the complexity of the problem (easy paths are planned very quickly, hard paths take time but are still solved).
- Dynamic Re-planning: Path planning structures are maintained in memory after planning is complete. When changes are made to the desired path, an updated robot joint trajectory can be generated nearly instantaneously.
- Offline Planning: Similar to MoveIt, but different than other planners, Descartes is primarily focused on offline or sense/plan/act applications. Real-time planning is not a feature of Descartes.

I first demonstrate the theoretical background of Cartesian path planning. The general diagram of trajectory planner interfaces is shown in Fig. 6, in which the robot action as the input of trajectory planner is described as a sequence of poses or configurations, and reference profile/values (continuous or discrete) for the robot controller are the output of the trajectory planner.

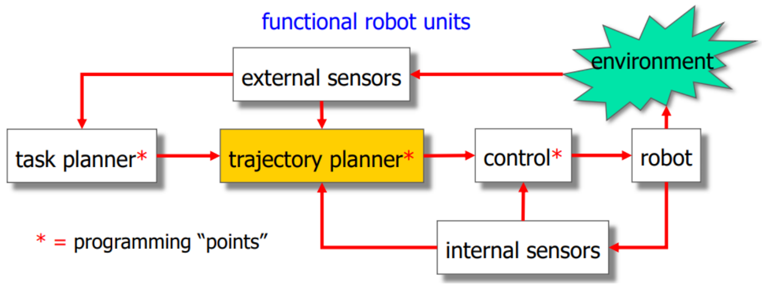


Figure 6: General diagram of trajectory planner interfaces

As demonstrated in Fig. 7, the trajectories of Cartesian space and joint space can be transformed mutually. And as shown in Fig. 8, sequence of way-points are first defined in Cartesian space, and then it can be converted into geometric path in joint space in virtue of interpolation, path sampling and inverse kinematics solver.

