Friedrich-Alexander-Universität
Erlangen-Nürnberg

WS 2025/26

Lab course: Human Robot Interaction

Task 3: Cooperative Lifting

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

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

Introduction

The goal of this task is to implement an application with the robot to lift and transport an object in collaboration. The algorithm for this comes from a scientific paper[1], which you will get to know. In the preparatory part there is an introduction to an additional ROS communication concept, which will then also be incorporated into the application in the execution part.

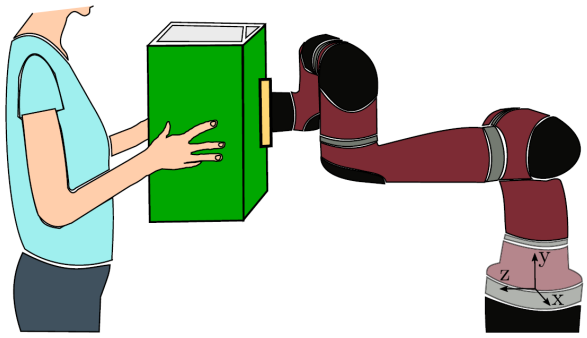

Figure 1: Cooperative lifting of an object with a robot[1].

[1]S. Marullo, Cooperative Human-Robot Grasping With Extended Contact Patches. *IEEE Robotics and Automation Letters,* 2020

Task: Preparation

To prepare for the task, at first there is a short description of robot forces and with that some theory about what you could already practically experience. Then another concept for communication with ROS services is introduced. In addition to this, you are expected to read parts of the scientific paper on a robot application in preparation, which will be implemented in the execution part. If you want to start with reading, then you can go to Section **Cooperative Lifting method** first.

Robot contact forces

In certain interaction tasks, such as object manipulation, it is essential to account for the forces exerted by the robot. Additionally, proper attention must be given to coordinate transformations to ensure that end-effector forces are calculated accurately - whether in the coordinate system of the end effector itself or in the base coordinate system. In the second task, you have already worked with cartesian forces on the robot and with torques in joint monitoring. An interesting thing about robots in general is the relationship between the forces acting on the end effector and the torques in the joints.

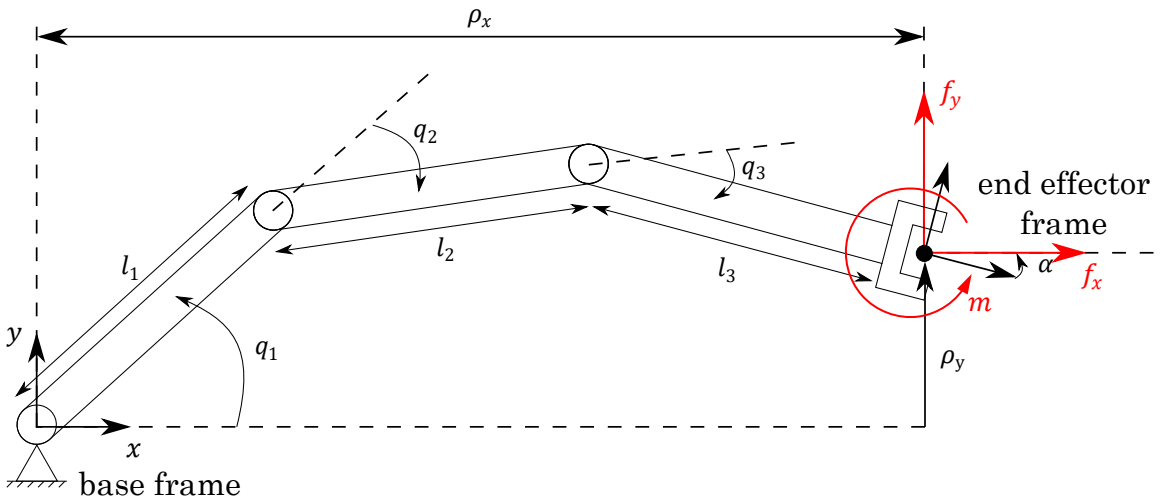

Figure 2: Structure of the three degree of freedom planar robot.

Remember the example of the two-dimensional robot with three degrees of freedom from the first assignment of the course. In this section, the relationships between the forces acting on the robot and the joint moments will be considered. However, this will be restricted to the static case here and therefore no robot dynamics will be considered.

