Friedrich-Alexander-Universität
Erlangen-Nürnberg

WS 2025/26

Lab course: Human Robot Interaction

Task 4: Teleoperation

Lehrstuhl für Autonome Systeme und Mechatronik
Friedrich-Alexander-Universität Erlangen-Nürnberg

Prof. Dr.-Ing. habil. Philipp Beckerle
Martin Rohrmüller, M. Sc.

Introduction

Teleoperation was used very early in the field of robotics and literally refers to robotics at a distance. The general idea is to have a human operator for the high-level planning or decision making in robot tasks. Remote control is used to overcome barriers, distances or operate at an environment as shown in Figure 1.

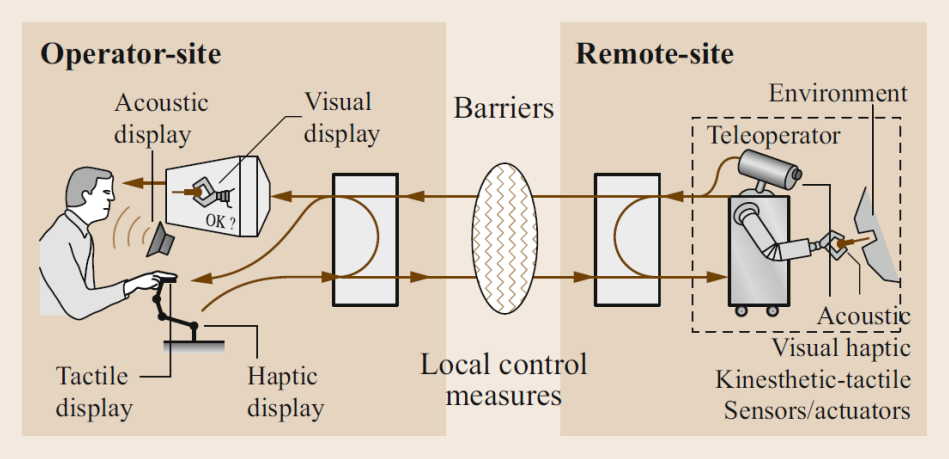


Figure 1: Teleoperation system[1].

In teleoperation, the system can in principle be separated into two sides. On the operator side, there is the connection of the system to the human. This are input devices like joysticks or keyboards on the one hand and on the other hand visual or tactile feedback devices. On the other side of the barrier could be a hazardous environment or a scaling to a very small or large environment. The robot with his control elements and sensors manipulates the environment.

Task overview

In this task, the goal is to set up a framework for the robot to control it with a gamepad as shown in Figure 2. The gamepad is connected to the workstation computer. With the controller inputs the end effector velocities of the robot is directly controlled and the gripper can be opened and closed with pressing buttons. In the jaws of the parallel gripper a force sensitive resistor is installed as sensor for getting the gripper force fed back to the gamepad and as vibration to the user. The sensor is therefore connected to an Arduino microcontroller which is again serially connected to the computer via USB.

[1]Siciliano, Bruno and Khatib, Oussama, eds. Springer handbook of robotics. Berlin, 2016. Chapter 43.