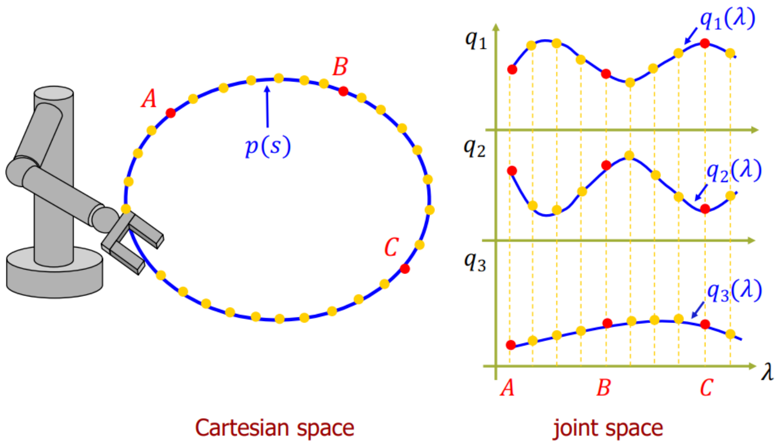


Figure 7: Comparison between Cartesian space and joint space

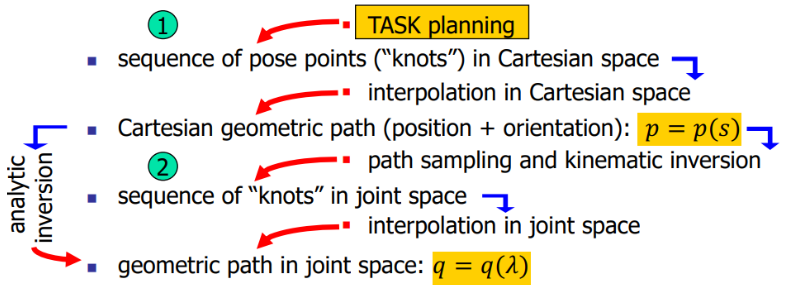
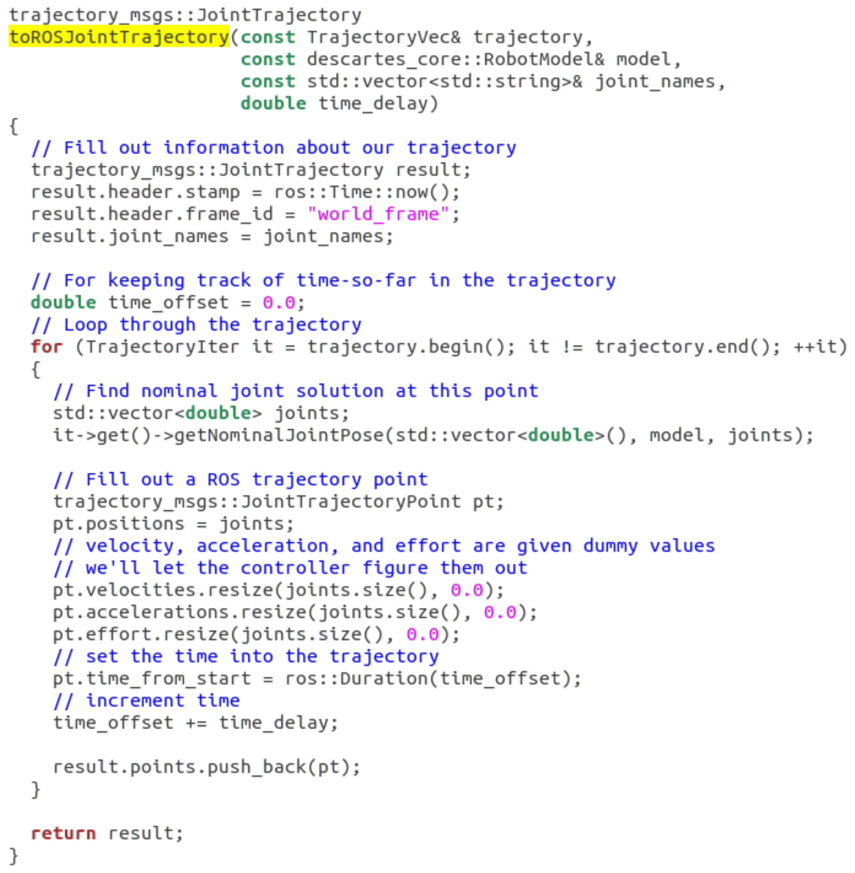

1
TASK planning
sequence of pose points ("knots") in Cartesian space
interpolation in Cartesian space
Cartesian geometric path (position + orientation): $p = p(s)$
2
path sampling and kinematic inversion
sequence of "knots" in joint space
interpolation in joint space
geometric path in joint space: $q = q(\lambda)$
analytic inversion

Figure 8: Transformation of trajectories from Cartesian space to joint space

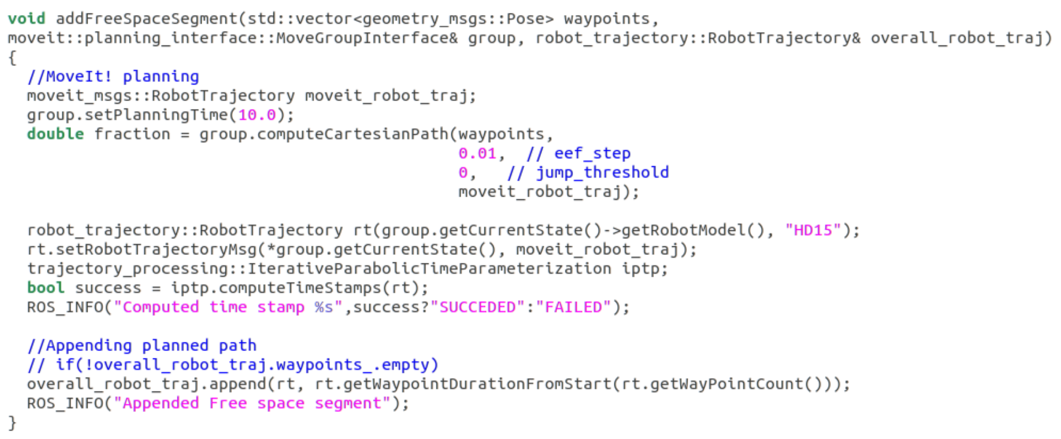
With functions shown in Fig. 9, I can convert the constrained Cartesian way-points defined in *descartes_core::TrajectoryPtPtr* into *trajectory_msgs::JointTrajectory* defined in joint space. And with the help of functions shown in Fig. 10, I can use parabolic interpolation to add a segment of free space path to the original *robot_trajectory::RobotTrajectory* path. I do experiment to convert path *descartes_core::TrajectoryPtPtr* of length 12 into trajectory in joint space with cost 4.026 and add it to *robot_trajectory::RobotTrajectory* path.



```
trajectory_msgs::JointTrajectory
toROSJointTrajectory(const TrajectoryVec& trajectory,
                     const descartes_core::RobotModel& model,
                     const std::vector<std::string>& joint_names,
                     double time_delay)
{
  // Fill out information about our trajectory
  trajectory_msgs::JointTrajectory result;
  result.header.stamp = ros::Time::now();
  result.header.frame_id = "world_frame";
  result.joint_names = joint_names;

  // For keeping track of time-so-far in the trajectory
  double time_offset = 0.0;
  // Loop through the trajectory
  for (TrajectoryIter it = trajectory.begin(); it != trajectory.end(); ++it)
  {
    // Find nominal joint solution at this point
    std::vector<double> joints;
    it->get()->getNominalJointPose(std::vector<double>(), model, joints);

    // Fill out a ROS trajectory point
    trajectory_msgs::JointTrajectoryPoint pt;
    pt.positions = joints;
    // velocity, acceleration, and effort are given dummy values
    // we'll let the controller figure them out
    pt.velocities.resize(joints.size(), 0.0);
    pt.accelerations.resize(joints.size(), 0.0);
    pt.effort.resize(joints.size(), 0.0);
    // set the time into the trajectory
    pt.time_from_start = ros::Duration(time_offset);
    // increment time
    time_offset += time_delay;

    result.points.push_back(pt);
  }

  return result;
}
```