Exercise 1.

a) Calculate the vector of joint torques $\boldsymbol{\tau}$ from the vector of end effector forces. Therefore, consider a three-dimensional vector

$$\boldsymbol{F} = \begin{bmatrix} f_x \\ f_y \\ m \end{bmatrix} \tag{1}$$

that contains the force component in x- and y-direction as well as the torque m in the plane acting on the end effector but expressed in the base frame. Set up the equations for the three joint torques τ_i by calculating the system of force and torque equilibrium from the geometry in Figure 2.

b) Now calculate the joint moments with the formula

$$\boldsymbol{\tau} = \boldsymbol{J}(\boldsymbol{q})^T \boldsymbol{F} \tag{2}$$

by using the Jacobian from the first task. Compare the results with those from a).

c) Check your equations for plausibility by substituting combinations for the following values for the joint positions and for the forces:

$$\boldsymbol{q}_I = \begin{bmatrix} \frac{\pi}{2} \\ 0 \\ 0 \end{bmatrix}, \quad \boldsymbol{q}_{II} = \begin{bmatrix} \frac{\pi}{2} \\ -\frac{\pi}{2} \\ 0 \end{bmatrix}, \quad \boldsymbol{F}_I = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad \boldsymbol{F}_{II} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad \boldsymbol{F}_{III} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{3}$$

In doing so, think about the lever arms and the directions of the forces. For simplicity you can choose the link length to be 1.

Add the solutions to the submission file.

Depending on the application, it may also be important to infer the forces at the end effector from given joint moments. These can generally be calculated with the equation

$$\boldsymbol{F} = \left(\boldsymbol{J}(\boldsymbol{q})^T\right)^{-1} \boldsymbol{\tau}. \tag{4}$$

This is what, for example the Franka Emika robot does with his torque sensing and publishing of derived end effector forces.

Exercise 2.

Consider the Franka Emika robot with its seven degrees of freedom. A six-dimensional force vector acts at the end effector. Which is easier, to determine the force vector from the joint moments, or the other way around? Why?

ROS Services

Previously, to communicate between the individual Nodes in ROS, messages were used to be sent and received. This is done with the Publishers and Subscribers in a continuous manner. So with the given rate values are sent and read continuously. When transferring messages with Publishers and Subscribers, there is both the possibility to send or receive a message via the terminal. This is possible because the Publishers or Subscribers work independently and constantly send or receive.

Exercise 3.
Before you get to know another way of communication in the following, think about the disadvantage of the previous way of message exchange. Which other possibilities for communication would make sense in this context?

In contrast, with ROS Services there is the ability to interact in a request-response paradigm. This means that one-time information can be transmitted at a single point in time. This is then processed by the other instance and a response message is returned. As a general rule, this is used for processes that can run very quickly or that require a confirmation of completion. With ROS Services, only the instance that receives a request can be accessed via the terminal. Thus, the Service part of the communication, that processes the request and gives the response must be integrated into a Node [2].

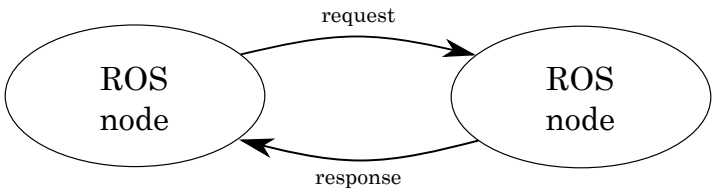

Figure 3: Basic structure of ROS services.

Figure 3 shows the basic structure of the ROS service. In this case, a Node responds with a unique message, composed of different variables with the common data types, to a request that looks similar but may have different data types. The special thing about the Service is that these variables and data types must first be defined in a separate, so called Service file, looking like this:

```
1    int64 a
2    int64 b
3    ---
4    int64 sum
```

The three dashes "- - -" separate the request message from the response message. In this example, there is a request of two integer values to a Node, from which a response message consisting of a single integer value is expected. Next, the following elements are required for a Node that will process the Service request:

[2] http://wiki.ros.org/Services

```
import ...

def handle_add_two_ints(req):
   ...

def add_two_ints_server():
   ...
   rospy.spin()

if __name__ == '__main__':
   add_two_ints_server()
```

These are mainly a function that starts the Service and keeps it running (`rospy.spin()`) and a function that handles the processing, similar to a callback function. The single elements will be explained next in detail. First the rospy library must be imported and with the next line the library files for the Service messages:

```
#!/usr/bin/env python
import rospy
from <package_name>.srv import AddTwoInts, AddTwoIntsResponse

...
```

These are generated from the *srv* file when the Workspace is built. This is one file with the name of the service file and one with the same name but with "Response" appended.