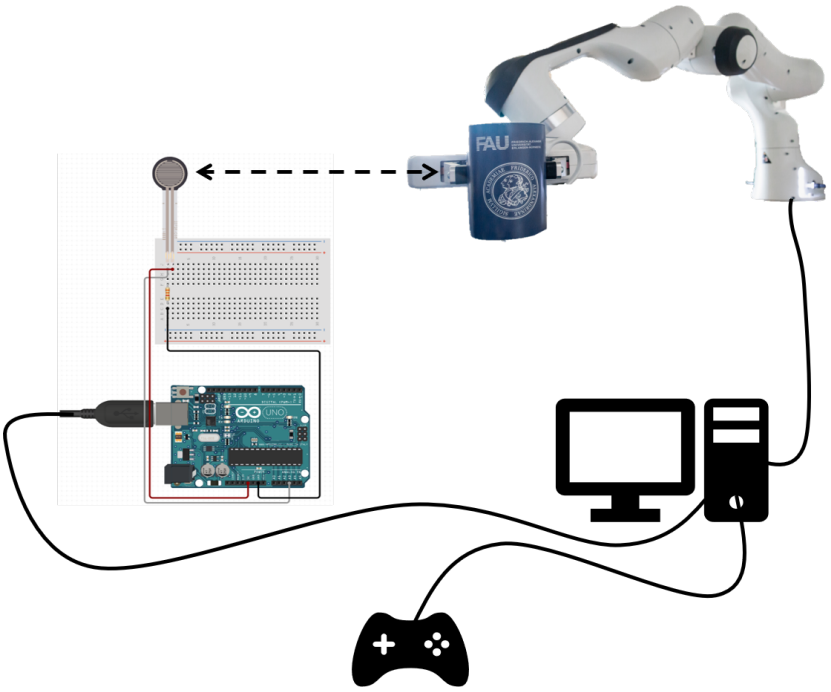


Figure 2: Structure of the physical framework for the task.

In the preparation section, the basics for this task will be explained with simple examples and derived so that the whole project can then be executed in the lab session afterwards. The overall goal of the assignment is to see, how different components can be integrated into ROS to solve a complex teleoperation task with a robot.

Task: Preparation

The preparation part again is supposed to prepare for the experiment that will be carried out in the lab. Therefore, at first the integration of a sensor is introduced. After that, you will learn about two new ROS concepts. This are ROS Parameters and the Launch files. Latter were already applied in other tasks, but now an explanation of those will finally follow.

Sensor integration to ROS

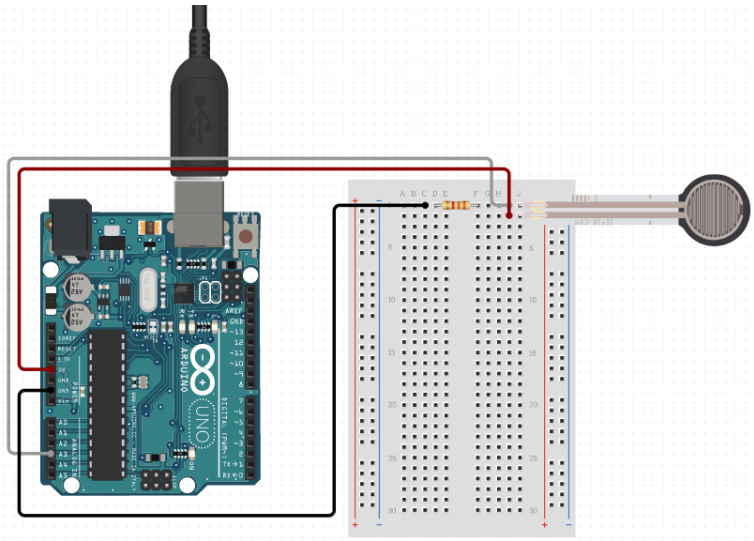


Figure 3: Arduino circuit setup with the FSR.

For the experiment in the lab a force sensor will be used. Therefore it needs to be integrated into the ROS framework to be able to process the measured data. One way to measure forces is with using a force sensing resistor (FSR). This is a sensing element that changes its electrical resistance when subjected to force or pressure. The sensor can be evaluated with a microcontroller. For this purpose, the FSR is supplied with 5V from the microcontroller in a voltage divider and the middle voltage value is read in via an analog pin. Figure 3 shows this setup.

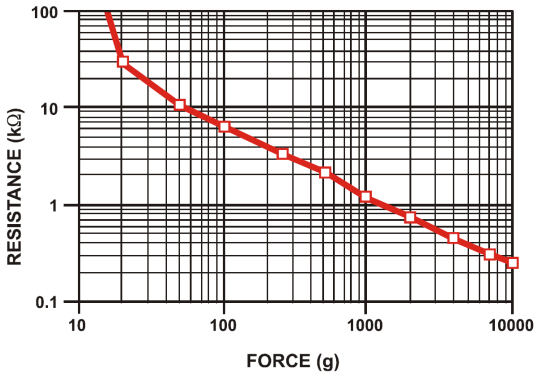


Figure 4: Relation between force and resistance at the FSR [2]