Figure 9: Function that convert constrained Cartesian way-points into path defined in joint space

The communication relationship of Descartes path planning nodes is shown in Fig. 11, we can observe that in this case */move_group* subscribes the topic */execute_trajectory/goal* from */descartes_path*, and */descartes_path* subscribes the topic */tf_static* from */robot_state_publisher* to facilitate path planning.



```
void addFreeSpaceSegment(std::vector<geometry_msgs::Pose> waypoints,
moveit::planning_interface::MoveGroupInterface& group, robot_trajectory::RobotTrajectory& overall_robot_traj)
{
  //MoveIt! planning
  moveit_msgs::RobotTrajectory moveit_robot_traj;
  group.setPlanningTime(10.0);
  double fraction = group.computeCartesianPath(waypoints,
                                               0.01,  // eef_step
                                               0,   // jump_threshold
                                               moveit_robot_traj);

  robot_trajectory::RobotTrajectory rt(group.getCurrentState()->getRobotModel(), "HD15");
  rt.setRobotTrajectoryMsg(*group.getCurrentState(), moveit_robot_traj);
  trajectory_processing::IterativeParabolicTimeParameterization iptp;
  bool success = iptp.computeTimeStamps(rt);
  ROS_INFO("Computed time stamp %s",success?"SUCCEDED":"FAILED");

  //Appending planned path
  // if(!overall_robot_traj.waypoints_.empty)
  overall_robot_traj.append(rt, rt.getWaypointDurationFromStart(rt.getWayPointCount()));
  ROS_INFO("Appended Free space segment");
}
```

Figure 10: Function that converts pose way-points into path defined in joint space

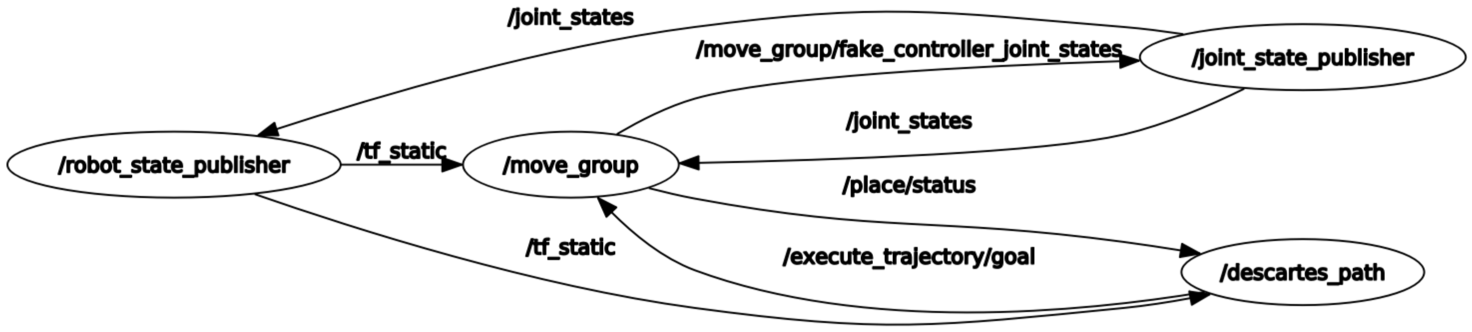
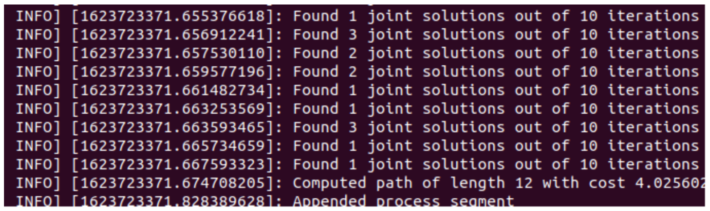