Second, the script must be initialized as a ROS Node. If this already happened in the script, for example if there was already a Publisher or Subscriber implemented, this step is skipped. The service is declared with the predefined message types ("AddTwoInts") and the name of the handle function ("handle_add_two_ints").

```
...

def add_two_ints_server():
   rospy.init_node('service_node')
   s = rospy.Service('add_two_ints_service', AddTwoInts,
       handle_add_two_ints)
   rospy.spin()

...
```

And lastly, an incoming request must be processed in the handle function. This function is called with the request instances of ("AddTwoInts") and returns the response instances.

```
...

def handle_add_two_ints_server(req):
   return AddTwoIntsResponse(req.a + req.b)

...
```

In this case, the two values are added and the sum is output directly in the `return()` function. These elements are therefore necessary to receive a request, process it and respond with a generally different message. If you want to see an overview of these elements in a complete script, you can refer to the official tutorial [3] on which these explanations are based. With the following exercises you can learn about the implementation of ROS Services.

Exercise 4.
Create the Service file needed for declaring the request and response messages. By ROS convention, these go into their own folder in the current Package.

a) Create a folder *srv* in the *ros_intro* Package with

```
$ cd ~/Desktop/robot_ws
$ mkdir -p src/ros_intro/srv
```

or by just creating a new folder in the file manager.

b) Create a file with the name *DivideInts.srv* in the folder you just created and define the file according to the instructions above, but so that the response message now consists of a `float64` variable and a `string` variable. You can do this with either Spyder or the basic Editor.

[3] `http://wiki.ros.org/ROS/Tutorials/WritingServiceClient(python)`

c) Next, the file *CMakeLists.txt* from the package needs to be edited. Therefore, add `message_generation` into

```
find_package(catkin REQUIRED COMPONENTS
 roscpp
 rospy
 std_msgs
 message_generation
)
```

This includes the module that generates the service messages. Additionally, the specific file created before needs to be added:

```
## Generate messages in the 'srv' folder
add_service_files(
 FILES
 DivideInts.srv
)
```

One last step is to uncomment the following lines:

```
## Generate added messages and services with any dependencies
    listed here
generate_messages(
 DEPENDENCIES
 std_msgs
)
```

With this, the source files will be built in the next step:

d) Compile the workspace!

You can check the correct implementation with the terminal command

```
$ rossrv show DivideInts
```

which should show the lines defined in the *srv* file. If you like to read more information, check out the tutorial online [4].

Exercise 5.
Implement a ROS Node that requests, processes and responses the messages of the Service with the file from Exercise 4.

a) Create a file *divide_ints_node.py* in the packages *scripts* folder.

[4] `http://wiki.ros.org/ROS/Tutorials/CreatingMsgAndSrv`

b) Write the node according to the instructions above but so, that it gets two integer values with the request message. In the handle function, those should be devided and the result be output as a float value. If there is a division by zero, the additional string variable should be "Division by zero!" and otherwise "Division ok.".

After sourcing the Workspace, starting ROS and running *divide_ints_node.py*, the Node is ready to receive requests. From the terminal, this can be done with:

```
$ rosservice call service_name args
```

With pressing ⇤⇥ twice while typing in the terminal, you can always see possible command completions. This can be very helpful to see, how for example arguments must be typed.

Exercise 6.
Call the Service via the terminal and try it with different integer values to be divided. Add screenshots of the terminals and both division cases to the submission file.

Often it is sufficient to request the Service from the terminal and thus trigger an event or receive information. Sometimes, however, the request must come from a script. This part of the Service is then called the Client. There are only a few elements needed

```
1  import rospy
2  from <package_name>.srv import *
3
4  def add_two_ints_client(x, y):
5      """
6      Function with the necessary lines to call the Service
7      """
8      rospy.wait_for_service('add_two_ints_service')
9      add_two_ints = rospy.ServiceProxy('add_two_ints_service',
           AddTwoInts)
10     resp = add_two_ints(x, y)
11     return resp.sum
12
13 if __name__ == '__main__':
14     # Integer values
15     x = ...
16     y = ...
17     print(add_two_ints_client(x, y))
```

which basically is at first a method that blocks, until the Service is available. After that, a handle for calling the Service is created with the `ServiceProxy()` method. This can then be used just like a normal function in the next line, that returns the response as it was structured in the *srv* file. Again, it is important to import the srv library files

here as well. This file also needs to be made executable and an entry to *CMakeLists.txt* needs to be created as always.

