SARAS CALL h2020-ICT-2016-2017

INFORMATION AND COMMUNICATION TECHNOLOGIES

SARAS

"Smart Autonomous Robotic Assistant Surgeon"

D7.2 – Software/Hardware architecture for the MULTIROBOTS-SURGERY platform

Due date of deliverable: December 28, 2018

Actual submission date: December 28, 2018

Grant agreement number: 779813
Start date of project: 01/01/2018

Lead contractor: Università di Verona
Duration: 36 months

Project funded by the European Commission within the EU Framework Programme for Research and Innovation HORIZON 2020	
Dissemination Level	
PU = Public, fully open, e.g. web	✓
CO = Confidential, restricted under conditions set out in Model Grant Agreement	
CI = Classified, information as referred to in Commission Decision 2001/844/EC	
Int = Internal Working Document	

D7.2 – Software/Hardware architecture for the MULTIROBOTS-SURGERY platform

Editors

Francesco Setti (UNIVR)
Riccardo Muradore (UNIVR)

Contributors

Nicola Piccinelli (UNIVR)
Giacomo De Rossi (UNIVR)
Fabio Falezza (UNIVR)
Robin Strak (MEDIN)
Sabine Hertle (MEDIN)
Stephan Nowatschin (MEDIN)
Alessio Sozzi (UNIFE)
Marcello Bonfé (UNIFE)
Marco Minelli (UNIMORE)
Federica Ferraguti (UNIMORE)
Cristian Secchi (UNIMORE)

Reviewers

All partners

The work described in this document has been conducted within the project SARAS, started in January 2018. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No.779813.

TABLE OF CONTENTS

List of Figures

List of Tables

1 Introduction

The aim of this document is to provide a detailed description of the SARAS architecture for the MULTIROBOTS-SURGERY platform with particular attention to the integration of multi-master multi-slave (MMMS) bilateral teleoperation architecture described in D3.1 – Multi-modal human-robot interfaces and architecture for the MULTIROBOTS-SURGERY platform into the robotic system developed in the project.

In the MULTIROBOTS-SURGERY platform, the main surgeon controls the da Vinci tools from the da Vinci console, whereas the assistant surgeon teleoperates standard laparoscopic tools mounted at the end effectors of the SARAS robotic arms from a remote control station equipped with virtual reality and haptic devices. The assistant surgeon will perform the same actions as in standard robotic surgery, but this time by teleoperating the tools instead of moving them manually.

The MULTIROBOTS-SURGERY platform is an example of multi-master/multi-slave (MMMS) bilateral teleoperation system, where two users cooperate on a shared environment by means of a telerobotics setup. Virtual reality is exploited to improve the visual feedback provided to the surgeons. Pre-operative and intra-operative data allow us to produce virtual fixtures (*e.g.* virtual walls impassable for the tools or optimal paths the surgeon is guided to follow during delicate phases of the procedure) to help the surgeon safely navigating the human body or to enhance his/her 3D perception. An Oculus Rift device provides the necessary visual feedback to the assistant surgeon, while the main surgeon's feedback is provided by the da Vinci console. The MMMS architecture will be validated in terms of transparency, usability, cognitive load, and command/action misunderstanding. Results will be reported in deliverable D1.2 – Experimental tests and validation activities on the MULTIROBOTS-SURGERY platform.

The MULTIROBOTS-SURGERY platform is a necessary step towards the autonomous platforms (SOLO-SURGERY and LAPARO2.0-SURGERY) because it will allow the acquisition of the training data for the cognitive and perception modules. The following outcomes will be exploited:

- A validation of the assistive robotic arms in terms of workspace, usability, accuracy, velocity
- The identification of the nominal trajectories for the tools during the entire surgical procedure, *i.e.* the pose of each tool as moved by the main surgeon and by the assistant surgeon. This will help developing the mathematical model of their individual actions and mutual interactions
- An analysis of the correlation between the actions performed by the two surgeons and their verbal communication for designing and training the speech recognition module
- A validation of the user interface for the main surgeon at the "enhanced" daVinci console and the novel interface for the assistant surgeon
- The collection of data during surgical interventions to be used to train the autonomous systems in the next platforms.

2 System architecture

The MULTIROBOTS-SURGERY platform is designed as a collection of functional modules, each one devoted to a single task. The tasks assigned to each module are detailed in deliverable D7.1 – Technical Specifications, while the functional architecture is shown in Figure 1.

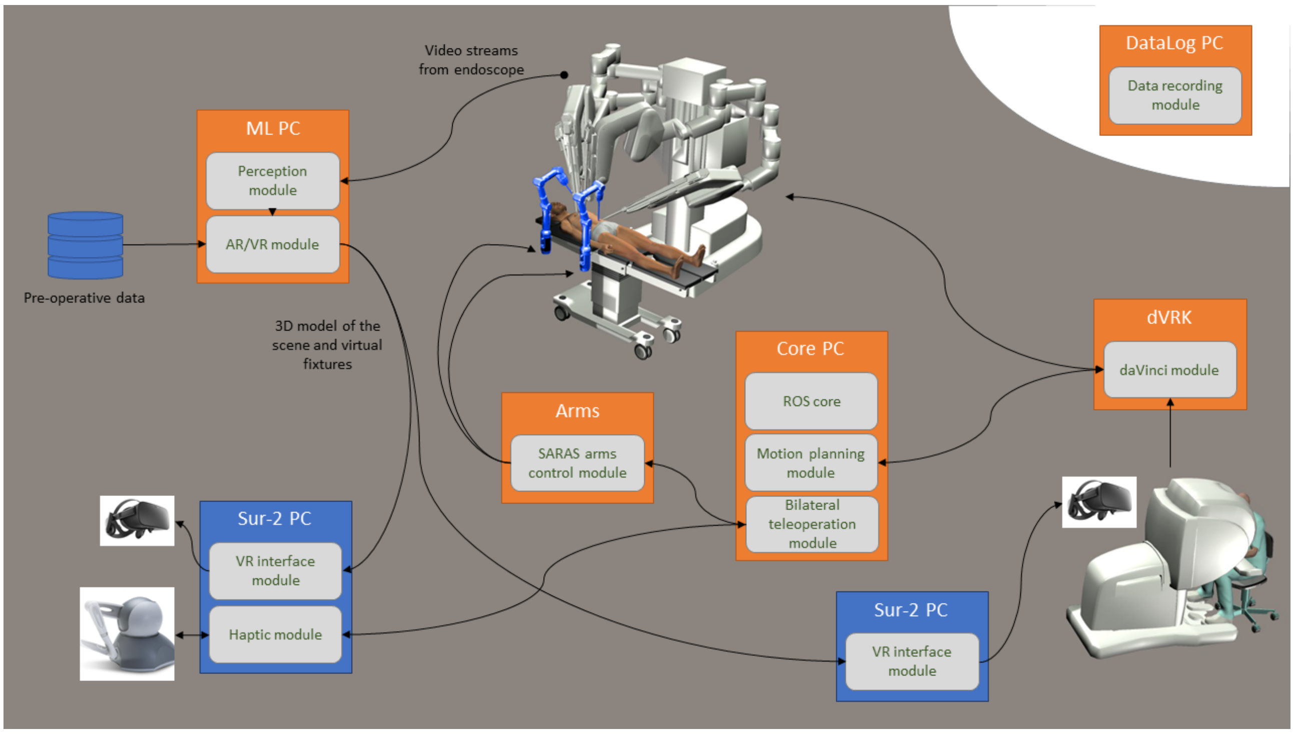


Figure 1: MULTIROBOTS-SURGERY platform functional architecture.

We implemented the MULTIROBOTS-SURGERY platform as a collection of ROS nodes. All the ROS nodes are connected to a central unit acting as ROS core, in charge of providing a common timestamp and driving the communication between nodes, and data recording module. Figure 2 shows the conceptual architecture of MULTIROBOTS-SURGERY platform in terms of ROS nodes. For the full list of nodes and how they are connected please refer to Figure 5.

2.1 ROS core and data recording node

ROS is an open source, distributed, component-based development framework that has become the standard *de facto* for interfacing heterogeneous hardware and software in robotics applications. Its versatility allows it to be used effectively in heterogeneous scenarios ranging from quad-copter control to surgical teleoperation systems, as in the dVRK (refer to Section 2.2 for more information). The focus for ROS is to foster an easy way for different institutions to cooperate by offering their independently-working applications through a standardized set of tools that allows them to communicate and process data efficiently. The main characteristics are:

- The **message passing system**, which revolves around a standardized data representation (message types) transmitted as a communication channel abstraction called topic;