The Arduino microcontroller, which will be used in the lab, samples the analog input with 10 bits and can thus provide an integer value between 0 and 1023. Note here that the FSR has a nonlinear behavior, as shown in the Figure 4 and keep in mind that this will also leave a somewhat nonlinear transfer between the force and the reaction in the experiment. In order to evaluate the measured force within ROS, the microcontroller needs to communicate the integer value. Therefore, the board can be converted to a ROS Node with the *rosserial* Package. With that, a Publisher can send data with ROS messages to the workstation running ROS. The value can then be transferred to the computer with the serial interface and an USB connection. With *rosserial*, a C++ Node can run on the Arduino. Shown below is the code used on the microcontroller:

```
#include <ros.h>
#include <std_msgs/Int16.h>

// Create a Node handle
ros::NodeHandle nh;

// Declare message and Publisher
std_msgs::Int16 force_msg;
ros::Publisher force_pub("pub_force", &force_msg);

int fsr_pin = 0; // the FSR and 10K resistor are connected to pin A0
int fsr_analog_reading = 0; // the analog reading from the FSR resistor
    divider

void setup()
{
   // Initializing ROS Node
   nh.initNode();

   // Creating Publisher object
   nh.advertise(force_pub);
}

void loop()
{
   // Read the sensor value on the analog pin
   fsrReading = analogRead(fsr_pin);

   // Publishing sensor value in a message
   force_msg.data = fsr_analog_reading;
   force_pub.publish( &force_msg );

   // Arduino code is already executed in a loop
   // Processing of ROS with spinOnce inside this loop
   nh.spinOnce();
   delay(300);
}
```

[2] https://cdn-learn.adafruit.com/assets/assets/000/010/126/original/fsrguide.pdf

Code on a microcontroller is always divided into two parts. The setup function is executed once at startup and is responsible for initialization. The loop function then runs in a continuous loop. The Node and the Publisher are therefore initialized in the setup function. In the loop function, the read sensor value is sent repeatedly[3].

Exercise 1.

a) Make yourself familiar with the code shown above. It will be used in the execution part of this task on the Arduino in the lab, but you do not need to do programming in C++.

b) Check out *Adafruit*[4] to get more information about FSRs.

In preparation, you do not yet have access to the sensor, of course. Therefore, take a look at the file *FSR_sensor.py* provided. This generates simulated values for the Topic */pub_force*, as you will get them from the microcontroller on the computer in the lab.

Exercise 2.
Run the Node *FSR_sensor.py* and have a look at the published values. Again, use the *ros_intro* Package for this and the following exercises.

ROS Parameter server

Until now, parameters were always defined directly in the individual scripts. Now, however, it may be that a parameter is used in several scripts or Nodes and a change is to be made globally valid for all. This could also make it possible to create and change parameters that are accessed by the microcontroller. Fortunately, with the start of ROS a parameter server runs, whereby parameters can be written and read at runtime. This can happen in different ways. The simplest way is to set and get parameters via the terminal. Therefore, following commands are given: The command

```
$ rosparam list
```

lists the currently existing parameters. Additional parameters can be set with

```
$ rosparam set parameter_name value
```

or read with

```
$ rosparam get parameter_name
```

Depending on how you write variables that you assign to the variable name, the data type is defined. A string for example is defined with "" and a list with "[..., ...]", where the comma separates the variables, which can even be of different types. According to

[3] https://maker.pro/arduino/tutorial/how-to-use-arduino-with-robot-operating-system-ros
[4] https://learn.adafruit.com/force-sensitive-resistor-fsr

the naming convention of ROS, an examplary name would be */gain* with the slash or */gain/p* with the parameter *p* that belongs to the gain namespace[5].

Exercise 3.
Write, change and read parameters by using the terminal. Do each of this for an integer, a string and a list data type. Remember to start ROS with `roscore` before. Note down the terminal commands you used or take screenshots and add to the submission file. What happens to the parameters and the server when ROS is terminated again?

ROS Parameters with Python

Similarly, parameters can be read and written from Python scripts with the `rospy` library. Therefore a line of code

```
variable_name = rospy.get_param("/parameter_name")
```

reads a parameter from the parameter server, whereas

```
rospy.set_param('parameter_name', value)
```

writes the parameter to the server. Those lines can be added in Python scripts independently of Nodes and at any position. Only if you try to get a non existing parameter, an error would occur. More to this again is in the wiki[6].