Exercise 7.
Create a file *divide_ints_client.py* that calls the Service and does what you did from Terminal in Exercise 6. Add screenshots of the terminals to the submission file.

It is interesting to note here, that no Node needs to be initialized in the Client. Thus, the lines for calling Services can be put into the code independently of ROS Nodes.

Additional ROS Service commands

Here are some interesting commands for the usage in the terminal: Similar to how the Topics can be listed in the terminal, there is a

```
$ rosservice list
```

command that can be used to list the individual running Services. Therefore, ROS must be started with `roscore` first. Specific information about the Service, the name of the Node that provides the Service and the type of the message can be obtained with

```
$ rosservice info service_name
```

and even more commands are listed online[5]. Those commands are accessing information in a currently running ROS environment. On the contrary, with `rossrv` information about structures of the messages and *srv* files defined in the Package can be obtained:

```
$ rossrv info package_name/service_name
```

This will show results about the Package structure, regardless of whether ROS is currently running or not. If you like to further train your handling of ROS Services, you are welcome to work on the next exercise. In the execution part there will be another application of this concept on the real robot.

Exercise 8.
(Voluntary) Add methods to the robot class in the file *robot.py* from the previous tasks in order to add a ROS Service. In the request, values for the end effector forces are to be passed and the individual joint moments are to be returned as a response. Therefore, you can calculate the torques with Equation 2. The request with a force vector could happen from an additional Client or just from the terminal. A definition of a dedicated *srv* file could be

```
float64[] force
---
float64[] torques
```

[5] `http://wiki.ros.org/rosservice`

with vector data types suitable to numpy arrays.

Cooperative Lifting method

As already mentioned, the objective of the execution in the lab is an experiment to lift an object together with the robot. This is based on an algorithm, that was developed for this application and described as "Single Point Method" in the Paper *Cooperative Human-Robot Grasping With Extended Contact Patches*.

Exercise 9.
Read at least the sections II.A, II.B and III.A of the provided Paper. Make yourself familiar with "Algorithm 1".

Task: Execution

In this task, the algorithm you learned about in the preparation is implemented on the robot and an experiment is performed with it. The cooperative lifting is to be started and stopped with the help of ROS Services. Those Services are to be implemented in the first part of the execution. In the second part, the actual algorithm is implemented. In the end, the entire project will be in a single Python script.

Exercise 10.
In order to work on the task, set up the computer:

a) Add your group's Workspace to the *.bashrc* file and comment potential others.

b) Open the browser and connect to the Franka Desk. Navigate to *Settings* > *End-effector* and set the end effector to "None". Apply the setting and go back to "Desk" and activate FCI.

This gripper setting results in proper compensation for gravity when the robot is moved by hand. If not edited, the compensation would still consider the weight of the gripper. For the exercises, again use the Package *hri_lab* in your group's Workspace.

Implementing the ROS Services

A quick reminder of the terms: A ROS Service is the concept for communication with a request-response paradigm as shown in Figure 4. The main element of a Service needs to be implemented in a Node and is there to get a request message, to process it and return a response message. This part is called Server. Then there is the second element called Client, that makes the request. This can also be replaced by a Service call from the terminal. A Client can be in a ROS Node, but can also stand alone in a script.

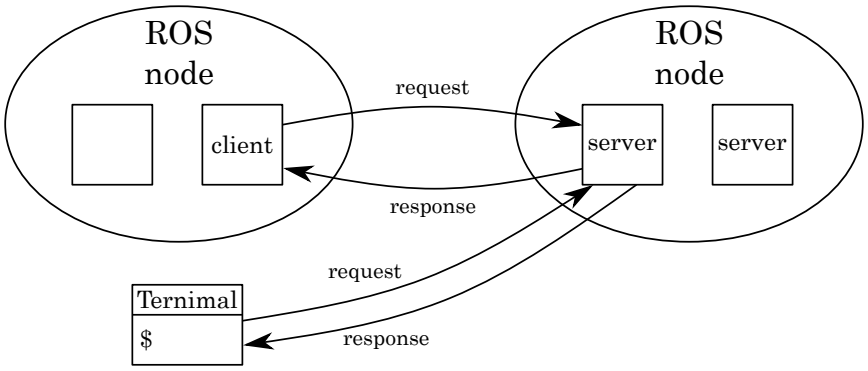

Figure 4: Basic structure of ROS Services.