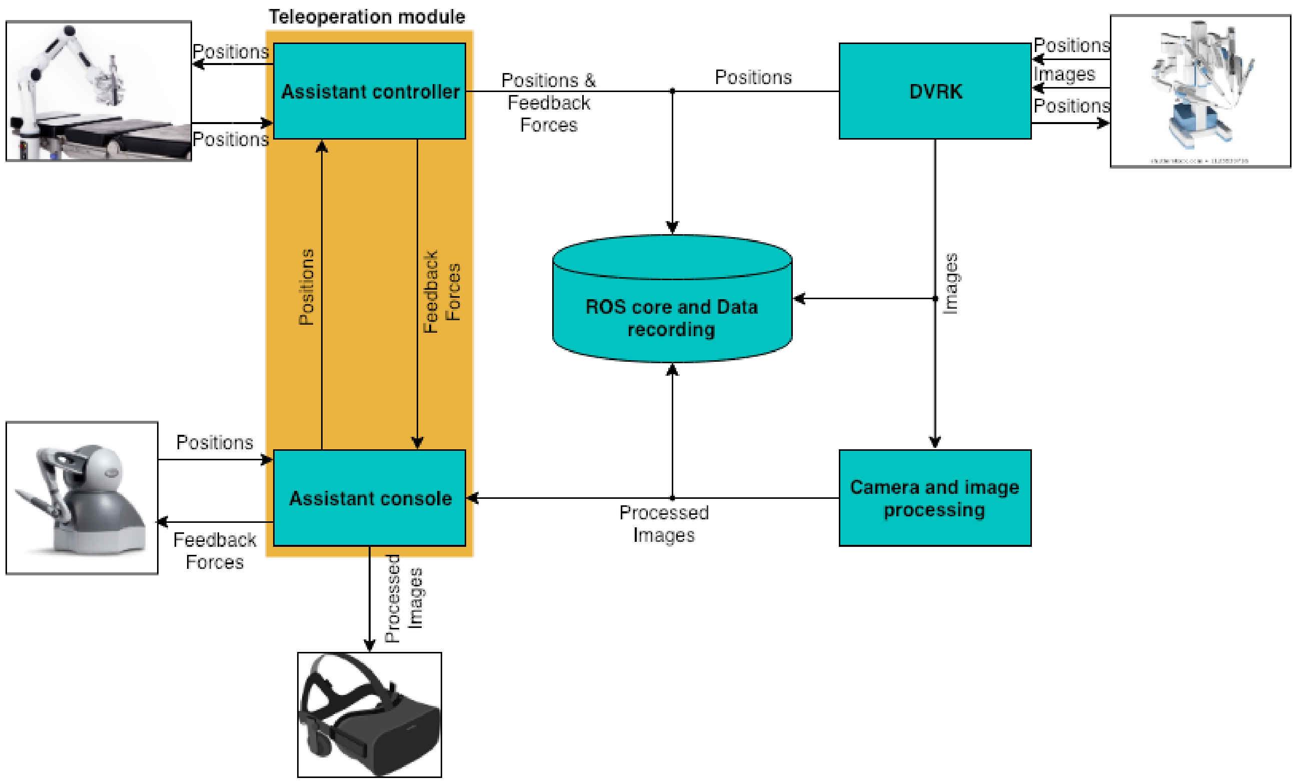


Figure 2: MULTIROBOTS-SURGERY platform conceptual ROS architecture.

- A **distributed multi-process architecture** that separates different functionalities between nodes, *i.e.* an abstracted process entity, which has the additional benefit of improving the overall system stability;
- **Common drivers** for various hardware configurations, tested by institutions worldwide for increased reliability and ease-of-access to data.

Communication in-between nodes is organized as a star-shaped network in which the central node, called *ROS core*, operates as a connection broker while also maintaining a message-driven common time-stamp for the entire network. The low-level connection is orchestrated via an asynchronous input/output library over TCP/IP. The ROS core module works also as Domain Name System (DNS) in the local network of nodes. A node that needs to subscribe to a topic published from another node for the first time, communicates with the ROS core to locate the node in the network, and then, it establishes a point-to-point communication. This mechanism does not overload the ROS core of work and makes the communication between nodes faster.

The ROS core and data recording module embed an efficient global time synchronization policy between modules through Network Time Protocol (NTP). NTP is a networking protocol for clock synchronization between computer systems over packet-switched, variable-latency data networks. NTP is intended to synchronize all participating computers to within a few milliseconds of Coordinated Universal Time. The connection between all modules to the NTP server permits the time synchronization and the precise messages exchange handling. It is possible to order the messages exchanged between nodes accurately, avoiding the reception from nodes of messages in different order with respect to the sending order.

The standard ROS library provides a functionality to record the messages, or a subset of them, exchanged in the network, in files called *rosbag*. These files are then playable inside the ROS environment. The recording functionality has been efficiently integrated into the ROS core and data recording module. Since all the nodes in the SARAS node net are connected to the ROS core and data recording module, and all nodes are synchronized, it is possible to record the messages exchanged between all nodes in the ROS environment. The recording of real data allows to store permanently experiments done with the architecture and to replay them other times in order to study the behavior of the whole system or particular phenomena happening during the experiments. The recording include:

- Videos from the da Vinci endoscope
- da Vinci arms kinematics
- SARAS arms kinematics
- Audio streams
- Commands executed in master consoles.

2.2 dVRK node

The da Vinci Research Kit (dVRK) is an "open-source mechatronics" system, consisting of electronics, firmware, and software that is being used to control research systems based on the first-generation da Vinci system [2]. Due to its license, it easily allows implementing new algorithms at any level of control for the da Vinci. In the SARAS system, the da Vinci is used by the main surgeon to perform the main surgical task. The surgeon teleoperates the da Vinci arms and endoscope from the standard da Vinci master console. The dVRK has a ROS wrapper used to publish all the information regarding the arms kinematics, endoscope positions and images to several ROS topics. Each node in the ROS node net can read the above-mentioned pieces of information to implement features like collision avoidance and image processing. In the SARAS environment, the dVRK node publishes the endoscope message under the namespace `endoscope`, publishing Raw, and compressed images. The dVRK node publishes also the robot's joints states and arm positions under the namespace `PSM1`, `PSM2` and `ECM` with respect to which arm the data refers.

2.3 Bilateral teleoperation module

The bilateral teleoperation module is a set of ROS nodes which implements a multi-master/multi-slave bilateral teleoperation, where two users cooperate on a shared environment by means of a telerobotics setup. In our scenario, the main surgeon controls the da Vinci tools, while the assistant surgeon teleoperates standard laparoscopic tools mounted at the end-effector of the SARAS robotic arms through other haptic devices. The module includes different nodes with different functionalities. The nodes main task in bilateral teleoperation module is to send commands to both master and slave devices and to get the status (*i.e.* the joint states and the end-effector position) of both master and slave devices. The ROS interface allows to retrieve the above mentioned functionality on the ROS environment. The bilateral teleoperation module consists of two sub-modules which represent the two agents involved in the teleoperation:

1. Master console, also called the Assistant console.
2. Slave robot, also called the Assistant robot.

In this section, we will detail all the nodes and software present in both sub-modules and which is their role inside the teleoperation architecture.

2.3.1 Assistant console module

The assistant console module consists of both hardware and software parts. The hardware setup is composed by:

- Two **Simball** devices used by the assistant surgeon to teleoperate the two assistant robots. Simball is commonly used to train surgeons on laparoscopic operations, due to its ability to emulate with realism the feeling of a real laparoscopic instrument (in particular the mechanical constraint of the trocars through which the instruments are inserted into the peritoneoum of the patient).
- Two **Touch haptic** devices, motorized devices that apply force feedback on the user's hands, allowing him/her to feel virtual objects and producing true-to-life touch sensations. The Simball device is bind to the end effector of *Touch haptic* device, allowing to propagate the force feedback generated by the haptic device to the surgeon's hands.
- An **Oculus Rift** device used to stream the da Vinci endoscope images with augmented informations (*e.g*. upcoming collisions).

Figure 3 shows the master console hardware setup used by the assistant surgeon during an experimental surgical operation.

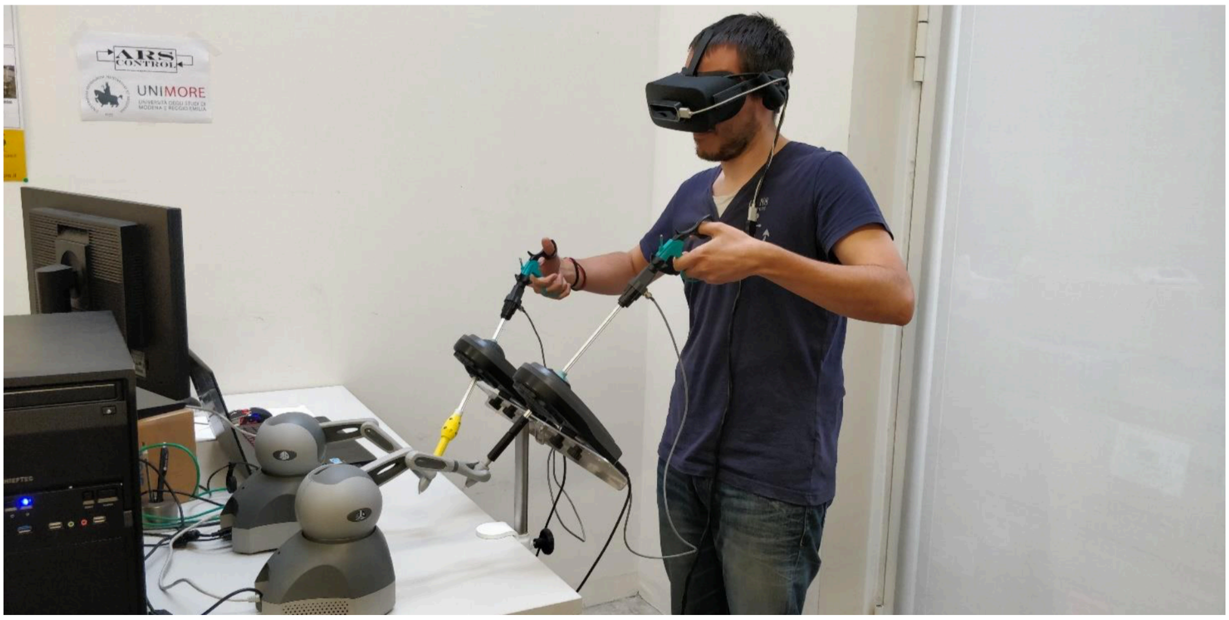


Figure 3: MULTIROBOTS-SURGERY assistant master console.