Figure 11: Communication relationship of Descartes path planning nodes



```
INFO] [1623723371.655376618]: Found 1 joint solutions out of 10 iterations
INFO] [1623723371.656912241]: Found 3 joint solutions out of 10 iterations
INFO] [1623723371.657530110]: Found 2 joint solutions out of 10 iterations
INFO] [1623723371.659577196]: Found 2 joint solutions out of 10 iterations
INFO] [1623723371.661482734]: Found 1 joint solutions out of 10 iterations
INFO] [1623723371.663253569]: Found 1 joint solutions out of 10 iterations
INFO] [1623723371.663593465]: Found 3 joint solutions out of 10 iterations
INFO] [1623723371.665734659]: Found 1 joint solutions out of 10 iterations
INFO] [1623723371.667593323]: Found 1 joint solutions out of 10 iterations
INFO] [1623723371.674708205]: Computed path of length 12 with cost 4.025602
INFO] [1623723371.828389628]: Appended process segment
```

Figure 12: Communication relationship of Descartes path planning nodes

3.2 Discussion

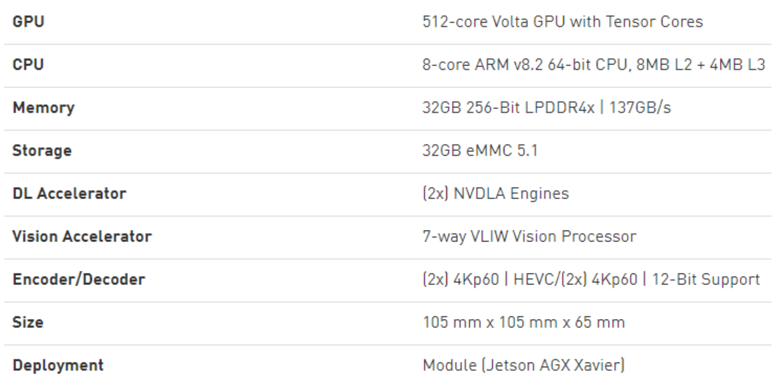
As responses to the objectives, I have modelled and visualized the robotic in ROS and simulated its motions with *joint_state_publisher* and *joint_state_publisher_gui*. And two path planning methods (task space and joint space) are achieved in ROS properly as well. And constraints such as time interval of two neighboring points and toleranced frames of points can be configured according to practical requirements to facilitate generate customized paths. However, considering that there are non-negligible differences between the real system and the simulated one, e.g., inertia matrix, joint limits, velocity and acceleration limits, settling time of control of the motors, it is important to first assemble the whole robotic arm and test the simplest control scheme, i.e., employ the desired joint states constantly dispatched as reference positions with CSP mode in EPOS4, and then discuss the impact of motors' settling time and sampling frequency of commands. Upon the communication frequency is tuned, Cartesian path generators and corresponding advanced control schemes that considers values of velocity and acceleration will be tested as well to improve the smoothness of motions.

4 Hardware Components

In this section, we introduce the hardware platform we use, their characteristics and promising experiments and tests after integration.

4.1 Nvidia Jetson Xavier

In our competition, Nvidia Jetson Xavier is used and its basic specification information is shown in Fig. 13



GPU	512-core Volta GPU with Tensor Cores
CPU	8-core ARM v8.2 64-bit CPU, 8MB L2 + 4MB L3
Memory	32GB 256-Bit LPDDR4x | 137GB/s
Storage	32GB eMMC 5.1
DL Accelerator	(2x) NVDLA Engines
Vision Accelerator	7-way VLIW Vision Processor
Encoder/Decoder	(2x) 4Kp60 | HEVC/(2x) 4Kp60 | 12-Bit Support
Size	105 mm x 105 mm x 65 mm
Deployment	Module (Jetson AGX Xavier)

