Friedrich-Alexander-Universität
Erlangen-Nürnberg

WS 2025/26

Lab course: Human Robot Interaction

Task 1: ROS Introduction and Robot Motion

Chair of Autonomous Systems und Mechatronics
Friedrich-Alexander-Universität Erlangen-Nürnberg

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

Introduction

The descriptions of the first task begin with basic Python instructions. After that an introduction to the core concepts of the Robot Operating System (ROS) follows. In the further tasks these are extended with other important concepts. ROS is a framework for collaborative and industrial robots, for manipulators and mobile robots. It is used in particular for control and sensor data processing. ROS was developed primarily for Linux and is an open-source software and thus offers many freely available packages. With advancing Linux versions, ROS is also constantly evolving and new versions are being developed. After introducing these concepts and the structure of ROS, this assignment teaches the first steps with the real robot.

Task 1: Preparation

The aim of this preparatory part is to teach the basic elements that can then be used to implement a first, simple motion task on the real robot in the execution part. To this end, the Python scripting language is introduced first, followed by the ROS robot framework.

Python-Introduction

For the successful completion of the lab course, you should already have basic programming knowledge. In order to be able to transfer these more easily to Pyhton, this language is explained below with some examples and important components. Python is used in this course mainly because it is one of the two languages supported by the robot operating system (ROS), along with C++. Since it is a high-level, therefore it is easy to learn and use. Furthermore, a large community of developers have provided a large number of freely available libraries and functions. This applies, for example, to functions for transformations and kinematics calculations in robotics. Getting started should be especially easy if you already have Matlab knowledge. A good comparison between Matlab and Python can be found on Sourcefourge [1]. Numpy here refers to the Python library for mathematical calculations. Python is an interpretive language whose focus is also on being easy to read. It fully supports object-oriented programming and also offers a pleasant implementation here. In the following, the language is introduced by means of some code elements.

Code structure

The basic structure of a Python program looks like this:

```
1  import numpy as np
2
3  if __name__ == '__main__':
4     print("Robots are human!")
```

The first lines import packages whose functions and objects can then be used in the code. Very often `numpy` is used here, a library for calculations. With this type of import, it can be addressed in a simplified way using the abbreviation np. At the end of a Python script is the main section. This is called only when the file itself is executed and is not called from another module. It contains the main part of the code.

While-loop

The while-loop in Python for example is structured as follows:

[1] `http://mathesaurus.sourceforge.net/matlab-numpy.html`

```
password = "1234"
word = ""

while word != password:
   print("Wrong password")
   word = input("Enter password: ")

print("Correct password")
```

This runs a loop as long as the checked statement is true. If the statement is false, the loop ends and the code execution continues. The loop can also run infinitely until a break statement ends the loop:

```
while(True):
   #inside the loop
   if ...:
      break
```

This causes the point of execution to jump to the next line after the loop.

Functions

A function in Python is declared with the keyword def:

```
def my_function(x):
   """
   Function to multiply the input by 5
   """
   return 5 * x
```

In the script it is placed above the main part:

```
import ...

def my_function(x):
   return 5 * x

if __name__ == '__main__':
   print(my_function(3))
```

From inside the main part, the functions can be called. Of course, functions inside the script can also be called by other functions.

Important notes

Keep the following things in mind when programming with Python:

- Missing or incorrect indentations lead to errors. The indentations indicate to which section a line of code belongs.

- Specific packages need to be imported in the first lines of the script.
- Indexing starts with 0 (by comparison Matlab starts with 1).
- Python is an interpreting language, there is no compilation necessary before executing the code.
- Data types can be declared, but do not have to be. With a declaration of the variables with direct value assignment these are determined automatically. This can be important to distinguish between integer and float.
- Control structures such as if statements and loops must be terminated with a colon. This also applies to the function definitions.
- There are no semicolons at the end of the lines.
- When copying an array `a` to `b`, you should always use `b = a * 1` because `b = a` does not copy the values, but initializes `b` as a pointer to the memory area of `a`.

Exercise 1.

a) Open the Spyder Integrated Development Environment (IDE).

b) Try out the examples given previously and make yourself familiar with the code. Take care of the Python specific details listed above and note, that there might occur some errors with the symbols when copying code from the PDF, for example with the apostrophes of the `'__main__'` part.

Arrays

The Numpy library, imported with the line

```
import numpy as np
```

enables for example efficient multiplications of matrices and vectors. With the code

```
1 vec = np.array([2, 3])
2 mat = np.array([[1, 2], [3, 4]])
3 np.dot(mat, vec)
```

a matrix *mat* and a vector *vec* are defined and multiplicated. Note that numpy arrays are not built in different dimensions as in Matlab, but are constructed in a nested manner[2].

Figure 1 shows a two-dimensional robot with three joints q_1, q_2 and q_3 with three links. It has a fixed base coordinate system in the first joint and a coordinate system attached to the end effector. This robot will serve as the basis for several exercises.

Exercise 2.

