4^{th} Exercise
Robot Operating System Essentials – Summer School
Direct kinematics

IGMR Institute of Mechanism Theory, Machine Dynamics and Robotics | RWTH AACHEN UNIVERSITY

A better visualization

In the last exercise, we managed to visualize the robot and summarized all necessary nodes in one launch file
However, the delta robot was displayed not in the correctly assembled state

This is because rviz as a *visualizer* is <u>not capable of enforcing the loop closure constraints of parallel robots</u>
In the robot urdf, the closed chain delta modeled with missing loop closure constraints so it becomes open chain again
(not a serial topology, but more a tree-like topology with multiple serial arms)
Consequently, the joint_state_publisher_gui parses the open-chain delta robot model and generates too many sliders
And since the robot_state_publisher just applies the joint angles and calculates the direct kinematics, it is still „correct“

If we want to display the robot correctly, we must write our own joint_state_publisher which calculates consistent angles
We can then reuse the robot_state_publisher to calculate the direct kinematics again and the resulting visualization is correct

For this purpose, we will create a new ROS2 package that coveres the necessary maths in the background
This ROS2 package will include a custom JointStatePublisher node which will replace the joint_state_publisher_gui

CMakeLists modification

Similarly to the ROS2 package *delta_robot_description*, we will now create a ROS2 package called **delta_robot_serial**
Follow the steps that you learned from the third exercise to do so
Once done, you can copy over the contents from /materials/delta_robot_serial to your own package
Furthermore, for serial communication, we need the package **serial**, so also copy over that package from materials to your ws

Inside the copied over src folder, there a file called pseudo_arduino.cpp which mimicks the real microcontroller
The goal is now to add a ROS2 executable of this cpp file, such that we can a pseudo_arduino node
For this, we must start by modifying the CMakelists.txt and the package.xml of our new package

We start with find_package(…)
Since we will be writing custom messages and C++ Code in this package, we will need more dependencies
Locate the following line which finds the package with the build tool (ament_cmake) dependency. This must be first/highest:

```
find_package(ament_cmake REQUIRED)
```

and copy/paste it twice to also include the following additional packages/dependencies:
rclcpp (ROS2 Client Library C++)
serial (communication with Arduino)

CMakeLists modification

After enumerating roughly the possible dependencies, we can start telling colcon which executables to generate
We want a ROS2 node from the pseudo_arduino.cpp code, colcon should built an executable based on that file
For this, we add an executable

```
add_executable(pseudo_arduino src/pseudo_arduino.cpp)
```

Here, the 1st argument is the name of the executable (for ros2 run) and the 2nd argument is the relative path to the C++ file

Furthermore, during the build process of the pseudo_arduino node, links are needed to other libraries
This we specifiy with ament_target_dependencies:

```
ament_target_dependencies(pseudo_arduino "rclcpp" "serial")
```

This tells the build system that the executable pseudo_arduino must be linked against the libraries rclcpp and serial

Finally, we need to install this executable from src (code) directory to the colcon shared directory again, which we do using

```
install(TARGETS pseudo_arduino DESTINATION lib/${PROJECT_NAME})
```

Once you are done, there is nothing else to change about the CMakeLists.txt for now
Instead we continue with the package.xml file

package.xml modification

While in the CMakeLists.txt, we specified what steps are needed to build code in the **current ROS2 package**, the package.xml specifies which **other ROS2 packages** are needed first in order to build/execute the current package

For this, we will add more dependencies under

```
<buildtool_depend>ament_cmake</buildtool_depend>
```

Specifically, for the packages **rclcpp and serial** which we need build time and run time dependencies
After the buildtool dependency, add the following:

```
<depend>rclcpp</depend>
<depend>serial</depend>
```

Once you are done, invoke colcon build and then you should be able to use ros2 run on the pseudo_arduino executable!
The output should be something like this

```
2024/08/07 14:46:51 socat[3575145] N PTY is /dev/pts/2
2024/08/07 14:46:51 socat[3575145] N PTY is /dev/pts/3
2024/08/07 14:46:51 socat[3575145] N starting data transfer loop with FDs [5,5]
and [7,7]
[ INFO] [1723034811.477086450]: serial port is opened.
```

Publisher node

As explained in the beginning of this exercise, we will write our own JointStatePublisher
Since it will replace the joint_state_publisher_gui, it should also publish to the /joint_states topic.

The arduino is programmed to send the current motor angles to the PC/to ROS2 via a serial interface.
These will be processed by a **delta_joint_pub** node using direct kinematics in order to calculate also the passive joint angles.