ROS Parameters files

Another way to integrate the parameters is by using a YAML file. With this markup language the parameters of all different types can be listed and loaded to the parameter server. By convention those files are saved in a *config* folder of a ROS Package. Especially if there are a lot of parameters, this is an easy way to list all of those and integrate to ROS with just a single command. It is structured as follows:

```
string: 'foo'
integer: 1234
float: 1234.5
boolean: true
list: [1.0, mixed list]
dictionary: {a: b, c: d}
```

and supports all parameter types. More to the YAML files can be found in the ROS wiki too. Similar to the single parameters, there is a command to load the file and thus the parameters into the server:

```
$ rosparam load path/to/file_name.yaml
```

[5] http://wiki.ros.org/rosparam
[6] http://wiki.ros.org/rospy/Overview/Parameter%20Server

Parameter type	Parameter name	Value
string	robot_name	FER
integer	robot_number	1
float	robot_mass	18.0
boolean	load_gripper	true
list	mixed_list	['text', 1.0, 1]
dictionary	gains	{p: 10.0, i: 1.0, d: 5.0}

Table 1: Parameters for YAML file.

In contrast, there is also the possibility to save the existing parameters from the server into a file with the command

```
$ rosparam dump path/to/file_name.yaml
```

which can especially be usefull for debugging. With that commands you must also specify the paths to the files.

Exercise 4.
Work with YAML files and integrate the parameters into ROS:

a) Create a *config* folder in your Package. This is by convention the correct place to store this kind of files.

b) Open the editor and create the file *robot_params.yaml* in the previously created folder.

c) Add the parameters from Table 1 to the file.

d) Load the parameters from the file to the parameter server.

e) Add additional parameters to the parameter server as in Exercise 3.

f) Dump all parameters to a new file *new_robot_params.yaml* and check out its structure.

Add screenshots of the files and terminals to the submission file.

Processing sensor data

Here comes another look back at the sensor: In the laboratory, the entire measuring range of the Arduino from *0-1023* will not be utilized. The maximum force in the gripper will only provide a value of approximately *700* on the Topic */pub_force*. For this purpose, the value will be measured at maximum sensor force at the beginning of the experiment and then set via a ROS parameter. For further processing, this range (*0-700*) must be mapped to a float value of *0-1*.

Exercise 5.
Create a script *mapping_node.py* based on the structure for object oriented Python code for ROS shown in Task 2. The purpose is to map the measured range to a float range of *0-1*. The following elements should be included:

a) A Subscriber to the Topic */pub_force*.

b) Processing the integer value in the callback function or in a separate function to map it to a float value.

c) A Publisher of the processed data to the Topic */force_mapped*.

d) Reading the maximum sensor force from the parameter server and use the integer range from *0-/max_sensor_force*.

You can try the code together with *FSR_sensor.py*. In it, you can change the integer value of *1023* to the maximum value you defined (for example *700*). As a result, the value in */force_mapped* should change between *0* and *1*. Bring the code to the execution part in the lab, as it will be used together with the real sensor. Add screenshots of the terminals to the submission file.

ROS Launch files

Until now, each Node in ROS was always started individually after the ROS master itself was started. The ROS tool *roslaunch* can run multiple Nodes locally with just a single command. This terminal command, typically

```
$ roslaunch package_name file_name.launch
```

will automatically start `roscore` if it is not already running. In addition, all elements defined in the launch file with an XML syntax are started, among which the Nodes are the most important. A descriptive example for a launch file is here below:

```
<launch> <!-- This is a comment -->
<!-- Run Python node -->
<node name="node1_name" pkg="package_name" type="script_name.py" />

<!-- Run C++ node -->
<node name="node2_name" pkg="package_name" type="file_name" />

<!-- Set a parameter -->
<param name="parameter_name" type="variable_type"
    value="variable_value" />

<!-- Load parameters from YAML -->
<rosparam command="load" file="$(find
    package_name)/config/file_name.yaml" />

<!-- Include another launch file -->
<include file="$(find package_name)/launch/launch_file_name.launch"/>
</launch>
```

The correct place for those launch files (file extension .launch) is the *launch* folder inside the related Package. Note that the order in the launch file does not automatically correspond to the order of execution and therefore all Nodes must be independently stable. In addition to the fact that several Nodes can be started with it, roslaunch offers the possibility to set parameters for the parameter server directly with the `param` command, to load several parameters from a YAML file with the `rosparam` command, or also to include further entire launch files. More to the Parameters within the launch file [7] and to the launch file in general [8] can be found on the ROS wiki.