The assistant console software runs over a Windows 10 machine. The software embedded in this module is responsible to communicate with the haptic devices and to get the images from the endoscope camera and display them inside the Oculus Rift displays.

The list of the software embedded in the assistant console module is:

Touchball controller interfaces with the two haptic devices and Simball devices. The software main functionality is to get the current state of the devices (Simball and Touch haptic) and stream it in a UDP socket to make it available at the Slave side of the teleoperation. The software retrieves the states using drivers provided by the device manufacturers. The software embeds also the option to calibrate the haptic devices and the clipping movement of the Simball devices.

Oculus endoscope is the software which takes the images from the da Vinci endoscope and displays them in the Oculus Rift displays. It is developed in Unity. The software builds a 3D environment having two non-fixed cameras representing the Oculus display. The cameras rotate simultaneously with the headset, and they keep the images aligned in front of the cameras. This behavior makes the images of endoscope fixed in front of the eyes even if the surgeon moves the head. This gives more freedom to the surgeon respect to the fixed view of the da Vinci console.

2.3.2 Assistant robot module

The Assistant console hardware is composed of two SARAS arms. The arms are formed by two mechanical parts: a passive arm with seven degrees of freedom, which allows to positioning the manipulator in a safe place before the operation starts, and an actuated redundant robot with four motors that moves the endoscopic tool. The robot controller guarantees the robot movements pivoting around a prior fixed remote center of motion.

The software responsible for the teleoperation is developed both as standard C++ and C++ ROS nodes. The choice to use parts of the code as standard C++ derives from the uses of different operating systems on the machine hosting the software. The master console is running Windows 10 to better interface with the Touch haptic device drivers, while the slave console is running Ubuntu 16.04, a Linux distribution, to have better integration with the ROS environment.

The SARAS arms are connected to the assistant controller computer through Ethernet cables. The robot's software exposes several control methods such as the setting of a target position for the end effector, the set of a maximum motor velocity, the opening, closing and rotating of the clip or scissors instrument. It exposes also methods to read the joints angles, and end effector position respect to a common reference system placed in the center of the robot base. The communication with the robot happens through a particular WebSocket implementation called *Socket.IO*, a library that enables real-time, bidirectional and event-based communication between client and server sides.

The list of the software embedded in the assistant robot module is:

SARAS arms drivers a ROS node which compiles and makes available as ROS package the `libmesa` library provided by Medineering GmbH, which is written in straight C++. The library main functionality is to wrap the methods exposed by the robot, to make them available in C++ code.

SARAS arms state a ROS node which interfaces with SARAS arms drivers node to retrieve the robot's state (*i.e.* joint angles, end-effector pose and remote center of motion position) and publish it in a ROS topic, making it available from all nodes connected to the same ROS core.

Teleoperation a set of ROS nodes which implement the bilateral teleoperation algorithms described in Section 5. Its main functionalities are to keep the passivity of the system and to have transparent teleoperation. It interfaces with the SARAS arms state node and Touchball receiver node to send target position commands to the robot and force commands to the haptic device.

Touchball receiver a ROS node in communication with the Touchball controller. The work-flow of this node is to listen to a prior chosen port and IP address (*i.e.* where Touchball Controller is streaming the state) and publish all the data received in specific ROS topics.

SARAS visualization a ROS node which allows to show the robot's state in a real-time simulation environment. The simulation environment is built in Rviz (ROS visualization) which is integrated with ROS and ready to use. In order to show the real behavior of the robot in a simulation environment, it needs to know the state of the joints and the robot model. The state of the robot is retrieved by the SARAS arms state node, while the robot model is composed by the robot mesh files and the URDF (Unified Robot Description Format, an XML format for representing a robot model). Rviz allows to show also a simulation of the agents behavior involved in the teleoperation as a response of certain simulation input, without having the real agents attached to the system. Figure 4 shows the simulation environment with both assistant robot arms, and the haptic devices displayed.

2.4 Camera and Image processing module

The Camera and Image processing module is a set of ROS nodes which includes all the procedures to calibrate the system –*i.e.* to set a common reference frame for all the agents involved in the teleoperation– and the processing of the images from the da Vinci endoscope.

The module embeds the following nodes:

Global frame registration a ROS node used for calibration purposes. It allows setting a unique shared reference frame for the da Vinci and the SARAS arms (see Section 4).

Decklink ROS a ROS node used to convert analog images into digital ones. This conversion is necessary in order to elaborate and process the images. The analog images flow from the da Vinci endoscope to the Decklink acquisition board, then the ROS node publishes the node in specific topics to make them available all over the ROS net.

Colliding regions a ROS node that computes the colliding regions between the instruments of the da Vinci and SARAS arms or between the robot instruments and the bones, or other forbidden anatomical regions (see Section 6).

Image processing a ROS node responsible to add virtual fixtures to the endoscope images, *e.g.* highlighting regions where a potential collision is estimated (see Section 7).

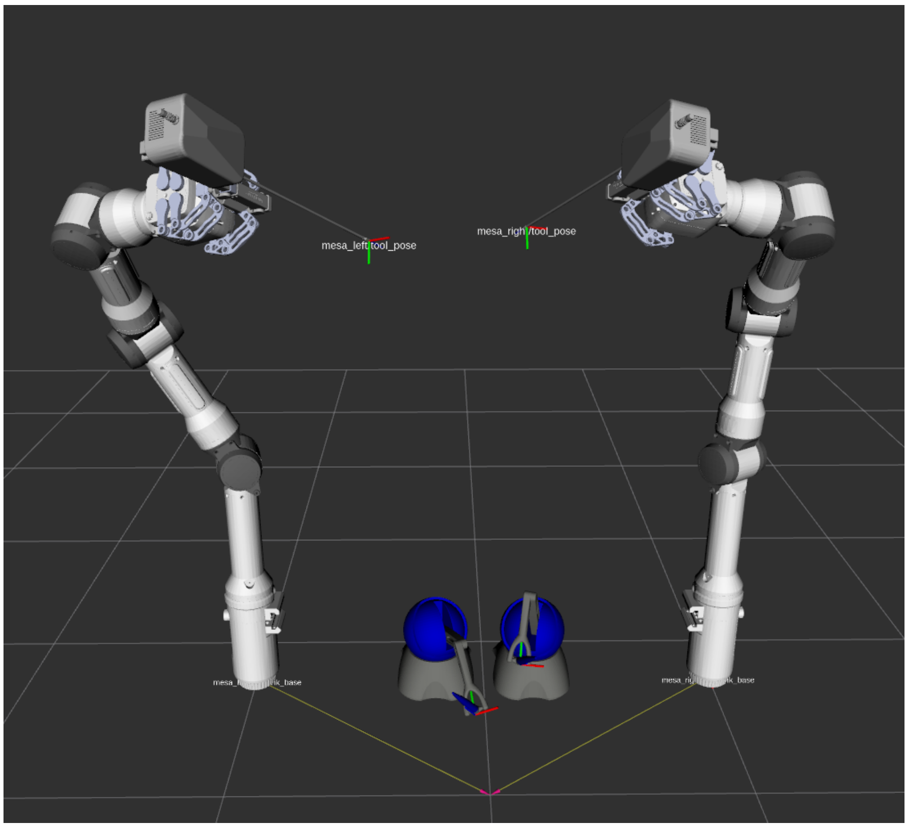


Figure 4: The simulation environment showing the actual state of the assistant robots and haptic devices.

Figure 5 shows a full representation of all the nodes involved in the bilateral teleoperation setup. It reports all the connections between nodes and data exchanged between them. In this figure the ROS core module has been omitted to better represent the other communications; actually all communication channels converge to the ROS core and data recording module to be then redirected to the destination node.

