ROS 2 exam

2h, documents / internet allowed (closed chats please)

1 Description

In this exam you will have to write a C++ node and run it two times through a launch file.

The package should first be compiled, then the simulation can be run once and for all with:

```
ros2 launch ecn_exam_2022 simulation_launch.py
```

This simulation involves the Baxter robot. The goal is to move the two arms by giving velocity commands on the end-effectors.

1.1 Controlling the robot

As in the labs, the simulation subscribes to:

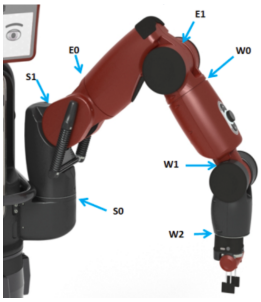
- `/robot/limb/left/joint_command`
- `/robot/limb/right/joint_command`

on these topics, expected messages are of type `baxter_core_msgs/msg/JointCommand`:

```
int32 mode # control mode (should be set to VELOCITY_MODE here)
float64[] command # command (position or velocity)
string[] names # names of the controlled joints

int32 POSITION_MODE =1
int32 VELOCITY_MODE =2
int32 TORQUE_MODE =3
int32 RAW_POSITION_MODE =4
```

Depending on the arm that is controlled, the names of the joints are given below:



joints (`left_` or `right_`)	`s0`	`s1`	`e0`	`e1`	`w0`	`w1`	`w2`

1.2 The Jacobian services

Each arm advertizes a service to get the current Jacobian:

- `/robot/limb/left/jacobian`
- `/robot/limb/right/jacobian`

These services are of type `baxter_simple_sim/srv/Jacobian`:

```
float64[] position # joint position to compute the Jacobian, current if empty
bool inverse false # if the returned value should be the Jacobian inverse (true in our case)
bool ee_frame false # if the Jacobian should be expressed in the end-effector frame (true in our case)
---
float64[42] jacobian # the 42 coefficients of the Jacobian (or inverse) matrix, row-major
```

1.3 Publishing velocity setpoints

Velocity setpoints can be published as `geometry_msgs/msg/Twist` through this launch file:

```
ros2 launch ecn_exam_2022 slider_launch.py
```

2 Tips

Compile the code once using `colbuild --packages-select ecn_exam_2022`. Then, use `gqt` in the package folder to configure QtCreator. Do not forget to use it with ROS 2 setup.

In order to efficiently debug your code, it is strongly advised to do the initial development with hard-coded values. This node should do all necessary things with hard-coded parameters and topics.
You can then make it generic and run it through the launch file.

Feel free to have a look at the `basic_node.cpp` from the lab templates.

Declaring node parameters is detailed in online tutorials and in my slides.

You can use the `simple_launch` package to write the launch files.

3 Writing the node

The node is already more or less setup as `control_node.cpp`. For now it does exactly nothing. The node should focus on one arm only. It is strongly advized to program it for a given arm, then to make it generic.
The sampling time can be set to $dt = 0.1$ s.
The node should:

- subscribe to velocity setpoints as published by the sliders
- publish to the joint command as subscribed by the simulation
- use a service call to the Jacobian to change a SE3 velocity setpoint to a joint velocity command

Note that in our case, the setpoint is given as the end-effector velocity twist, and we want to let the simulation compute the inverse for us. The two fields `inverse` and `ee_frame` of the service request should be set to `true`.

3.1 Helper function to compute the command

The `eigen.h` header defines a helper function that will take care of computing the command:

```
std::vector<double> computeCommand(const JacobianInverseCoefs &Jinv_coeffs,
                                   const geometry_msgs::msg::Twist &v)
```

This function boils down to applying the following computation:

$$\dot{\mathbf{q}}^* = \mathbf{J}^+ \mathbf{v}$$

In practice, the function expects an array of 42 double (that is the `jacobian` field of the response) and a `Twist` that expresses the current setpoint. It will return a `std::vector<double>` of dimension 7, that corresponds to the joint velocity to apply on the robot.

4 The launch file

Once your node runs for a given arm, you should write a launch file that does the following for the left and right arms:

- Run (include) the `slider_launch.py` in a proper namespace
- Run your node to control this arm with the slider