Write a function that calculates the forward kinematics of the robot shown in 1. The function should receive the joint vector $\boldsymbol{q}$ as input and output the cartesian

[2] `https://www.w3schools.com/python/numpy/numpy_creating_arrays.asp`

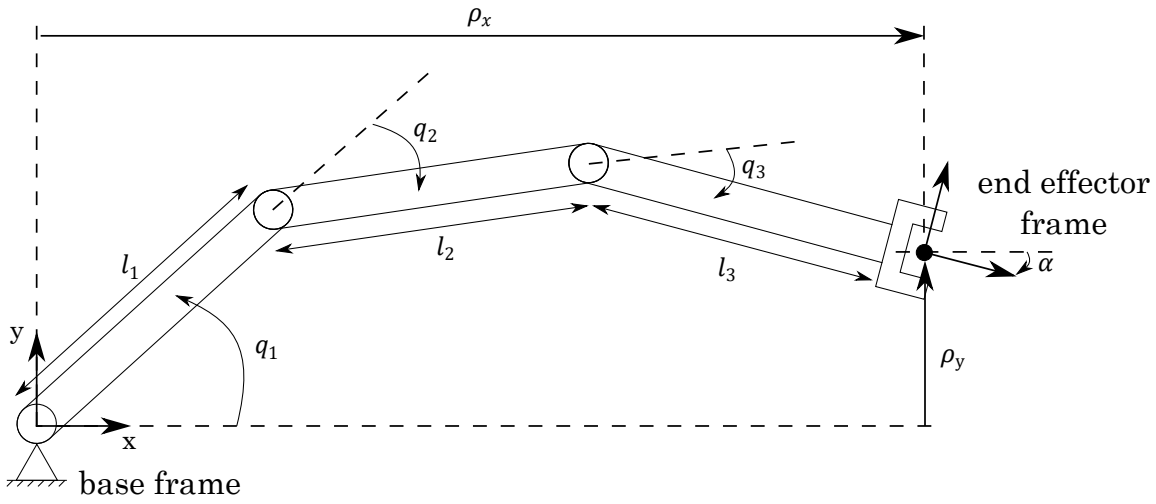


Figure 1: Kinematic structure of the three degree of freedom planar robot.

position of the end effector.

a) Derive the forward kinematics of the robot. It is a two-dimensional robot, thus the resulting cartesion position should be a vector containing the $x-$ and $y-$ positions as well as the angular position of the end effector frame.

b) Define a function that takes the joint angles and the link lengths both as a 3x1 vector as input.

c) Implement the function so that it returns the vector of the cartesian position as numpy-array. Besides vectors and matrices, you can use the numpy functions `np.sin()`, `np.cos()` and `np.pi()`.

d) Try the function in a script by calling it in the main part.

In your submission file, include the kinematic calculations, along with screenshots of both the input and the resulting output from your code solution.

Classes

To use Python in an object-oriented way, classes are available which can be created with the keyword `class` and are structured as follows:

```
1  class Class_1:
2
3      def __init__(self):
4          """
5          Class constructor
6          """
7          self.var = 5
8
9      def function_1(self, x):
10         """
11         Class method
```

```
12          """
13          return(self.var * x)
14
15  if __name__ == '__main__':
16      object = Class_1()
17      print(object.function_1(2.0))
```

Here, the class values respectively the attributes are created in the constructor with the keyword `self` which shows the affiliation. The constructor is the method `__init__(self)` which is called when the class object is created (here in the main part). Class methods are created like normal functions but need the `self` parameter as well to access class variables. More information to Python classes can for example be found at w3schools [3].

Exercise 3.
Create a script with a robot object, that has the joint vector $\boldsymbol{q}$ as a variable and the forward kinematics as a method. Another method will be to compute the jacobian matrix.

a) Create a new script with the name `robot.py` and define a class with the name `Robot`. The class should have an init method that predefines the joint vector to an arbitrary value.

b) Create a method, that prints the current joint values.

c) Create a method, that changes the joint values to passed values.

d) Add the function from the previous task as a method.

e) Add a method that calculates the Jacobian matrix. It can be computed by deriving the cartesian position $\boldsymbol{p}$ by the joint vector:

$$\boldsymbol{J} = \frac{\partial \boldsymbol{p}}{\partial \boldsymbol{q}} \tag{1}$$

f) In the main part of the script, create a Robot object, call the forward kinematics and the Jacobian matrix methods.

In the submission file, include the result of the Jacobian. Create screenshots of both the commands and the outputs of task f).

Exercise 4.
(Voluntary) Complement your robot to output the cartesian pose graphically. The basic way to plot graphs with Python is to use matplotlib [a]. Feel free even to create a graphical output of the entire robot arm, exemplarily shown in Figure 2.

[a] `https://matplotlib.org/stable/tutorials/introductory/usage.html#sphx-glr-tutorials-introductory-usage-py`

[3] `https://www.w3schools.com/python/python_classes.asp`

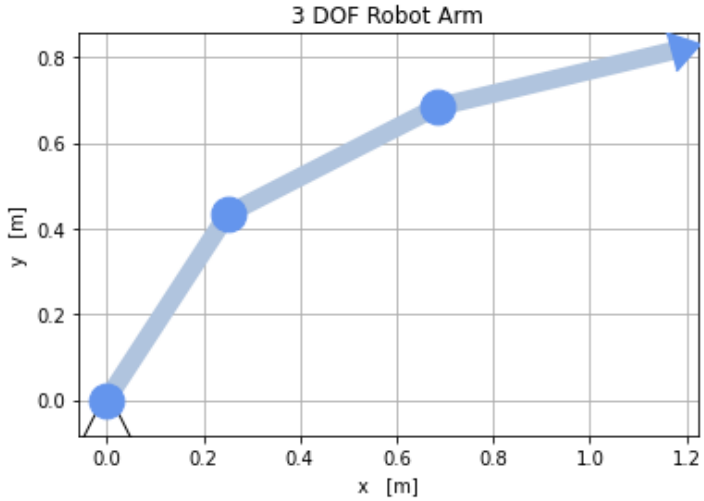


Figure 2: Example output of Exercise 4.

ROS-Introduction

The next section will focus on the Robot Operating System(ROS). This is an open source framework for robot development with a set of libraries and tools to build robot applications[4]. It offers the possibility to integrate hardware and to communicate between different hardware and software instances. Additionally, it allows communication between Python and C++ applications. This already addresses a core idea, which is the exchange of messages between different instances. Following steps align partially with the official tutorials[5]. To begin with, however, the focus lies on the core idea of ROS, which is difficult to grasp with the official tutorials initially. There are also many more tutorials online for the different concepts of ROS that you are welcome to check out. However, to get information about specific topics or about the conventions for implementation, this is the go-to address.