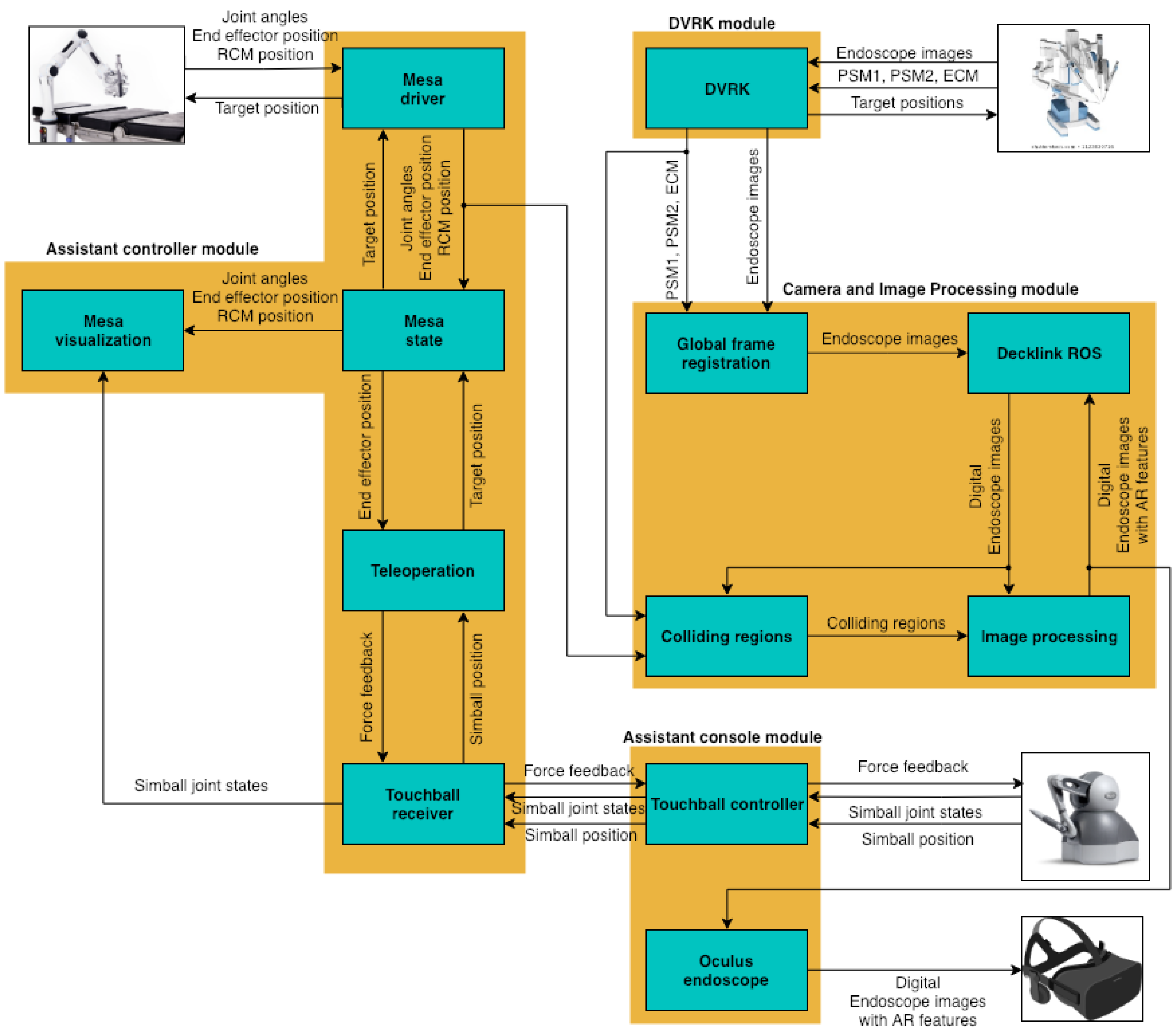


Figure 5: MULTIROBOTS-SURGERY platform detailed ROS architecture.

3 Hardware implementation

The two SARAS arms at the slave side of the platform are composed of three subsystems:

1. the passive **Positioning Arm** is needed for the rough and static positioning of the laparoscopic instruments.
2. the **Endoscope Robot** is the active robot for the fine positioning of the laparoscopic instruments teleoperated by the assistant. It guarantees the safe pivoting of the instruments through the trocar.
3. the **SARAS Adapter** which provides two further degrees of freedom: the rotation of the instruments along its main axis and the opening/closing of the tools (forcep and grasper).

Such mechatronics systems were already described in D7.1 – Technical Specifications: In this deliverable we provide more in details on the SARAS Adapter that is the main outcome from the hardware development point of view.

3.1 SARAS Adapter

The SARAS adapter consists of three segments and has two degrees of freedom. A description of the three segments and their parts is provided in Table 1 and Figure 6.

Segment	**Parts of the segment**	**Description**
Segment 1 (S1) Linear components	SARAS S1-1	Frame
	SA RAS S1-2	Motor including spindle and nut
	SARAS S1-3	Encoder
	SARAS S1-4	Gear, friction bearing and bearing block
	SARAS S1-5	Blocking slot for instrument
	SARAS S1-6	Frictionless table
	SARAS S1-7	Slider
Segment 2 (S2) Rotatory components	SARAS S2-1	Frame
	SARAS S2-2	Motor and gearbox
	SARAS S2-3	Encoder
	SARAS S2-4	Gear and drivebelt
Segment 3 (S3) Electronic/ Software	SARAS S3-1	Circuit board
	SARAS S3-2	Electronic LEMO connectors

Table 1: Components of the SARAS adapter.

The SARAS adapter is a mechatronic device for holding and guiding endoscopic instruments. The adapter can be used for surgical instruments like grasping forceps in the laparoscopy and allows the

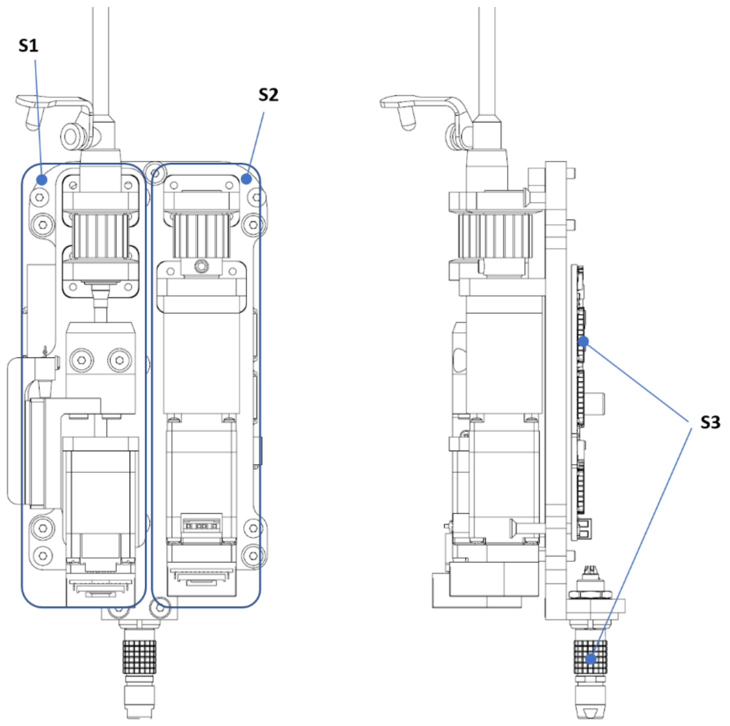


Figure 6: Segments and basic dimensions of the adapter top view, with attached instrument.

instrument to open, close and rotate. The internal part of the SARAS adapter is composed of a linear mechanical part (consisting of motor, encoder, gear and bearings) and a rotatory mechanical part (consisting of motor, encoder and gear). The linear part also contains a port providing a connectivity between instrument and adapter. Underneath the software and electrical system is mounted to the frame. It is subdivided into the power supply and the printed circuit board. Altogether these components are surrounded by the housing which provides a mechanical interface to the linear axis of the Medineering Endoscope Robot.

The SARAS adapter can be used with the Medineering Robotic Endoscopy which contains the Positioning Arm, the Endoscope Robot and an input device (*e.g.* foot pedal). Power supply and data supply of the adapter is provided by the Endoscope Robot via an according plug connector.

In Figure 7 is shown: a) isometric view of the adapter; (1) mechanical interface; (2) housing; (3) instrument. b) interior-top view; (4) port; (5) rotatory part; (6) linear part. c) interior-back view; (7) circuit board; (8) power supply.

3.2 Demands to the System

3.2.1 Modularity

The SARAS system's components can be separated and recombined. Primarily the Endoscope Robot is connected to the Positioning Arm via a mechatronic interface. Further, the SARAS adapter is linked with the Robot through a mechanical interface. As these interfaces are standardised within the Medineering product range, the three different modules can be exchanged by – for example – a stronger Positioning Arm or another SARAS Adapter for different instruments in the field of

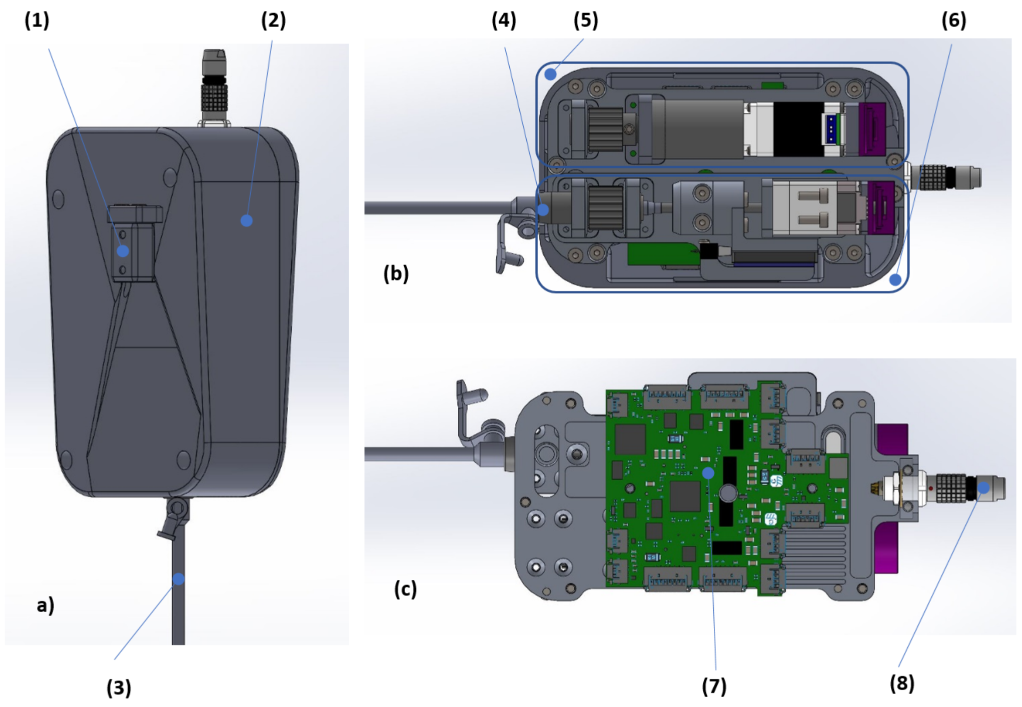


Figure 7: Side, top, and back view of the SARAS adapter.

laparoscopy. This modularity leads to the benefit of flexibility and variety in use. Initially, the first SARAS Adapter is dimensioned for all instruments of the Karl Storz Click Line Series.