Exercise 6.
Create a launch file with the text editor of Ubuntu according to the information above. It should start the two Nodes *FSR_sensor.py* and *mapping_node.py* and load the parameter */max_sensor_force* into the parameter server. Create a *launch* folder in the Package and save the file there as *sensor.launch*. Source the Workspace and launch the launch file from the terminal. This should now give you the same results as with Exercise 5 when echoing the Topics. Add the launch file (or a screenshot) to the submission file.

As you can see, this is a convenient way to start ROS projects with only a single command and to reduce the number of terminals required.

Exercise 7.
(Voluntary) Integrate the YAML file from Exercise 4 into the launch file. Verify it with getting the parameters from the parameter server after launching it.

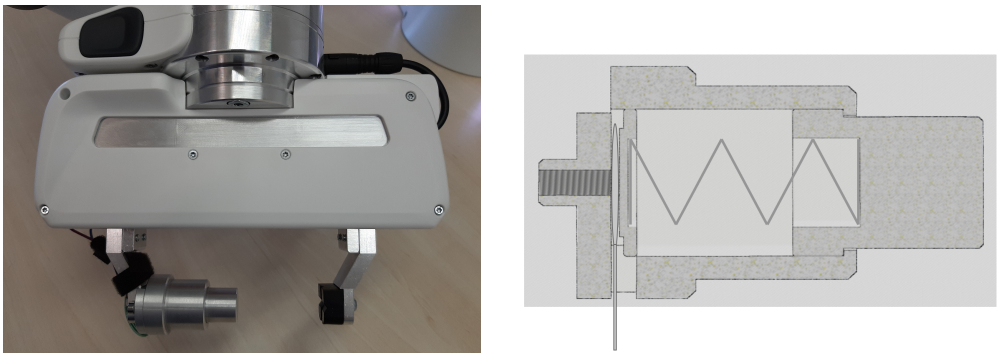
[7] `http://wiki.ros.org/roslaunch/XML/param`
[8] `http://wiki.ros.org/roslaunch`

Task: Execution

In this task, a comprehensive teleoperation application with force feedback is implemented. Components introduced during the preparation phase are utilized, and the provided code is completed to achieve full integration. The entire application is encapsulated in a Launch file, enabling it to be started with a single command. Finally, an experiment is conducted to evaluate the usability of the force feedback mechanism.

The gripper

In this task, the gripper on the robot comes into play. It is a parallel gripper with two fingers which can apply forces up to 70 N. In the laboratory application, a separate attachment is mounted to one of the fingers. Inside, there is an FSR force sensor to measure the grip force. Figure 5 shows the cross-sectional view of this attachment. The piston on the right can slide to the left and is pressing on the sensor over a spring. This setup allows the gripper position to be controlled while applying force.



(a) (b)

Figure 5: Gripper with attachment for a force sensor on the Franka Emika robot in the lab. (a): Parallel gripper, (b): Cross section of the attachment.

Implementing the teleoperation of the robot

In this task, the processing of the inputs and outputs for the teleoperation is done centrally in a Python script named *teleoperation.py*. It is responsible for processing operator input, calculates a cartesian velocity for the low level velocity controller and a gripper position. The latter is processed by the gripper manager, which drives the gripper. Figure 6 shows the structure of the teleoperation application.