Similarly to the pseudo_arduino node, the delta_joint_pub node must also be specified in the CMakeLists.txt to build
Follow the steps from before to add an executable for the **pseudo_arduino.cpp** file called **delta_joint_pub**

In addition to **rclcpp** and **serial** however, the delta_joint_pub also depends on **sensor_msgs**,
so you should modify the CMakeLists.txt and package.xml accordingly

Furthermore, since the delta_joint_pub.cpp also depends on header files inside /delta_robot_serial/include, we should add

```
include_directories(include ${Boost_INCLUDE_DIRS} include/Eigen)
```

between the find_package calls and the add_executable statements

Afterwards, you should be able to build the delta_joint_pub executable without any problems

Publisher node

The node builds successfully, but this is because unfortunately it is missing most of its code

You will need to add code for the following three sections:

1. Header inclusions
2. The publisher node/class
3. Filling the main body

The first part is relatively easy:
We start with including rclcpp (this is basically always needed)

```
#include <rclcpp/rclcpp.hpp>
```

Next, since we are going to publish joint_states we need to include the message header

```
#include <sensor_msgs/msg/joint_state.hpp>
```

Lastly, because we will have to calculate the direct kinematics we need to include our own header file

```
#include <direct_kinematics.h>
```

Publisher node

For the Publisher Node,

In the constructor, initialize the node with your desired name
Afterwards, declare two parameters necessary for serial communication with the Arduino:
First we declare the default values for the parameters using **declare_parameter**:

```
this->declare_parameter("baudrate", 115200);
this->declare_parameter("serial_port","socatpty1");
```

Next, we override them with the supported values (if they are provided) using **get_parameter**:

```
this->get_parameter("baudrate", baudrate);
this->get_parameter("serial_port", serial_port);
```

The second argument denotes the variable to override into, so you need to define the baudrate and serial_port variables first
After retrieving the parameters, set them using the (given) function **setSerialPort():**

```
setSerialPort(serial_port, baudrate);
```

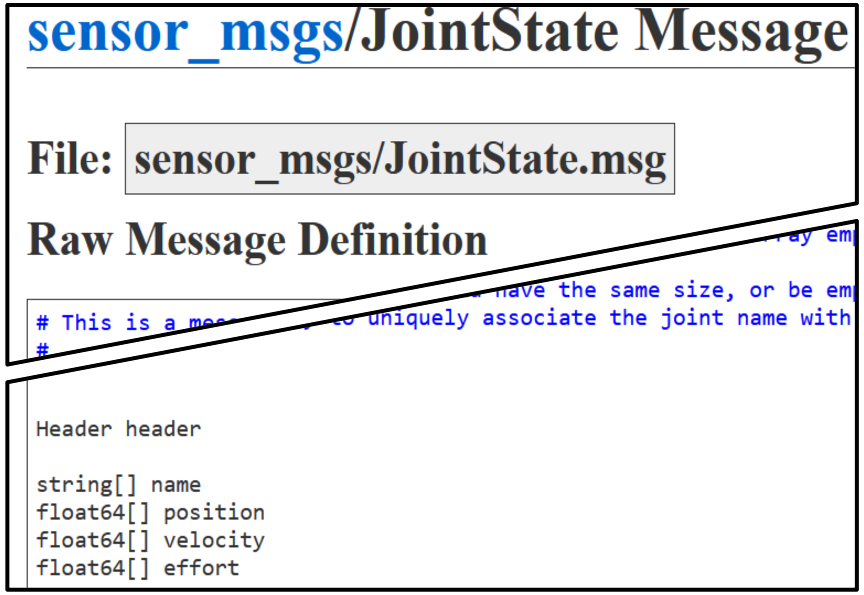
Lastly, create the publisher instance with a queue length of 10 and a timer instance for a publishing frequency of 50 Hz

Publisher node

In the publishing callback, we want the following:

Attempt to read the joint_angles from the serial connection using the (given) function **readDeltaAngles()**
If this is successful, continue with creating a variable to hold the joint_state message to publish

Populate the fields of the message
Refer to the documentation to find out how joint_state is structured



Robot Operating System Essentials
Institute for Mechanism Theory, Machine Dynamics and Robotics
RWTH Aachen University

Publisher node

As a hint for how to get the data for the fields, we get the time (**rclcpp::Time**) using

```
this->get_clock()->now()
```

The link names have already been defined at the top of the delta_joint_pub.cpp file
You can obtain the link position values using the **direct_kinematics()** function from the direction_kinematics.h file

Lastly, do not forget to publish the message you have created

This concludes the Publisher class

Publisher node

In the main loop, we need to do the following

First, initialize rclcpp and create a sharedptr node of the publisher
Then, we check if we can connect to the Arduino and immediately return if it is unsuccessful

```
if(!connectToArduino()) return -1;
```

With the serial connection established, spin the node

We exit the main loop when a shutdown is issued, and do not forget to close the serial port

```
rclcpp::shutdown();
sp.close();
```

Checking the publisher node

After building the node, try running it to see if there are any errors
If there are no errors, echo the contents of /joint_states
You should see something like this:

```
adam-jof4p8dhe0mr2wn@027-fastx:~$ rostopic echo /joint_states
header:
  seq: 391
  stamp:
    secs: 1723106791
    nsecs: 504442751
  frame_id: ''
name:
  - platform_base_x
  - platform_base_y
  - platform_base_z
  - proximal_base1
  - distal_proximal_1_y
  - distal_proximal_1_x
  - proximal_base2
  - distal_proximal_3_y
  - distal_proximal_3_x
  - proximal_base3
  - distal_proximal_5_y
  - distal_proximal_5_x
position: [2.6134905223360007e-18, -1.1091078081476899e-33, -0.07269800547470337, 0.0, 2.244207235944208, 0.0, 0.0, 2.244207235944208, 0.0, 0.0, 2.244207235944208, 0.0]
velocity: []
effort: []
---
header:
  seq: 392
  stamp:
    secs: 1723106791
    nsecs: 524435650
  frame_id: ''
name:
```