3.2.2 Inter-Operability

The mechatronic interface supplies the Endoscope Robot with power and data, and moreover, a LEMO cable provides the same connection between Robot and Adapter. A web interface operates through certain commands as an input device for the Positioning Arm. The integrated electronic of the Positioning Arm can be used to supply the connected Endoscope Robot via the intern bus system with all sensor data of the arm. All in all, the three modules are compatible for mutual communication.

3.2.3 Dependability

The Positioning Arm receives its power from a standard 230 V socket and transfers it to the other modules. So the Endoscope Robot needs the Positioning Arm and the SARAS Robot would not work without the combination with the Endoscope Robot. The arm represents the basis of the whole system and can operate autonomously.

3.2.4 Safety Requirements

The Positioning Arm and the Endoscope Robot provide sufficient safety precautions to prevent damage, as a detailed hazard analysis led to the implementation of different actions. As the SARAS

System generates a magnetic field it must not be used closer than 200mm to patients wearing a pacemaker. To prevent the possibility of an electric shock the user should not touch the mechatronic interface of the Positioning Arm and the patient at the same time. Before use of the System it has to be ensured that the Positioning Arm is firmly mounted to the OR table. None of the SARAS system's components – except the instrument – must get in touch with the patient. The payload of the Positioning Arm must never exceed 2kg. As the prescribed payload of the Endoscope Robot is set to 0,5kg the SARAS Adapter including its instrument should not be heavier. A violation of those limits can lead to injuries of the patients. The user always has to keep in mind that once the brakes are opened all the weight is in the user´s hands.

3.2.5 Sterility

The components of the SARAS System are not sterile. To prevent a danger of infection during surgery, all components are covered with a sterile drape to provide sterility. Here the sterile border is located between the instrument and the adapter. Surgical instruments are sterilized in clinical standard procedures.

3.2.6 Robot Dexterity

The Positioning Arm possesses 7 degrees of freedom (DOFs). Through this increase of the usual six DOFs the Arm can be moved to different positions without moving the distal end. The kinematics and the linear axis of the robot provides its five DOFs. It can either be used parallel within one plain or pivoting around a certain pivot point. On top of that, the SARAS adapter has two degrees of freedom. All in all, the rough positioning is performed by the Positioning Arm, whereas the Endoscope Robot is responsible for the fine positioning. Finally, the SARAS Adapter with its inserted laparoscopic instrument provides gripping, closing and opening of the instrument.

3.2.7 Plug and unplug-mechanism

The first prototype of the SARAS Adapter doesn't possess a mechanism for plugging and unplugging the instrument yet. The instrument is a fixed integrated part of the SARAS Adapter. For the later instruments a mechanism will be developed for an easy plugging and unplugging of the instrument and thus, an easy exchange of the instruments.

4 Multirobots Calibration procedure

The MULTIROBOTS-SURGERY setup is a complex multi-robot system where five independent arms cooperate to accomplish a task in a common workspace. Four arms are devoted to manipulation tasks, *i.e.* two arms of the da Vinci system managed by the main surgeon and two SARAS arms controlled by the assistant surgeon, while the last one is used for image acquisition. The da Vinci arms are called:

PSM1 Patient side manipulator 1

PSM2 Patient side manipulator 2

while the SARAS arms will be called:

APSM1 Assistance patient side manipulator 1

APSM2 Assistance patient side manipulator 2

Finally the endoscope is attached to the third da Vinci's arm and is called:

ECM Endoscope control manipulator

To guarantee safety in collaboration between multiple robots, it is crucial to allow all the systems to refer to a common reference frame, usually called *world reference frame*. In other words, we need to compute the transformation matrices holding between each pair of local reference frames to which different robots refer to.

To account the estimation of these relations, we have to address the following issues:

1. Endoscope intrinsic and extrinsic parameters estimation
2. Hand-eye calibration between the ECM arm and the endoscope cameras
3. Endoscope and arms base frame calibration

It's important to point out that the camera parameters estimation and the hand-eye calibration are related to the da Vinci robot and so are not task dependant. Differently, the arms and endoscope base calibration is mandatory every time the positioning arms of the da Vinci and/or the SARAS arms are moved from the previous calibrated positions.

4.1 Endoscope intrinsic and extrinsic parameters estimation

The camera parameters estimation is mandatory to transform the features position extracted in the image plane to the Euclidean space where the manipulators act.

The procedure to estimate the intrinsic parameters of a camera is well known in the literature, and the most common technique is based on the Zhang approach [4]. Briefly, as shown in Figure 8, a

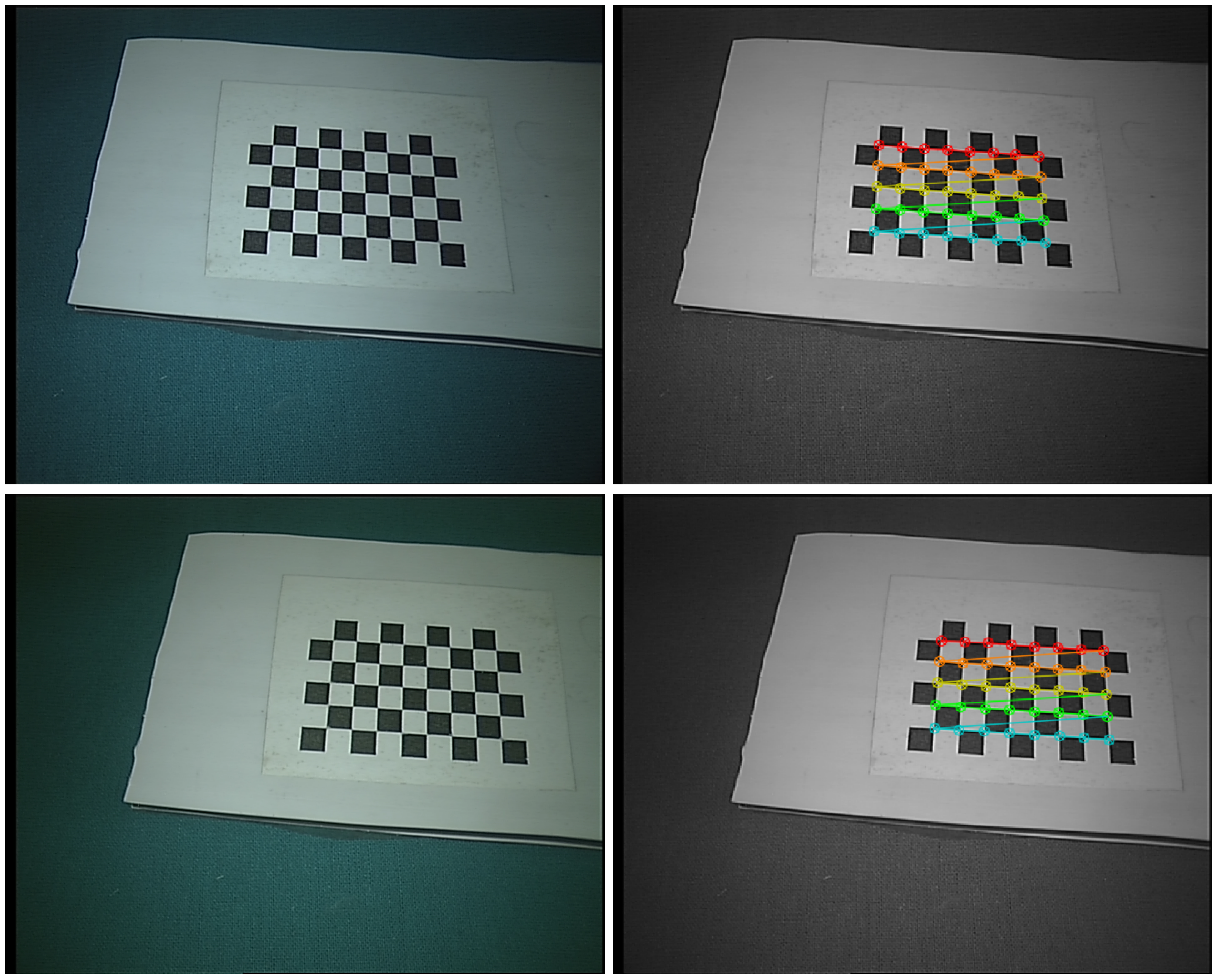


Figure 8: The calibration board used to estimate the intrinsic and extrinsic parameters estimation of the da Vinci endoscope.

specific pattern (a black and white checkerboard) has been shown and moved in front of the stereo-camera (*i.e.* the endoscope); the corners are detected and an optimization algorithms allows to minimize the back-projection error assuming a pinhole camera model and a polynomial deformation model.

For stereo vision system, besides the intrinsic parameter estimation, it is necessary to estimate also the transformation –*i.e.* the relative translation and rotation– between the two cameras. To this purpose, we used the `camera_calibration` package, provided by the ROS community, that makes available camera calibration algorithm implemented in the the OpenCV library.

4.2 Hand-eye calibration

The robotics hand-eye calibration is the task of computing the relative 3D position and orientation between the camera reference frame and the ECM's end-effector. In our case the camera is rigidly connected to the ECM arm, and this configuration is commonly referred to as *eye-on-hand* [3].

This procedure is crucial because the features extracted by the vision sensor are relative only to the camera reference frame, which is not fixed with respect to the world reference frame since the endoscope is hold by the ECM arm, which is moved by the main surgeon during the surgical procedure.