The Workspace

To begin working with ROS it is necessary to set up the Workspace. The Workspace is a folder structure containing different packages. While working on a project, everything is implemented inside the current Workspace. During the lab course, you will complete this on the stick.

Exercise 5.
Start with creating a Workspace folder that can for example be on the Desktop and that contains another folder called *src*. The folder can also be created in the terminal (Ctrl+alt+t) with the command

```
mkdir -p Desktop/robot_ws/src
```

To initialize the Workspace, switch into the Workspace

[4] https://www.ros.org/
[5] http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment

```
cd ~/Desktop/robot_ws/
```

and run the ROS keyword

```
catkin_make
```

that builds the Workspace.

The terminal could be used to execute all input commands to control Ubuntu. For the following, it is used to execute ROS commands and to launch various items. Also take advantage of the fact that multiple terminals can be active at the same time. After successfully executing the build command you can have a look at the Workspace. Two additional folders *devel* and *build* were created. Important for the developer is the *src* folder which should now contain the file *CMakeLists.txt* which later needs to be edited for dependencies. In *src* we will have our packages. A Package can most likely be considered as a project folder. Now it is time to create the first Package.

Exercise 6.

a) Change into the *src* folder of the Workspace

```
cd ~/Desktop/robot_ws/src
```

within the terminal and then type

```
catkin_create_pkg ros_intro std_msgs rospy roscpp
```

and run with ↵ to create the Package.

b) Now build the package, again with

```
cd ~/Desktop/robot_ws
catkin_make
```

which builds not only the Package but also the entire Workspace.

A Package with the name *ros_intro* was created with the dependencies `std_msgs`, which is the standard for sending and receiving information in the ROS framework, `rospy` which relates to the libraries for Python and `roscpp` similar for C++. Figure 3 shows the folder structure of ROS graphically. If this does not match the created structure, the steps should be carefully revisited (The folders "launch" and "scripts" will be created later).

In the progress of the course further Packages will added to the Workspace. You will learn concepts of ROS throughout, but this task will start with the basic concepts of Publishers and Subscribers first.

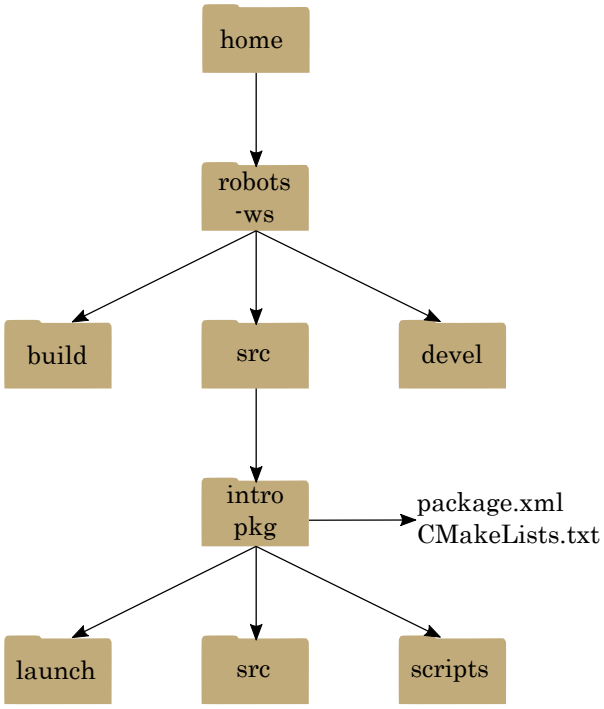


Figure 3: Folder structure of the ROS workspace.

The Publisher and Subscriber

In the ROS framework, the two terms Publisher and Subscriber vividly denote sender and receiver sending continuous messages. This means that data can be exchanged between different Python and C++ scripts, but also with external devices such as microcontrollers that have been integrated into the ROS framework. However, there will be a small introduction to the latter in the *Teleoperation* experiment. So, in other words, Publishers can for example be used to output values of variables in a loop and Subscribers to read those values back in somewhere else. In Python, at first the script needs to run in a loop. The execution of the script should end only when ROS, which is running in the background, is shut down.

```
1   import ...
2
3   def sender(x):
4      ...
5      while not rospy.is_shutdown():
6         hello_str = "hello world %s" % rospy.get_time()
7         ...
8
9   if __name__ == '__main__':
10     sender()
```

In the beginning, the ROS libraries for Phyton and for the message types to be published need to be imported. Those messages can be of normal variable types, but also somewhat more nested structures.

```
#!/usr/bin/env python

import rospy
from std_msgs.msg import String

...
```

The first line of the code makes sure that the script is executed as a Python script, it thus refers to the interpreter. It needs to be the first line of every Python script. In the function called `sender` in this example, the Publisher is initialized as an object. The name of the message to send is chosen to be "hello" here. Then the Node needs to be initialized. This is the step that makes the script a Node that can communicate within the framework. The third line defines the rate the message is later sent with:

```
def sender():
   pub = rospy.Publisher('hello', String, queue_size=10)
   rospy.init_node('sender_node', anonymous=True)
   rate = rospy.Rate(10) # 10hz
   while ...

...
```

To complete the Publisher, the message can be sent repeatedly with the defined rate in the while loop of the function:

```
...
   while not rospy.is_shutdown():
      hello_str = "hello robot %s" % rospy.get_time()
      pub.publish(hello_str)
      rate.sleep()
...
```

Here, the `publish()` method is sending the variable (which is a string in this case). In the official tutorials you can find a complete example of such a minimal script with a Publisher and additional explanations of the code lines [6].