Figure 13: Basic specification information of Jetson Xavier

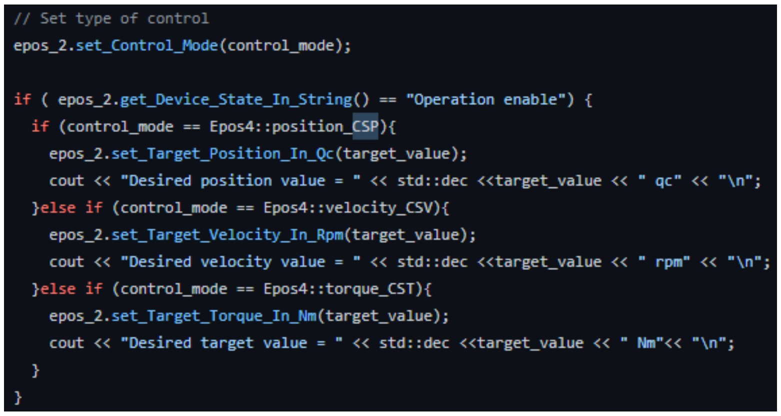
And Jetson Xavier Module Interface are comprised of x8 PCIe Gen4/×8 SLVS-EC, Gigabit Ethernet, 2×USB 3.1, NVMe, PCIe×1+USB 2.0+UART (for Wi-Fi/LTE)/I^2S/PCM, UART+SPI+CAN+I^2C+I^2S+DMIC+GPIOs, High-Definition Audio, SATA Through PCIe×1 Bridge (PD+Data for 2.5-inch SATA)+USB 3.0, HDMI 2.0, SD/UFS.

4.2 Maxon Motor&Micro-controller

The maxon brushed DC motors with normal voltage 24V are employed, and they are powered by the EPOS4 3-axis micro-controllers. SSI absolute encoders are used to record velocity and position of motor. EPOS Studio is used to configure, calibrate and test the motors.

5 EtherCat Communication Protocol

In this section, we introduce the EtherCat communication protocol we employ to establish communication between Jetson and EPOS4. In our case, the Simple Open EtherCAT Master Library (SOEM) is employed to achieve the master (host PC) and slave (EPOS4) EtherCat communication Multiple control mode like cyclic synchronous velocity and cyclic synchronous position modes are configured. The JointState message is subscribed by main function, and its position values are conveyed to EPOS as command for CSP modes. And CSP/CSV/CST modes are called as shown in Fig. 14.



```
// Set type of control
epos_2.set_Control_Mode(control_mode);

if ( epos_2.get_Device_State_In_String() == "Operation enable") {
  if (control_mode == Epos4::position_CSP){
    epos_2.set_Target_Position_In_Qc(target_value);
    cout << "Desired position value = " << std::dec <<target_value << " qc" << "\n";
  }else if (control_mode == Epos4::velocity_CSV){
    epos_2.set_Target_Velocity_In_Rpm(target_value);
    cout << "Desired velocity value = " << std::dec <<target_value << " rpm" << "\n";
  }else if (control_mode == Epos4::torque_CST){
    epos_2.set_Target_Torque_In_Nm(target_value);
    cout << "Desired target value = " << std::dec <<target_value << " Nm"<< "\n";
  }
}
```

Figure 14: Illustration of call of three control modes

6 Conclusion

There are promising tests and experiments upon integration of components that are related to content of the project can help the work improved and developed further by other members in the future :

- Test if the arm's algorithm allows stopping when hitting an object: place an obstacle in the arm's trajectory and see if arm detects excessive torque applied to motor to stop action, start by performing for each joint then the full arm assembly.
- Validate cable management of the arm: perform maximum range of motion for each joint on the arm and observe if any cables come undone/loose
- Test the time required to perform an action: test the different path generation algorithms present to determine optimal one as well as time required to perform an action (ex: go from point A to point B).
- Test the arm's precision at the end effector: place a plate with a position marker identifying the desired position (which is known relative to the arm) and let the arm to go to that position multiple times, and then observe the errors.
- Test the soil sampling mechanism of the gripper (scoops): attach gripper to sliding test-bench (manually), and actuate gripper motor to try collecting soil samples of different types with the gripper.
- Cascade objects detection and arm reaching: perform target detection and approaching with the closed loop and have the processed camera feed as an command for robotic arm motions.
- Test the arm during a simulation of science sampling and probing task in real-life conditions: perform complete simulation of the probing and science sampling tasks and evaluate the obtained score according to the rule-book.

And the above potential tests indicate promising directions that need to be improved, e.g., consideration of static and dynamic obstacles, automatic collision detection without camera, integration of objects detection and approaching, modelling and control of gripper in ROS.