This is the current_time we added

These are the joint_names we added

These are the kinematic values of the joints (we only have positions)

This is the start of the next message

Writing another launch file

As soon as you have a bunch of repetitive commands to execute, you should try to automate – for example now
While we are at it, we might as well start Rviz to visualize the joint angles

We will edit the **PseudoArduino.launch** file inside delta_robot_description/launch to fit our needs

In contrast to JointStatePublisher.launch, we need to run the pseudo_arduino node aswell as substitute the joint_state_publisher_gui with our own delta_joint_pub

Furthermore, in the launch file we need to supply the baudrate and the serial_port that the node expects:
For this, we first define the variables (keyword arg) baudrate and serial_port with their respective initial values

```
<arg name="baudrate" default="115200"/>
<arg name="serial_port" default="$(env HOME)/socatpty1"/>
```

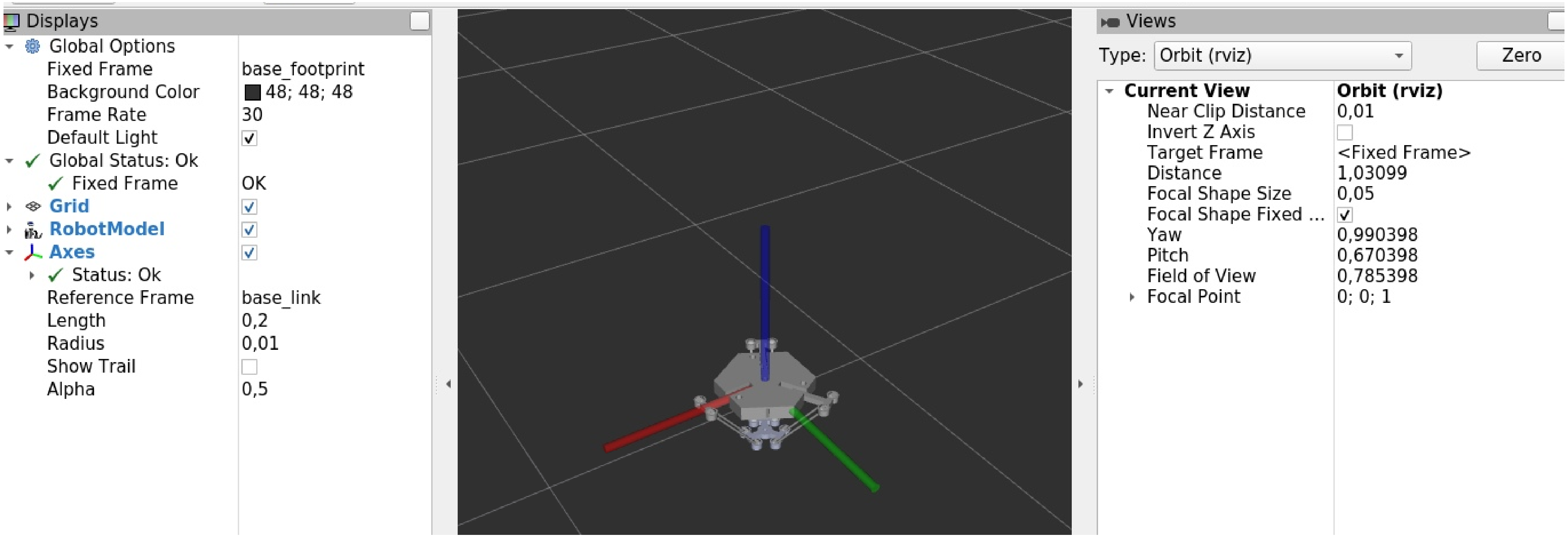
These arguments must be passed to the nodes as its parameters (which is picked up during the get_parameters() call):

```
<param name="baudrate" value="$(var baudrate)"/>
<param name="serial_port" value="$(var serial_port)"/>
```

Writing another launch file

Once your done, launch the PseudoArduino file

You should see an assembled robot in rviz like this:



Forward kinematics

To test if your direct kinematics is working correctly, you have to change the motor angles

This is done by changing the **target_angle** vector in the pseudo_arduino.cpp file, lines 14-16

The target_angle is currently in a symmetric zero position (0,0,0)
Test out some asymmetric values to see if everything is working as expected!
(The angles are in degree, not in radians)

Don‘t forget to build to apply the changes

Thank you
for your kind attention

Contact:

Institute of Mechanism Theory, Machine Dynamics and Robotics
RWTH Aachen University
Eilfschornsteinstraße 18
52072 Aachen

(+49)-241 80-95546
intac-rosdelta@igmr.rwth-aachen.de

www.igmr.rwth-aachen.de