Figure 9 shows how the Hand-eye calibration technique works. The operator moves the ECM arm of the da Vinci system in different and redundant poses. At each step, the ECM's end effector pose and the measured marker pose are stored. Then an optimization problem in the form $AX = XB$ is solved in order to find the relative transformation between the end effector and the camera reference frames. We use the implementation of [3] provided by the `visp_hand2eye_calibration` and `easy_handeye` packages.

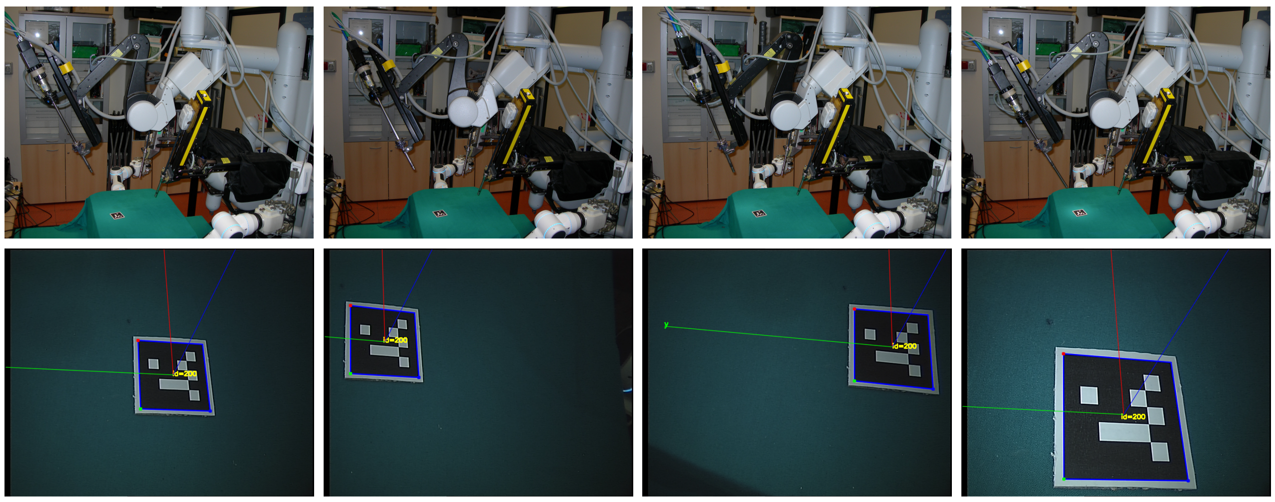
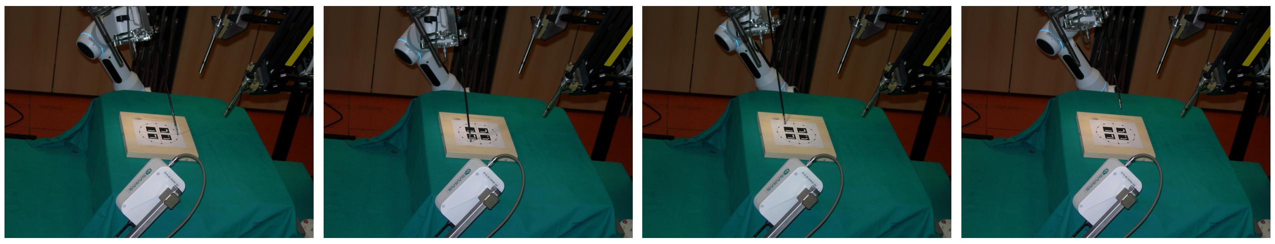


Figure 9: To account the hand-eye calibration the ECM is manually moved on different poses while it acquires the relative position of a fiducial marker.

4.3 Base frame calibration

This procedure aims at estimating a set of Euclidean transformations between a fixed frame –the base frame– and the local reference frames of each arm. The proposed calibration technique is based on the *three points* calibration method commonly used in robotic tool calibration routine [1]. This procedure uses the forward kinematics to estimate the relative pose of a circular pattern called *positioner* with respect to the robot's base frame. This procedure is repeated for each robot creating a set of transformations between the positioner plane and the robots base frame.

In our solution we use a printed calibration pattern, shown in Figure 10, which is composed by a set of black dots lying on a circle and by four fiducial markers lying on a concentric circumference. The black dots are the target points touched by the arms, and the fiducial markers are the positioners used by the endoscope. The printed board has an outer circle radius of 100 mm, the black dots have a diameter of 5 mm and the fiducial markers have a side of 25 mm. The fiducial markers are needed to use the same calibration procedure also for the endoscope. Since the endoscope is unable to touch the calibration pattern due to the mechanical limitation of the ECM kinematics, we estimate the relative pose of the positioners using a marker detector library called `ArUco` which provides the relative pose and orientation of each marker with respect to to the camera frame and then by the hand-eye calibration with respect to the ECM.



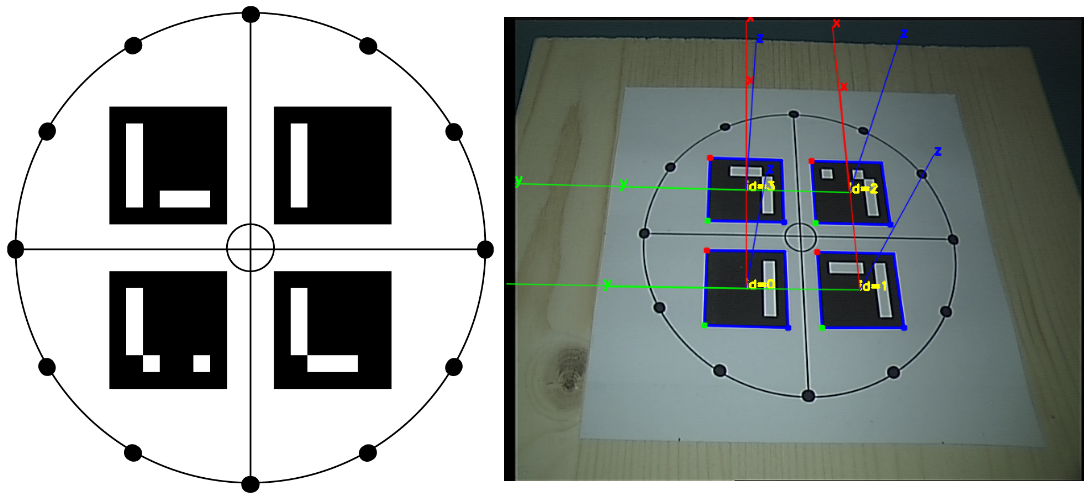


Figure 10: The calibration pattern and the printed version used during the base frame calibration.

The calibration algorithm is implemented in a ROS node called `global_frame_registration`.

5 Bilateral Teleoperation

The bilateral teleoperation architecture is divided into the following set of ROS nodes:

tf_transformer computes the main frame transformations between the master reference frames and the slave reference frames. In particular, (1) it translates the master position information provided by the `touchball_reciever` node from the Simball reference frame to the SARAS arm reference frame, and (2) it translates the force information from the SARAS arm reference frame to the Touch haptic reference frame. At the same time it computes the master pose frame velocity with respect to the slave reference frame, in order to command properly the robots.

tank_master deals with the management of the energy of the masters with the aim of preserving the stability of the master system, exploiting the passivity theory. Once the energy is computed, it provides the force set-points to the master devices.

tank_slave deals with the management of the energy of the slaves. Once the energy is computed, it provides the velocities set-points to the slave devices.

transparency provides the desired set points for the master and slave sides to the corresponding energy tank (force for the master side and velocity for the slave side), in order to implement the desired type of control.

velocity_integrator send the desired position information to the SARAS arms. It takes as inputs the initial position of the arm and the desired velocity. Then, integrating, it computes the position set-point.

The `tf_transformer` and the `velocity_integrator` nodes are designed to manage only one device at a time. For this reason, within the architecture, a node will run for the right channel and another one for the left channel. The `tank_master`, the `tank_slave` and the `transparency` nodes are designed to manage multiple devices at the same time. For this reason only one instance of each of them will be launched within the architecture.

In the deliverable D3.1 – Multi-modal human-robot interfaces and architecture for the MULTIROBOTS-SURGERY platform more details about the theoretical aspects of the bilateral teleoperation are provided. Here we just report pieces of code to explain how the theoretical results are implemented in the SARAS architecture.