Next, for using a Subscriber, a callback function is necessary in addition to a function that contains the lines for the Subscriber:

```
1  import ...
2
3  def callback(msg):
4     # do something with the message data
5
6  def receiver(x):
```

[6] http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29

```
7         ...
8         rospy.spin()
9
10    if __name__ == '__main__':
11        receiver()
```

Here, the receiver function is called and prevented from exiting with the spin command. This is not like the while-loop, which is running the code inside in a loop, it is just preventing the run-out of the code, so that the Subscriber can stay active. To receive the message, those lines can be applied:

```
    ...

    def receiver(x):
        rospy.init_node('receiver_node', anonymous=True)
        rospy.Subscriber("hello", String, callback)
        rospy.spin()

    ...
```

As before, the Node is first initialized with a name. Then it is declared that the Subscriber should receive the message with the name "hello", with official ROS words this would also be called to subscribe to the Topic "hello". As soon as a message is received, it is processed in the callback function:

```
...

def callback(msg):
    print(msg.data)

...
```

Here, the string is contained in the data argument of the message. This is the usual case with the standard message types. It is simply printed, but it could also be written into a variable of the script.

Exercise 7.
Create a script that publishes a `float` variable instead of a `string` in the described example.

a) Create a new folder for the Python scripts in the previously created package next to the *src* folder. Give it the name *scripts*. We use this name by general convention for Pyhton scripts, whereas the *src* folder usually contains C++ scripts.

b) Create a new Python file in this folder and give it the name `send_float.py`

c) Add following lines to the file *CMakeLists.txt* in the package or uncomment them, respectively.

```
catkin_install_python(PROGRAMS scripts/send_float.py
 DESTINATION \${CATKIN_PACKAGE_BIN_DESTINATION}
)
```

This makes sure, that the python script will be installed and interpreted correctly.

d) Make the script executable with the terminal command

```
chmod +x /path/to/scripts/send_float.py
```

so that it can later run inside ROS. Adapt the path to the Workspace location.

e) Write the script according to the given example but use the `Float64` message type from `std_mgsg` instead and name the message `float_robot` and publish any values you like.

f) Go to the Workspace and run the `catkin_make` command. The new build needs to be done for newly created scripts, but after that, Python scripts can be edited without having to run the build command afterwards again.

If you try to run the code in an IDE like Spyder, it will not work. This is because for ROS you first need to start a ROS environment. To use the functions in Python you need to run the script with a ROS specific command which will be part of the next exercise. Note, that similar to the IDE, if the code is incorrect, the errors are displayed in the terminal. This can help to debug the code.

Exercise 8.

a) Open up a Terminal window (if there is no one open yet) and change into the workspace with

```
cd ~/Desktop/robot_ws
```

b) Type in and execute the command

```
source devel/setup.bash
```

to source the workspace. This is very important to do whenever you open up a new terminal in order to be able to find all the files within the Workspace.

c) Now, use the command

```
roscore
```

to start ROS.

d) Open up a new Terminal, change again to the Workspace and source it. Now we can eventually run the Python script as a ROS Node. Therefore execute

```
rosrun ros_intro send_float.py
```

Note, that the command is always structured as
`rosrun <package_name> <node_name>` and that it will fail, if there is any programming error in the Python script. This is also one way to check for errors in the workflow while making changes in the code. You can shut down a Node with Ctrl + c .

e) If the run worked, the Node is now constantly publishing the message. All the published messages can be listed with

```
rostopic list
```

which should give `/float_robot` as part of the output.

f) Finally, to have a look at the message, use

```
rostopic echo /float_robot
```

The terminal should now display the message repeatedly at the specified rate.

Create a screenshot showing all active terminals used for this task while the message is published and add it to the submission file.

This message is now sent in the ROS framework. If one wants to receive the message, which is called subscribing, one needs to implement another node with a Subscriber, what will also be the next step.

Exercise 9.
Create a script that subscribes to the `float_robot` message. If you wish to have a look at a minimal script, check out the official ressources again [a].

a) Create a new Python file in the *scripts* folder and give it the name "receive_float.py".

b) Write the script according to the given elements you need for a Subscriber and again use the `Float64` data type. Print the received message data directly in the callback function.

c) Add the script to *CMakeLists.txt* and make it executable.

d) Build the Workspace and follow Exercise 8 a-d again, but now also run the receiver_node.

Create a screenshot showing all active terminals used for this task while the message is subscribed to and add it to the submission file.

[a] `http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29`

If the two nodes are running, then one should now be sending and the other receiving and outputting the value in the terminal. Graphically, this could be represented with

Figure 4.

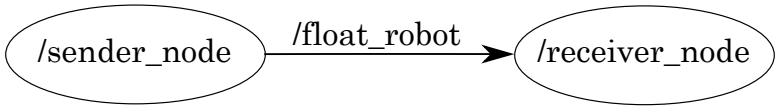


Figure 4: Publishing and subscribing with a message.

Exercise 10.
(Voluntary)

a) Run only the Subscriber node. Then, try to publish the message directly from the terminal. There is a similar command to the one in Exercise 8 f.

b) Try to write the received message data in the subscriber script into a global variable with the callback function instead of printing it directly.

Task 1: Execution

During the task preparation, you have already learned about the basic idea of ROS. For completeness, the official terminology is briefly introduced here. The individual scripts are referred to as nodes. With the `roscore` command the ROS Master is started, with the `rosrun` command a Node is registerd to the framework. As shown in Figure 5 the Nodes can be not only on the computer but also on different platforms. This could be a microcontroller for example, as you will discover in the upcoming *Teleoperation* experiment.