Controller input is handled using the Pygame library. This library enables the conversion of analog joystick movements into continuous float variables and maps digital events, such as button presses or holds, to Python functions. These inputs are converted in the script into commands for the cartesian velocity for the robot and for opening and closing the gripper. In two steps the implementation is performed based on given code templates. In the first step, the reading of the controller inputs into the Python script is realized and the velocities as well as the commands for the gripper are generated from

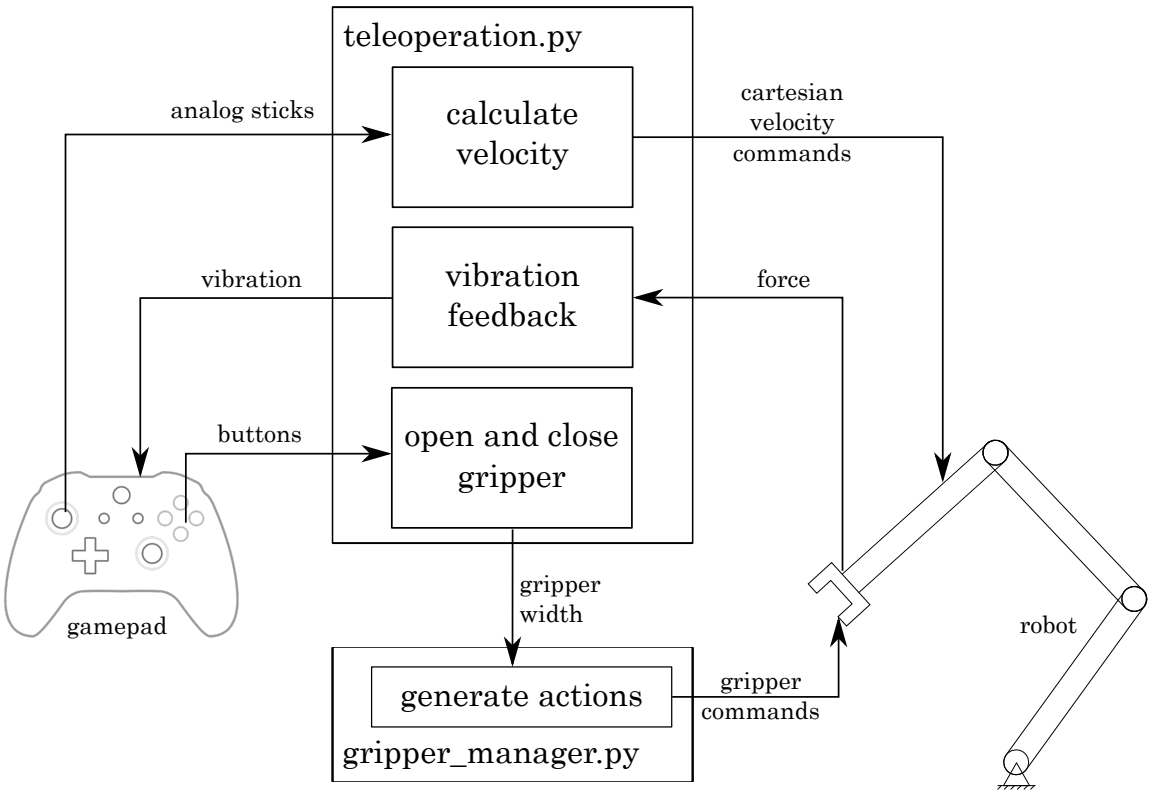


Figure 6: Structure of the whole application.

it. The second step is to wrap the script in ROS and thus add the necessary Publishers and Subscribers.

To perform the first part, the script *teleoperation.py* is provided. In it, the basic functionality of the Pygame library is already integrated. In the `run()` function it is initialized and checks for events in every iteration. This can be key or button presses and releases or motions of analog sticks. With the buttons of the gamepad, boolean variables are set to `True` or `False` and the analog inputs are written into float variables. Also in the loop of the function, those variables are then processed and translated into commands for the gripper and commands for the cartesian velocity for the robot. With Pygame, there are the following event types:

- `JOYAXISMOTION`: Event occurs, when a motion of an analog stick is detected.
- `JOYBUTTONDOWN`: Event occurs once, when a button is pressed and thus only when the state from unpressed changes to pressed.
- `JOYBUTTONUP`: Event occurs once, when a button is released.

Since there are six axis for the velocity (three translational and three angular velocities) but only two analog sticks with two axis each, one button is used to switch between a translational and angular mode. Figure 7 shows the controller assignment.

Exercise 8.
Read the gamepad inputs and use them to generate the presets for the robot and the gripper. To do this, follow the steps below. Every step can be evaluated with executing the script and printing the variables.