5.1 `tf_transformer` node

First, the code wait at least a time equal to 2.5 seconds in order to verify that the reference systems of which we want to identify the transforms are actually available within `tf`.

```
while (t_new - t_old < 2.5){
    // Pose tracking
    try {
        pose_tracking_start = true;
        tf_listener.waitForTransform(pose_reference_tf_frame_name,
                                     pose_tracking_tf_frame_name,
                                           ros::Time(0),
                                           ros::Duration(10.0));
        tf_listener.lookupTransform(pose_reference_tf_frame_name,
                                    pose_tracking_tf_frame_name,
                                    ros::Time(0),
                                    tracked_pose_transform);
    } catch (tf::TransformException ex) {
        pose_tracking_start = false;
    }

    // Twist tracking
    try {
        twist_tracking_start = true;
        tf_listener.lookupTwist(pose_tracking_tf_frame_name,
                                pose_reference_tf_frame_name,
                                pose_tracking_tf_frame_name,
                                tf::Point(0.0, 0.0, 0.0),
                                pose_tracking_tf_frame_name,
                                ros::Time(0),
                                ros::Duration(0.001),
                                tracked_twist.twist);
        } catch (tf::TransformException ex) {
            twist_tracking_start = false;
        }

    // Wrench tracking
    try {
        wrench_tracking_start = true;
        tf_listener.waitForTransform(wrench_tracking_tf_frame_name,
                                     wrench_reference_tf_frame_name,
                                     ros::Time(0),
                                     ros::Duration(10.0));
        tf_listener.lookupTransform(wrench_tracking_tf_frame_name,
                                    wrench_reference_tf_frame_name,
                                    ros::Time(0),
                                    tracked_wrench_transform);
    } catch (tf::TransformException ex) {
        wrench_tracking_start = false;
    }

    t_new = ros::Time::now().toSec();

    if ((pose_tracking_start == true) &&
        (twist_tracking_start == true) &&
        (wrench_tracking_start == true)){
            break;
        }
}
```

If all transformations are available the algorithm stores the static wrench transformation and then proceeds, otherwise an error message is displayed.

```
if ((pose_tracking_start == true) &&
    (twist_tracking_start == true) &&
    (wrench_tracking_start == true))
        {
```

```
    std::cout << "Start␣tracking␣succeed!" << std::endl;

    // Setting up kdl wrench transformation
    tracked_wrench_transform_kdl.p[0] =
        tracked_wrench_transform.getOrigin().x();
    tracked_wrench_transform_kdl.p[1] =
        tracked_wrench_transform.getOrigin().y();
    tracked_wrench_transform_kdl.p[2] =
        tracked_wrench_transform.getOrigin().z();

    tracked_wrench_transform_kdl.M = KDL::Rotation::Quaternion(
        tracked_wrench_transform.getRotation().x(),
        tracked_wrench_transform.getRotation().y(),
        tracked_wrench_transform.getRotation().z(),
        tracked_wrench_transform.getRotation().w());

    while (public_node_handler.ok()) {
        //main loop
        ...
        }
    }
    else
        {
                std::cout << "Start␣tracking␣failed!" << std::endl;
        }
```

The main loop is organized as follow:

the master pose frame related to the slave base frame is continuously tracked and stored into a ROS `geometry_msgs/PoseStamped` message.

```
// Pose tracking
try {
    tf_listener.waitForTransform(pose_reference_tf_frame_name,
                                 pose_tracking_tf_frame_name,
                                 ros::Time(0),
                                 ros::Duration(10.0));
    tf_listener.lookupTransform(pose_reference_tf_frame_name,
                                 pose_tracking_tf_frame_name,
                                 ros::Time(0),
                                 tracked_pose_transform);
} catch (tf::TransformException ex) {
    ROS_ERROR("%s", ex.what());
}

tracked_pose.header.stamp    = ros::Time::now();
tracked_pose.header.frame_id = robot_namespace+"/sbs_world";

tracked_pose.pose.position.x = tracked_pose_transform.getOrigin().x();
tracked_pose.pose.position.y = tracked_pose_transform.getOrigin().y();
tracked_pose.pose.position.z = tracked_pose_transform.getOrigin().z();

tracked_pose.pose.orientation.x = tracked_pose_transform.getRotation().x();
tracked_pose.pose.orientation.y = tracked_pose_transform.getRotation().y();
tracked_pose.pose.orientation.z = tracked_pose_transform.getRotation().z();
tracked_pose.pose.orientation.w = tracked_pose_transform.getRotation().w();
```

The algorithm continuously tracks also the master pose frame velocity, filters the signals using a low pass filter and then stores the data into a ROS `geometry_msgs/TwistStamped` message.

```
// Twist tracking
try {
    tf_listener.lookupTwist(pose_tracking_tf_frame_name,
                            pose_reference_tf_frame_name,
                            pose_tracking_tf_frame_name,
                            tf::Point(0.0, 0.0, 0.0),
                            pose_tracking_tf_frame_name,
                            ros::Time(0),
                            ros::Duration(0.001),
                            tracked_twist.twist);

} catch (tf::TransformException ex) {
    ROS_ERROR("%s", ex.what());
}
tracked_twist.twist.linear.x  =
    low_pass_filter(tracked_twist.twist.linear.x,
                   old_tracked_twist.twist.linear.x,
                   1.0 / frequency,
                   low_pass_filter_frequency);
tracked_twist.twist.linear.y  =
    low_pass_filter(tracked_twist.twist.linear.y,
                   old_tracked_twist.twist.linear.y,
                   1.0 / frequency,
                   low_pass_filter_frequency);
tracked_twist.twist.linear.z  =
    low_pass_filter(tracked_twist.twist.linear.z,
                   old_tracked_twist.twist.linear.z,
                   1.0 / frequency,
                   low_pass_filter_frequency);
tracked_twist.twist.angular.x =
    low_pass_filter(tracked_twist.twist.angular.x,
                    old_tracked_twist.twist.angular.x,
                    1.0 / frequency,
                    low_pass_filter_frequency);
tracked_twist.twist.angular.y =
    low_pass_filter(tracked_twist.twist.angular.y,
                    old_tracked_twist.twist.angular.y,
                    1.0 / frequency,
                    low_pass_filter_frequency);
tracked_twist.twist.angular.z =
    low_pass_filter(tracked_twist.twist.angular.z,
                    old_tracked_twist.twist.angular.z,
                    1.0 / frequency,
                    low_pass_filter_frequency);

old_tracked_twist = tracked_twist;

tracked_twist.header.stamp = ros::Time::now();
```

It computes the wrench conversion of the commanded master forces, from the slave base frame to the Touch haptic reference frame using the transformation stored at the beginning. The result is stored into a ROS `geometry_msgs/WrenchStamped` message.

```
// Wrench conversion
    applied_wrench_kdl = tracked_wrench_transform_kdl * des_wrench;

    applied_wrench.header.stamp    = ros::Time::now();
    applied_wrench.wrench.force.x  = applied_wrench_kdl[0];
    applied_wrench.wrench.force.y  = applied_wrench_kdl[1];
    applied_wrench.wrench.force.z  = applied_wrench_kdl[2];
    applied_wrench.wrench.torque.x = applied_wrench_kdl[3];
    applied_wrench.wrench.torque.y = applied_wrench_kdl[4];
    applied_wrench.wrench.torque.z = applied_wrench_kdl[5];
```

The commanded master forces are captured exploiting the following callback function.

```
void in_wrench_callback(const geometry_msgs::WrenchStamped::ConstPtr& msg) {

    des_wrench[0] = msg->wrench.force.x;
    des_wrench[1] = msg->wrench.force.y;
    des_wrench[2] = msg->wrench.force.z;
    des_wrench[3] = msg->wrench.torque.x;
    des_wrench[4] = msg->wrench.torque.y;
    des_wrench[5] = msg->wrench.torque.z;
}
```

Once all data are computed, the algorithm sends the messages.

```
// Publishing
    tracked_pose_pub.publish(tracked_pose);
    tracked_twist_pub.publish(tracked_twist);
    out_wrench_pub.publish(applied_wrench);
```

5.2 `tank_master` node

First, the information relative to the cycle time are updated.

```
// Update dt
    time2 = ros::Time::now().toSec();
    dt    = time2 - time1;
    time1 = time2;
```

The energy stored in the tank need to be limited. The algorithm checks if the energy introduced in the tank needs to be stored or can be sent outside.

```
// Check if the tank is full
    if (T < p_Tmax)
        sigma = 1.0;
    else
        sigma = 0.0;
```

In order to implement the energy exchange between the tanks, the conditions for which energy need to be exchanged are verified.

```
// Check if the tank can provide energy
    if (T > p_Tava)
        beta = 1.0;
    else
```

```
        beta = 0.0;

// Check if the tank requires energy
    if (T < p_Treq)
        Ereq = 1.0;
    else
        Ereq = 0.0;
```

The value of the damping applied to the master devices depends on the energy stored in the tank and need to be computed.

```
// Evaluate damping value
    for (int i = 0; i < 6; i++) {
        p_Damping[i] = get_damping(p_Tmax, p_Tmin, p_Treq, p_Tava,
            p_Damping_std[i], p_Damping_max[i], T);
    }
```

If the energy in the tank is low, then a request of energy is sent to the slave tank.

```
// Send a request for energy if needed
    out_Ereq.data = Ereq;
    Ereq_pub.publish(out_Ereq);
```

The power associated to the damping is evaluated in order to update the evolution of the energy tank state and the energy stored.

```
// Evaluate the power associated to the damping
    D = 0.0;
    for (int i = 0; i < 6; i++)
        D += p_Damping[i] * pow(Right_measCartVel[i], 2) +
            p_Damping[i] * pow(Left_measCartVel[i], 2);
```

The power flows are then evaluated and the state is updated, bounding the energy level if needed.

```
// Evaluate Pout and read Pin
    Pout = (1 - sigma) * D + extEreq * beta * p_Pconst;
    out_Pout.data = Pout;
    Pout_pub.publish(out_Pout);

// Evaluate xt_dot, xt and T
    xt_dot = sigma * D + U + (sigma * Pin - Pout) / xt;
    xt = xt + xt_dot * dt;

    if (xt > xt_max)
        xt = xt_max;

    T = 0.5 * pow(xt, 2);
```

If the energy is low, a set-point reduction factor is applied to the desired set-points, and need to be computed.

```
// Evaluate decreasing set-point reduction factor
    KE = Ereq * (1 - (T - p_Tmin) / (p_Treq - p_Tmin));
```

The energy associated to the desired actions is evaluated.

```
// Compute the energy associated to the set-point
```

```
    deltaE = 0.0;
    for (int i = 0; i < 6; i++) {
        deltaE = deltaE + (Right_desWrench[i] * Right_measCartVel[i] * dt -
            KE * Right_desWrench[i] * Right_measCartVel[i] * dt);
        deltaE = deltaE + (Left_desWrench[i] * Left_measCartVel[i] * dt -
                  KE * Left_desWrench[i] * Left_measCartVel[i] * dt);
    }
```