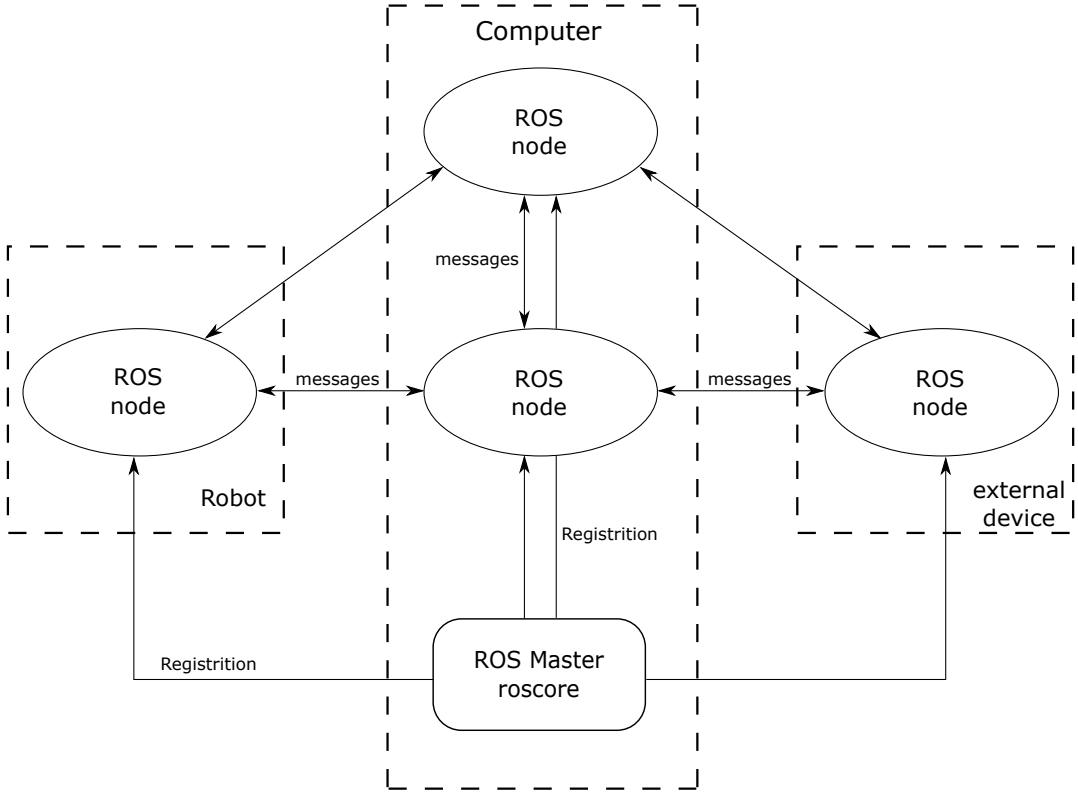


Figure 5: Communication between ROS nodes.

The Nodes communicate with exchanging information. The method with Publishers and Subscribers is one example to do this, you will discover another method with the *Cooperative lifting* task. In the official ROS terminology, one would call the exchange of messages with Publishers and Subscribers publishing and subscribing to Topics. The term Topic describes the exchange buses. As you could already see in the preparation, different names are defined in the Python code. With the `init_node()` function, names for the Nodes are assigned. With the `Publisher()` and `Subscriber()` functions the Topic names are passed. When using ROS, the names of the Nodes are less important, but the names of Topics are, because you need them to access the respective messages.

In the execution part, you will operate the real robot with ROS and perform a simple motion and kinematics task. First, however, the start up and handling will be described.

Starting up the robot

This section describes how to start the Franka-Emika robot. These steps will not be repeated in future tasks. If you do not remember, simply fall back on these descriptions.