In the first part of this lab task, two Services are to be implemented. The first is to start the robot movement and the second is to stop it. Only a single Node is to be used for this, which means that the two Server parts must be in it together. For testing, both Services should also be called with only a single script. You can use the provided file

cooperative_lifting_servers.py for this first part of today's task. This contains the basic framework for this Node.

Service to start the Robot

For the start service, the associated *srv* file for defining the request and response message is already given. This definition is the first step necessary to implement the Service as you did already in the preparation. Additionally, a part of the associated Server is given in *cooperative_lifting_servers.py*.

Exercise 11.
The srv file for this Service (*StartCooperativeLifting.srv*) is already available in the ressources folder for the task. Create a folder *srv* in the Package and copy the file into it. Adapt the following things in the *CMakeLists.txt* of your Package:

- Add message_generation to `find_package()`
- Add the *srv* file to `add_service_files()`.
- Uncomment the part `generate_messages`.

Finish it with building the Workspace.

From this provided Service message definition you can abstract all the necessary information that is needed for the structure of the ROS Service. This will be important for the next exercise. Next, the Server part of the start Service will be tackled. As shown in the preparation, this is the part which processes the request and returns a response.

Exercise 12.
Edit *cooperative_lifting_servers.py* at the TODO comments to carry out the following steps. A few elements need to be completed in the script.

a) Finish the declaration of the Server.
b) Look at the *srv* file to see, how the response needs to be structured.
c) Edit the handle function. As defined in the *srv* file, in the response should be a string that contains information similar to "Cooperative lifting starts". Also enclose the force to the string output. Return the response message.
d) Test the Service from the terminal by calling it.

Service to stop the Robot

With the start Service the variable `self.start` is set to `True`. With the stop Service this is to be set accordingly again to `False`. The following function is given here ready for this:

```
def handle_stop_cooperative_lifting(self, req):
    """
    Set the minimal normal force to zero and set the start
        variable to False
    """
    self.f_n_min = 0.0
    self.start = False
    stop_string = "Cooperative Lifting was stopped! The force
        limit was set back to " + str(self.force_n)
    return StopCooperativeLiftingResponse(stop_string)
```

Figure 5: Service handle for stopping.

However, in contrast to the exercise before, the message file is now to be derived from this function.

Exercise 13.
Create the *srv* file for the stop Service. Therefore, have a look at the *handle_stop_cooperative_lifting* function from Figure 5. What are the request and the response messages? Save the file in the correct folder and integrate it to ROS as well.

The next step is to integrate this stop Server into the same script for the Servers as well.

Exercise 14.
Integrate the stop Server into *cooperative_lifting_servers.py*.

a) Insert the function from Figure 5 in the right place in the code.

b) Declare the Server with the name `stop_cooperative_lifting_server`.

c) Import the messages from the new *srv* file.

d) Start the Server in the Node.

e) Again, run tests from the terminal.

With ROS Services and its cross-platform capabilities, it is for example possible to trigger a start command from a microcontroller with an attached button.

Cooperative lifting with a single contact

In the following, the first algorithm from the paper[6] will be implemented. From this you should get insights into the behavior of the robot and the difficulties associated with this application. This variant of a cooperative lifting algorithm assumes a single contact from the robot side to the object to be lifted. However, on the robot side there is a rectangular contact area. The human must grasp the object on the other side while

[6] S. Marullo, Cooperative Human-Robot Grasping With Extended Contact Patches. *IEEE Robotics and Automation Letters,* 2020

moving it. Figure 6 shows the setup in the lab with the robot and the attachment with the contact area.

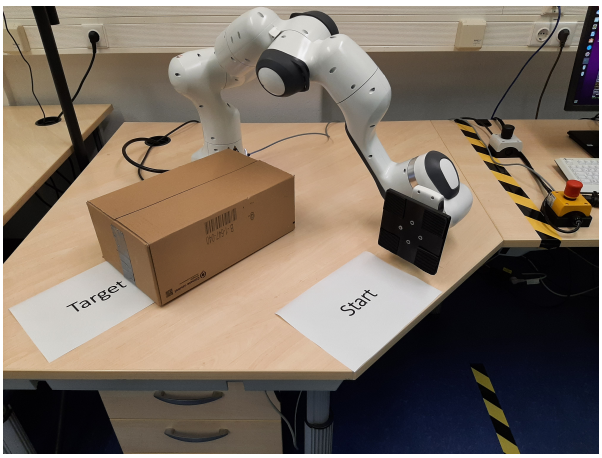

Figure 6: Cooperative lifting setup in the lab.