Figure 7: Controller assignment.

a) Connect the Gamepad to the computer.

b) Open the script *teleoperation.py* in the Spyder IDE. Since the ROS elements are still missing here, this can be executed directly in Spyder after filling the gaps.

c) Analyze the `run()` function for how the gamepad events are stored to variables.

d) Implement an incrementally increasing or decreasing width of the gripper with $\frac{k_{gripper}}{rate}$. As long as the buttons are pressed, the value `self.gripper_width` should increase or decrease, respectively.

e) Map the values from the analog sticks to the cartesian velocity variable that is in the form of $[v_x, v_y, v_z, \omega_x, \omega_y, \omega_z]^T$.

f) Scale the analog inputs to the cartesian veloctiy with the gain $k_{trans/rot}$ for the translational or rotational velocity.

g) Test with printing the cartesian velocity and the gripper commands. The velocity is supposed to be controlled continously with the sticks, the gripper width with the X and O buttons. Changing to angular velocity and back can be done with pressing the triangle button once, pressing the square button quits the program.

In the script, there is the function `normalize_analog_input()`. It is there, to avoid measuring really small values from the analog sticks when they are in their center position and should give values of zero. In other words, it avoids drift. Figure 8 shows graphically, what the function does.

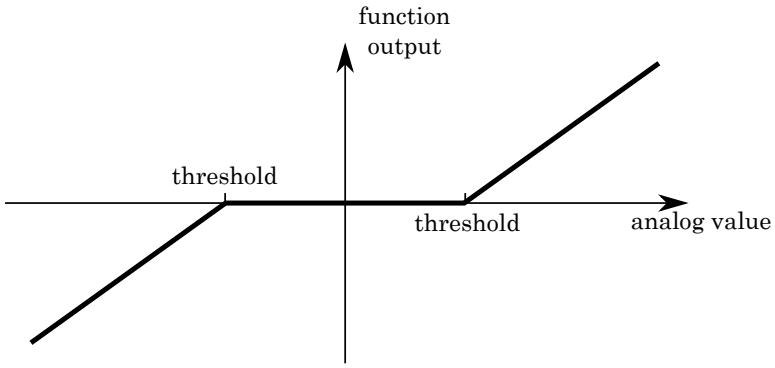


Figure 8: Filter for analog stick.

If changing the variables with the gamepad works, the second step can now be performed. In this one, the elements necessary for the ROS integration have to be integrated.

Exercise 9.
Integrate the script into ROS by following the steps below. For your convenience, you can take the corresponding lines of code directly from the provided script *teleoperation_subscriber_publisher.py*. Put the elements to the correct places.

a) Integrate the Python script into the ROS framework (Reminder: make the file executable and make an entry to the CMakeLists file in the package, then build the workspace). For this exercise and for the rest of today's task again use your group's Workspace and the Package *hri_lab*.

b) Add the line to initialize the script as a ROS-Node.

c) Add the Publisher for the cartesian velocity and the Publisher for the gripper width.

d) Add the Subscriber to the gripper force from the mapping Node.

e) Construct the cartesian velocity command message and the gripper width message.

f) Add the line for the ROS loop and define the rate variable.

g) Assign the computed variables to the messages and publish them.

h) Test the script with the `rosrun` command and check if the velocities and the gripper width are published correctly.

The script just created generates the commands for the robot and for the gripper manager, that will be included later. As the commanded velocity is the same as in Task 2 with the admittance control, The teleoperation of the robot arm can already be tested.

Exercise 10.
Test the teleoperation of the robot with the gamepad. Therefore, launch the velocity controller as this was done in the task before:

```
$ roslaunch franka_control franka_control_cartesian_velocity.launch
```

Additionally run the teleoperation Node. Test controlling the robot with the gamepad. You can remap the analog sticks to the cartesian directions to a more intuitive manner.

The measured force in the gripper and the feedback is still missing here. How this is obtained will be explained in the next section.

Setting up the force sensor on the robot

In this section, the goal is to integrate the force sensor into ROS such that the mapped force value is available as a float from $0 - 1$. Therefore, the Arduino microcontroller is attached to the gripper of the robot. The code for the microcontroller was already introduced in the preparation section of this task and can also be found in the complete version in the ressources folder for the provided course material.