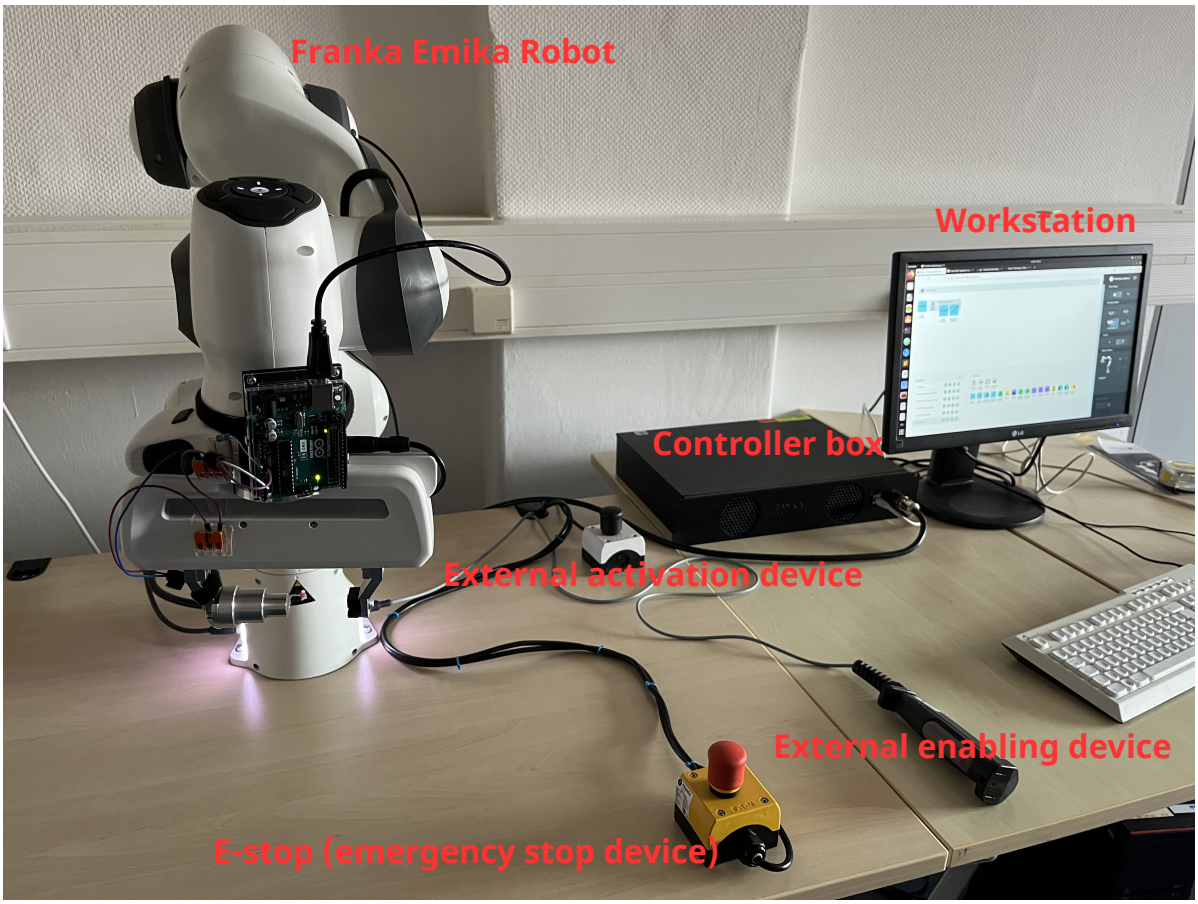


Figure 6: Franka Emika robot and control components.

Figure 6 shows the setup of the robot as it is in the laboratory. The robot is connected to the controller box. This contains the hardware necessary to generate the motor inputs for the robot. In order to receive commanded control inputs (for example position commands) it is connected to the Workstation computer. This computer is the platform for the implemenations during this lab course with an installation of ROS and all the robot related software on it. There are three switches connected to the system. The external activation device and the external enabling device both do the same thing. One of them needs to be activated, to be able to move the robot. If not activated, the robot remains in its current position and won't accept motion commands. Additionally, there is the E-stop, which is an emergency stop device. Pressing it cuts the power supply to the entire system. This releases the breaks of the robot, which block all joints in their motion.

⚠The E-stop must be kept in a well reachable position while working with the robot. This applies for the remainder of the whole lab course!

Exercise 11.
Switch on the robot!

a) Press the power switch on the back side of the controller box under the table.

b) Open the Firefox browser on the Workstation computer and type `192.168.1.101` into the address field.

This is the ip address of the robot and gives access to the graphical panel as shown in Figure 7. The connection might take some time if the boot process of the robot hasn't finished yet.

c) Unlock the robot by clicking the padlock symbol in the browser window.

During this process, the robots makes loud clicking noises while the physical breaks in the joints are deactivated. In this mode, the robot can already be moved to different positions easily by compensation for gravity.

d) Grab the gripper on the end effector of the robot and press the two opposite buttons at the same time. Carefully move the robot's end effector.

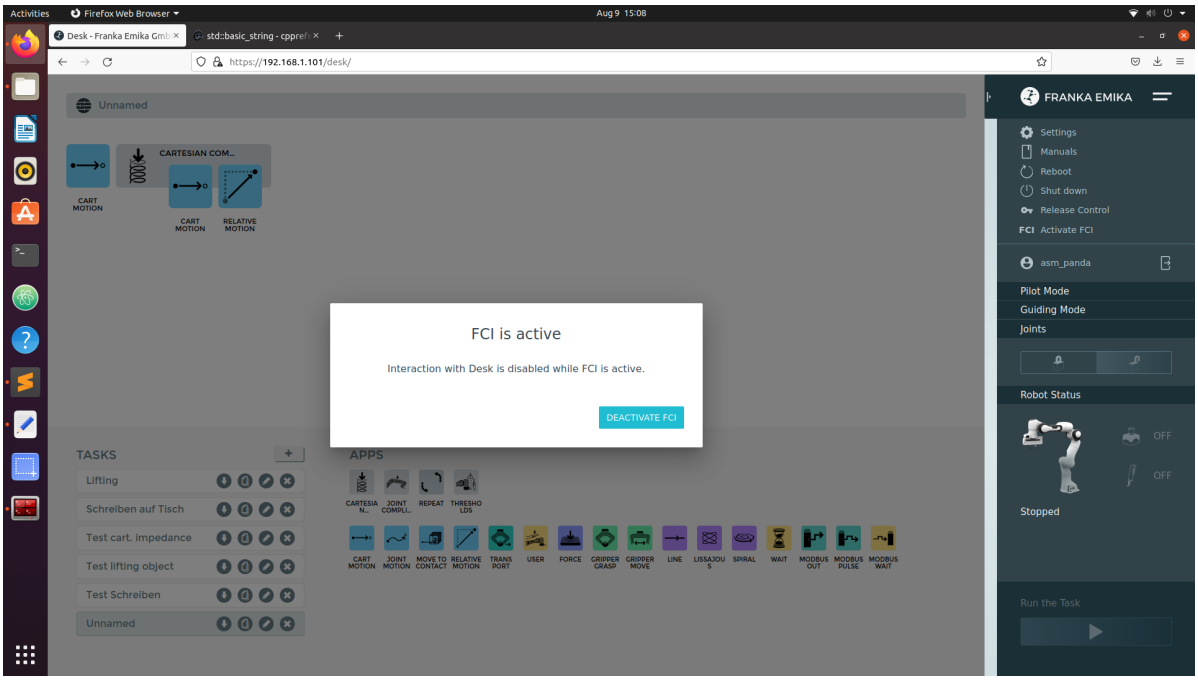


Figure 7: Franka Emika Desk interface.

The graphical panel Desk gives access to robot settings, shows the robot status and provides applications for simple graphical programming of robot tasks.