Exercise 15.
Before the implementation, think about and discuss how the robot will behave when lifting with only one contact point considered. What performance do you expect? What problems might arise?

One point to take care of for the implementation is the following: The end effector forces obtained from the ROS Topic `/franka_state_controller/F_ext` are expressed in the end effector coordinate system (as shown in Figure 2). In contrary, the velocities that are to be computed for the low level velocity controller are expressed in the base frame. Since the algorithm used also calculates the quantities in the end effector frame, it is adopted in this way. In the end, the calculated goal quantity is transformed into the base frame. With that, a velocity command can be generated. A low-level velocity controller then realizes the cartesian end effector velocities on the robot. This type of robot controller will be used for generating robot motions in this task. In the code template *cooperative_lifting.py*, the rough algorithm structure and some ROS elements are already given. Some of those you know from the previous tasks. However, the Service implementations are missing, which you can simply copy and paste after successfully completing the previous exercises. Later, the application is supposed to be started with a Service call from the terminal, so you do not need the Client part.

Exercise 16.
Follow the TODO comments in *cooperative_lifting.py* and thus the following steps to write the entire application in one Python script.

a) Copy the Server elements for the start and the stop Service into the script. With the handle of the start service, additionally set the variable for the maximum limit of the normal force as `self.f_n_max = self.f_n_min + 5.0`.

b) The algorithm part is still missing, but the ROS parts should be integrated now. The script should already be executable. To access the Topics, run the velocity controller with

```
$ roslaunch franka_control
    franka_control_cartesian_velocity.launch
    load_gripper:=False
```

and leave the robot deactivated. Verify that it is error-free by running it in ROS and checking, if the correct cartesian velocity array is published.

c) Implement the single patch (SP) algorithm according to the pseudo algorithm of the paper. The Numpy functions `np.dot`, `np.linalg.norm`, `np.cross`, `np.hstack` and `np.matmul` can be very helpful here.

Likewise, the robot must be brought into the initial situation with which the lifting task can be performed. Therefore, a launch file is available.

Exercise 17.

a) Bring the robot to the start position with holding the enabling switch and executing

```
$ roslaunch franka_example_controllers move_to_side.launch
```

in the terminal.

b) Launch the velocity controller as in Exercise 16 b) and then finally run the previously created Node.

c) If the implementation is successful, you can now start the algorithm with calling the start Service from the terminal. Hold the object in place while you do this. Set the minmal force to `10`. The robot should make contact with it. Slowly move the object upwards and observe how the robot follows the movement.

Experiments with the single contact algorithm

To finally test the implementation, a few lifting experiments should be performed with the provided box, similar to the paper. Note the following aspects during the execution:

- Start with a slow movement so that the robot can react.
- The focus should be on not loosing contact to the robot.
- Work together: One person should activate the enabling switch, monitor the robot and start the application, while another person should perform the task on the robot.

Exercise 18.
Carry out the experiment three times for each participant (nine times in total). Note down in Table 1 for each try, whether it was successful (contact can be maintained, time for fulfillment less than one minute) or not. Follow these steps for each run:

a) Bring the robot to the start.

b) Start the cooperative lifting as you did in the exercise before.

c) Move the box together with the robot to the target position.

Table 1

Experiment	#1	#2	#3	#4	#5	#6	#7	#8	#9
Success (yes/no)									

As you could see, it is quite complex to implement such an algorithm with an additional ROS integration and with considering different robot parameters. Therefore, in the context of this task, the extended method from the paper can no more be implemented.

Exercise 19.
Think about the following questions. You should answer them for the follow-up assignment.

a) Read the section *B. Pick-and-Place Experiments* from chapter IV. EXPERIMENTS from the paper. Which of the findings can you share, where do you have other experiences?

b) List the low-level control modalities/levels used in this task and the previous tasks and name other possibilities.

c) Research about ROS Actions. What is the difference between ROS Services and ROS Actions? What are the advantages and disadvantages respectively?