Exercise 11.
Load the Arduino code onto the microcontroller and test the Arduino with the force sensor.

a) Connect the Arduino to the computer. Open the software *Arduino IDE* with

```
$ Desktop/arduino-ide/arduino-ide
```

and open the code *arduino_code_force_node.ino*. Load it to the microcontroller with a click on "Upload". It should automatically detect the connected microcontroller.

b) Start ROS and then use the command (last digit "zero", not O)

```
$ rosrun rosserial_python serial_node.py /dev/ttyACM0
```

to run the Node on the Arduino in the ROS framework on the computer.

c) Read the published force value on the */pub_force* Topic, press the piston on the gripper and check out the gripper forces.

d) Record the maximum integer value that can be achieved when the piston is pressed in level with the housing. This value will be used as parameter as in the preparation.

This force range is then used in *teleoperation.py* as a vibration feedback to the gamepad. Before that, it is supposed to be mapped a number between zero and one with *mapping_node.py*

Launch file for the task

Now the Node can run on the Arduino and one Node can control the teleoperation on the computer. Another Node maps the forces with a recorded parameter. Furthermore, an additional script is provided which manages the gripper commands (*gripper_manager.py*). These components have been tested individually in the last exercises

but must be run together for the whole project to work. To do this, it is a good idea to create a launch file that can be used to run the Nodes with just one command.

Exercise 12.
Create a Launch file for the teleoperation task.

a) Create the *launch* folder in the Package.

b) Create the file *teleoperation.launch* in this folder.

c) Copy the file *gripper_manager.py* to the Package *hri_lab*, make it executable and add it to the *CMakeLists.txt*.

d) Include the file *mapping_node.py* from the preparation into the Package.

e) Write three Node lines for the scripts *teleoperation.py*, *gripper_manager.py* and for the *mapping_node.py* to the launch file.

f) Furthermore, include the lines

```
<node name="serial_node" pkg="rosserial_python"
    type="serial_node.py">
   <param name="port" type="string" value="/dev/ttyACM0"/>
</node>
```

for the Arduino Node. Here, the Arduino-ROS Package is used to integrate the microcontroller via the serial USB port. This does not need to be in the Workspace but is already installed on the computer.

g) Add the ROS parameter */max_sensor_force* for the sensor calibration value from Exercise 11 d).

h) Launch it with the `roslaunch` command.

The concept of ROS parameters could also be used well here to change variables on the Arduino without having to upload new code.

If everything was integrated correctly, the Nodes run and communicate with each other via the Topics */pub_force*, */force_mapped* and */gripper_width* after the start of the launch file. Pressing the piston on the gripper should already result in a vibrating gamepad. A setpoint for the speed is generated from the inputs of the gamepad, which however still goes nowhere, since no controller has been started yet. Finally, this will also be integrated into the launch file.

Finishing the launch file and testing on the robot

As already mentioned, the low level velocity controller is still missing in order to apply the cartesian velocity commands to the robot.

Exercise 13.
Integrate the launch file for the cartesian velocity controller `franka_control_cartesian_velocity.launch` into the teleoperation launch

file using

```
<include file="$(find
    franka_control)/launch/franka_control_cartesian_velocity.launch" />
```

and save the file. Finally, launch the file to start the entire project with this single command. Control the robot with the gamepad.

Carry out the following experiment with the teleoperation framework. Here, especially the force feedback comes into play. One reason to use it, is to give an idea about applied forces back to the operator. With that, you can try not to crush the object to be grasped, that is vulnerable to high forces.

Exercise 14.
At first, start the launch file but without activating the robot. Press the piston a few times while having the gamepad in your hand. Make yourself familiar with the force feedback and how much force it represents. After that teleoperate the activated robot to grasp and lift the provided objects.

Exercise 15.
Display the available Topics. Draw a flowchart containing the Nodes and the Topics and the messages with their types. You can get information about this with the following commands:

```
$ rostopic info /topic_name
$ rostopic type /topic_name
$ rostopic type /topic_name | rosmsg show
```

The flowchart with the Nodes and Topics should reflect the teleoperation pipeline created in this task. You can find more about ROS flowcharts online [9].

[9] http://wiki.ros.org/rqt_graph