Starting with ROS

The next steps will focus on ROS. Therefore, a Workspace is needed. In this Workspace, different Packages will be integrated. Some Packages will be given due to the installation

of Franka ROS, which is provided by the robot manufacturer.

Exercise 12.
Creating and starting the ROS-environment.

a) Create a new Workspace in the folder *Desktop/Workspaces* and name it "catkin_ws_group<number of the group>". The word "catkin" is not a keyword, but in ROS mostly an identifier, for everything related to workspaces.

b) Create the *src* folder and run

```
catkin_make
```

within the Workspace folder.

c) Integrate the Franka Packages: Copy the folder *franka_ros* from *Desktop/resources* to the *src* folder of the group's Workspace.

d) Additionally, create a new Package with

```
catkin_create_pkg hri_lab std_msgs rospy roscpp
```

add a *scripts* folder and finalize it again with the `catkin_make` command. This might take a few minutes.

In the preparation part of this task it was shown how to source the Workspace. This means, how to make the terminal and ROS starting from terminal able to access all the elements from the different Packages inside one Workspace. To avoid having to run the source command in every new terminal, do the following:

Exercise 13.
Add a source line to the *.bashrc* file:

a) Open up a terminal with (Ctrl+alt+t). With that, the current path is the root location of Ubuntu and not a specific folder. Execute

```
$ gedit .bashrc
```

to open the file in an editor.

b) Scroll down to the bottom of the file and add the line

```
source /home/<computer_name>/Desktop/Workspaces/
catkin_ws_<groupname>/devel/setup.bash
```

adapt the paths and save the file. Close the editor and the terminal.

When the command completion in the terminal with pressing double tab works correctly for the files in the Workspace, the editing of *.bashrc* was succesful. Please make note, that only one Workspace should be included in the *.bashrc* file at a time, otherwise it is not ensured, that files are accessed from the Workspace one is working in. To do this, all

other lines for sourcing other Workspaces should be commented. For the further course you can also do this on the USB stick.

Robot motion

The next steps are to read the robot states and to get the measured joint positions. With that, kinematic calculations can be carried out and trajectories created.

Exercise 14.
Read the current robot positions. Make sure that the enabling switches are not activated and that there is a white light on the robot's base.

a) Activate the FCI in the browser window. This always needs to be done when there is any interaction with the robot besides the desk applications.

b) Open a new terminal and run the command

```
$ roslaunch franka_control franka_control_joint_position.launch
```

in order to power up the robot and the ROS framework. This is done with a launch file. What this exactly is and does will be explained in another task of this lab course. Please notice, that this already includes the `roscore` command. In this case, the low level controllers for receiving position inputs are started as well as several Nodes that publish joint and robot states and thus let us access a lot of robot related values. The connection to the robot is done via the ip address. An error similar to "move command rejected" should appear, which means that no activation device is currently pressed.

c) Check out the existing Topics.

d) Access the values of the joint positions via the terminal. Therefore use the Topic */joint_states/position*. This should show nine values in the terminal, whereby the first seven represent the joint angles, the last two the positions of the gripper fingers.

e) Move the robot with pressing the button on the robot gripper and see, how the position values change. Figure out, how the values are mapped to the robot joints.

f) Move the robot roughly to the position

$$\boldsymbol{q} = [0, -\frac{\pi}{4}, 0, -\frac{3\pi}{4}, 0, \frac{\pi}{2}, \frac{\pi}{4}]^T \approx [0, -0.79, 0, -2.36, 0, 1.57, 0.79]^T$$

in order know where the robot will move to. This will be the goal position for the trajectory, as we need a starting and a goal position to plan a trajectory.

g) Move the the gripper to another position that is roughly 10 cm away from the goal position. Note the joint positions after releasing the gripper button when the robot is at rest.

In the next step, the robot finally is going to move. This motion will be between the positions from last exercise. Therefore, the way between those two positions will be executed step wise and thus a trajectory between the positions will be created. This means, that the distance between the positions needs to be split up in a number of way points.

Exercise 15.
Create a joint trajectory between the positions from the last exercise. This trajectory will be defined and executed in a Node, created in the script *joint_trajectory_steps.py*. Copy it from the *resources* folder on the Desktop into the *hri_lab* Packages *scripts* folder. Add it to *CMakeLists.txt* and make it executable. Fill the gaps in the script according to the TODO comments. To be able to access all the necessary published values, run the command from Exercise 14 b) in another terminal first, then run the script as a Node.

⚠For the next exercise use the external enabling device to let the robot be moved!

Exercise 16.
Joint trajectory motion.

a) Execute the trajectory created in *joint_trajectory_steps.py* on the robot. Now, the enabling switch needs to be pressed before launching the launch file and running the Node and needs to be hold while the program is running.

If your implementation was successful, the robot should go to the goal position set in the script and given in Exercise 14 f).

b) Record two more feasible goal positions and run the motion task again.

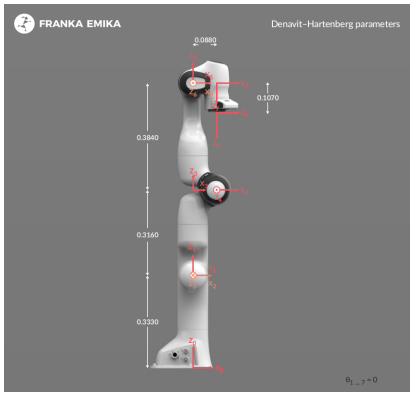
c) (Voluntary) Create and execute a trajectory to not just one goal position but over multiple waypoints.

In the next section the focus will be on the kinematics of the robot. With this new insights, another motion task will be executed.

Robot kinematics

The Denavit-Hartenberg (DH) convention provides a systematic description of the relationships between two joints. The joints of a robot are considered one after the other and a distinction is made between a translation and a rotation. For this purpose, coordinate systems are defined in the individual joints according to certain rules. Figure 8 (a) shows the coordinate frames defined for the Franka-Emika robot[7]. The distances between the frames along the rotation axis of the joints are then given by a set of parameters a and d that specify the geometrical values of the real robot. The result is, that a set of three parameters for each robot joint is enough to define the kinematic chain when sticking

[7] `https://frankarobotics.github.io/docs/control_parameters.html#denavithartenberg-parameters`



(a)

Joint	a (m)	d (m)	α (rad)	θ (rad)
Joint 1	0	0.333	0	θ_1
Joint 2	0	0	$-\frac{\pi}{2}$	θ_2
Joint 3	0	0.316	$\frac{\pi}{2}$	θ_3
Joint 4	0.0825	0	$\frac{\pi}{2}$	θ_4
Joint 5	-0.0825	0.384	$-\frac{\pi}{2}$	θ_5
Joint 6	0	0	$\frac{\pi}{2}$	θ_6
Joint 7	0.088	0	$\frac{\pi}{2}$	θ_7
Flange	0	0.107	0	0

(b)

Figure 8: Coordinate frames (a) and Denavit-Hartenberg parameters (b) of the Franka-Emika robot.

to the convention. For the Franka-Emika robot, this is the table in Figure 8 (b). The individual parameters have the following meaning:

- d_i: distance between x_{i-1} and x_i. Translation in direction of z_{i-1}.
- θ_i: angle of rotation between the link i and $i-1$ around z_{i-1}. Here, θ_i is variable and corresponds with the joint variable q_i.
- a_i: length of the link i. Translation in direction of x_i.
- α_i: angle between z_{i-1} and z_i. Rotation around x_i.

The transformation matrices

$$\boldsymbol{A}_i^{i-1} = \boldsymbol{S}_i^{i-1} \cdot \boldsymbol{Q}_i^{i-1} = \begin{bmatrix} 1 & 0 & 0 & a_i \\ 0 & cos\,\alpha_i & -sin\,\alpha_i & 0 \\ 0 & sin\,\alpha_i & cos\,\alpha_i & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} cos\,\theta_i & -sin\,\theta_i & 0 & 0 \\ sin\,\theta_i & cos\,\theta_i & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2}$$

can be used to calculate the representations of the individual coordinate systems from the parameters, where $\boldsymbol{S}_i^{i-1}$ contains the translation a_i and rotation α_i and $\boldsymbol{Q}_i^{i-1}$ contains the translation d_i and rotation θ_i. This follows the so called modified DH convention. With

$$\boldsymbol{A}_i^{i-1} = \begin{bmatrix} \boldsymbol{R_i^{i-1}} & \boldsymbol{p_i^{i-1}} \\ \boldsymbol{0} & 1 \end{bmatrix} \tag{3}$$

the matrices are composed of the relative rotations and positions of the coordinate systems. With the multiplication of the transformations, the end effector position $\boldsymbol{p_n^0} = [p_{nx}^0, p_{ny}^0, p_{nz}^0]^T = \boldsymbol{p_{ee}^b}$ and orientation with respect to the base coordinate system can be calculated with

$$\boldsymbol{A}_n^0 = \begin{bmatrix} \boldsymbol{R}_n^0 & \boldsymbol{p}_n^0 \\ \boldsymbol{0} & 1 \end{bmatrix} = \boldsymbol{A}_1^0 \cdot \boldsymbol{A}_2^1 \cdot ... \cdot \boldsymbol{A}_n^{n-1}. \tag{4}$$

of the n frames. Since with the Denavit-Hartenberg convention, certain rules are followed with the placement of the coordinate systems, the transformations can be represented

with the given formulations in simple way and the description with the robot can be supplied by few parameters. For more information on the parameters and the underlying rules, for example refer to the book "Robot Mechanisms" [8] which also is the source of the formulas in this section.

Exercise 17.
Calculate the forward kinematics of the Franka-Emika robot.

a) Copy the code template *forward_kinematics_dh_franka_emika_robot.py* to your Package and fill in the gaps at the "TODOs" to calculate the transformations. Since this file doesn't contain any ROS libraries (rospy), it can be executed directly in the Spyder IDE.

b) Again run franka control as in Exercise 14 b) and subscribe to the joint variables.

c) Move the robot to at least three different configurations and use the measured joint positions to calculate the transformations with the script. Verify your implementation with a comparison to the end effector position of the real robot. Therefore, have a look at Figure 8 to see how the base frame of the robot was placed. Note down the pairs of values in Table 1.

Table 1

Joint angles	**x, y, z pos. end effector**
[, , , , , ,]	[, ,]
[, , , , , ,]	[, ,]
[, , , , , ,]	[, ,]

[8] J. Lenarcic, Robot Mechanisms. Springer 2013

You have now learned the forward kinematics of the Franka-Emika robot and executed a first movement, planned in the joint space. In particular, you have become familiar with the procedure using ROS. This will be built upon in the next task.

Exercise 18.
Think about the following questions. You should answer them for the follow-up assignment.

a) When planning the trajectory as you did with Exercise 15, what do you have to consider, so that the execution is possible?

b) Describe the behavior of the end effector in space when traversing a trajectory that was planned linearly in joint space.

c) Control was applied at the position level in this case. What other level of robot commands would be independent of the starting position?

d) How can you proceed, if you want to reach a certain point with the end effector in the workspace, respectively if it should move linearly?

e) $\boldsymbol{R}_n^0$ of Equation 4 is the rotation matrix of the end effector to the base frame. Research and describe a method for calculating the orientation of the end effector from the rotation matrix and describing it using three angles..