If there is enough energy in the tank the desired actions can be implemented, otherwise not.

```
U = 0.0;
if ((T - deltaE) > p_Tmin) //  desired action is implementable
    for (int i = 0; i < 6; i++) {
        Right_tankWrench[i] = Right_desWrench[i] - KE * Right_desWrench[i];
        U = U - Right_tankWrench[i] / xt * Right_measCartVel[i];
        Left_tankWrench[i]  = Left_desWrench[i] - KE * Left_desWrench[i];
        U = U - Left_tankWrench[i] / xt * Left_measCartVel[i];
    }
else // desired action is not implementable
    for (int i = 0; i < 6; i++) {
        Right_tankWrench[i] = 0.0;
        Left_tankWrench[i]  = 0.0;
        U                   = 0.0;
    }
```

In order to physically implement a damping to the master device, the damping force is added to the implementable actions.

```
// Add damping feedback
    for (int i = 0; i < 6; i++) {
        Right_tankWrench[i] = Right_tankWrench[i] -
            p_Damping[i] * Right_measCartVel[i];
        Left_tankWrench[i]  = Left_tankWrench[i] -
            p_Damping[i] * Left_measCartVel[i];
    }
```

The resulting actions are then stored into a `geometry_msgs/WrenchStamped` message and then published.

```
// Write the implementable action
    out_Right_tankWrench.header.stamp    = ros::Time::now();
    out_Right_tankWrench.header.frame_id = "mesa_right/sbs_world";
    out_Right_tankWrench.wrench.force.x  = Right_tankWrench[0];
    out_Right_tankWrench.wrench.force.y  = Right_tankWrench[1];
    out_Right_tankWrench.wrench.force.z  = Right_tankWrench[2];
    out_Right_tankWrench.wrench.torque.x = Right_tankWrench[3];
    out_Right_tankWrench.wrench.torque.y = Right_tankWrench[4];
    out_Right_tankWrench.wrench.torque.z = Right_tankWrench[5];

    out_Left_tankWrench.header.stamp    = ros::Time::now();
    out_Left_tankWrench.header.frame_id = "mesa_left/sbs_world";
    out_Left_tankWrench.wrench.force.x  = Left_tankWrench[0];
    out_Left_tankWrench.wrench.force.y  = Left_tankWrench[1];
    out_Left_tankWrench.wrench.force.z  = Left_tankWrench[2];
    out_Left_tankWrench.wrench.torque.x = Left_tankWrench[3];
    out_Left_tankWrench.wrench.torque.y = Left_tankWrench[4];
    out_Left_tankWrench.wrench.torque.z = Left_tankWrench[5];

    Right_tankWrench_pub.publish(out_Right_tankWrench);
    Left_tankWrench_pub.publish(out_Left_tankWrench);
```

This procedure is continuously replicated in order to manage the energy in the system.

All the input data are collected using callback functions.

```
void Right_measCartVel_callback(
    const geometry_msgs::TwistStamped::ConstPtr& msg) {

    Right_measCartVel[0] = msg->twist.linear.x;
    Right_measCartVel[1] = msg->twist.linear.y;
    Right_measCartVel[2] = msg->twist.linear.z;
    Right_measCartVel[3] = msg->twist.angular.x;
    Right_measCartVel[4] = msg->twist.angular.y;
    Right_measCartVel[5] = msg->twist.angular.z;
}

void Right_desWrench_callback(
    const geometry_msgs::WrenchStamped::ConstPtr& msg) {

    Right_desWrench[0] = msg->wrench.force.x;
    Right_desWrench[1] = msg->wrench.force.y;
    Right_desWrench[2] = msg->wrench.force.z;
    Right_desWrench[3] = msg->wrench.torque.x;
    Right_desWrench[4] = msg->wrench.torque.y;
    Right_desWrench[5] = msg->wrench.torque.z;
}

void Left_measCartVel_callback(
    const geometry_msgs::TwistStamped::ConstPtr& msg) {

    Left_measCartVel[0] = msg->twist.linear.x;
    Left_measCartVel[1] = msg->twist.linear.y;
    Left_measCartVel[2] = msg->twist.linear.z;
    Left_measCartVel[3] = msg->twist.angular.x;
    Left_measCartVel[4] = msg->twist.angular.y;
    Left_measCartVel[5] = msg->twist.angular.z;
}

void Left_desWrench_callback(
    const geometry_msgs::WrenchStamped::ConstPtr& msg) {

    Left_desWrench[0] = msg->wrench.force.x;
    Left_desWrench[1] = msg->wrench.force.y;
    Left_desWrench[2] = msg->wrench.force.z;
    Left_desWrench[3] = msg->wrench.torque.x;
    Left_desWrench[4] = msg->wrench.torque.y;
    Left_desWrench[5] = msg->wrench.torque.z;
}

void Pin_callback(const std_msgs::Float32::ConstPtr& msg) {

    Pin = msg->data;
}

```

```
void extEreq_callback(const std_msgs::Float32::ConstPtr& msg) {

    extEreq = msg->data;
}
```

5.3 `tank_slave` node

The software architecture of the `tank_slave` node is the same of the `tank_master` node with the only difference that the desired action is a velocity and not a force. For this reason the `tank_slave` node software description is not reported here.

5.4 `transparency` node

The `transparency` node software architecture is organized into two states. In the first state, it waits until position information of all master and slave' robots are available. Then, it switches to the second state and stores the position information as initial positions.

```
if (state == 0) {
    if ((meas_right_master_pose_new_data == true) &&
        (meas_left_master_pose_new_data == true) &&
        (meas_right_slave_pose_new_data == true) &&
        (meas_left_slave_pose_new_data == true)) {
            for (int i = 0; i < 3; i++) {
                init_meas_right_master_pose[i] = meas_right_master_pose[i];
                init_meas_right_slave_pose[i]  = meas_right_slave_pose[i];
                init_meas_left_master_pose[i]  = meas_left_master_pose[i];
                init_meas_left_slave_pose[i]   = meas_left_slave_pose[i];
            }
        state++;
    }
} else {
    //State 1 description
    ...
}
```

The `new_data` flags are acquired by means of callback functions, as the position information. An example of callback function is reported here.

```
void meas_left_master_pose_callback(
    const geometry_msgs::PoseStamped::ConstPtr& msg) {

    meas_left_master_pose[0] = msg->pose.position.x;
    meas_left_master_pose[1] = msg->pose.position.y;
    meas_left_master_pose[2] = msg->pose.position.z;

    meas_left_master_pose_new_data = true;
}
```

In the second state the node continuously evaluates the position error between the master and the slave of each channel (left and right):

```
// compute right pose error
```

```
    for (int i = 0; i < 3; i++) {
        meas_right_pose_error[i] =
            (meas_right_slave_pose[i] - init_meas_right_slave_pose[i]) -
            (meas_right_master_pose[i] - init_meas_right_master_pose[i]);
    }

// compute left pose error
    for (int i = 0; i < 3; i++) {
        meas_left_pose_error[i] =
            (meas_left_slave_pose[i] - init_meas_left_slave_pose[i]) -
            (meas_left_master_pose[i] - init_meas_left_master_pose[i]);
    }
```

Then it computes the desired force set-points for the master side,

```
// Evaluate desired wrenches
    for (int i = 0; i < 3; i++) {
        des_right_wrench[i] = pose_error_gain * meas_right_pose_error[i];
        des_left_wrench[i]  = pose_error_gain * meas_left_pose_error[i];
    }

    for (int i = 3; i < 6; i++) {
        des_right_wrench[i] = 0.0;
        des_left_wrench[i]  = 0.0;
    }
```

and for the slave side.

```
// Evaluate desired velocity
    for (int i = 0; i < 6; i++) {
        des_right_velocity[i] = meas_right_master_velocity[i];
        des_left_velocity[i]  = meas_left_master_velocity[i];
    }
```

The force set-points are then stored into `geometry_msgs/WrenchStamped` message, the velocity set-point are stored into `geometry_msgs/TwistStamped` message and all the messages are published:

```
//Send desired set-points
    out_des_right_wrench.header.stamp     = ros::Time::now();
    out_des_right_wrench.wrench.force.x  = des_right_wrench[0];
    out_des_right_wrench.wrench.force.y  = des_right_wrench[1];
    out_des_right_wrench.wrench.force.z  = des_right_wrench[2];
    out_des_right_wrench.wrench.torque.x = des_right_wrench[3];
    out_des_right_wrench.wrench.torque.y = des_right_wrench[4];
    out_des_right_wrench.wrench.torque.z = des_right_wrench[5];

    out_des_left_wrench.header.stamp     = ros::Time::now();
    out_des_left_wrench.wrench.force.x  = des_left_wrench[0];
    out_des_left_wrench.wrench.force.y  = des_left_wrench[1];
    out_des_left_wrench.wrench.force.z  = des_left_wrench[2];
    out_des_left_wrench.wrench.torque.x = des_left_wrench[3];
    out_des_left_wrench.wrench.torque.y = des_left_wrench[4];
    out_des_left_wrench.wrench.torque.z = des_left_wrench[5];

    des_right_wrench_pub.publish(out_des_right_wrench);
    des_left_wrench_pub.publish(out_des_left_wrench);

    out_des_right_velocity.header.stamp    = ros::Time::now();
    out_des_right_velocity.twist.linear.x  =
        des_right_velocity[0] * scaleFactor;
    out_des_right_velocity.twist.linear.y  =
        des_right_velocity[1] * scaleFactor;
    out_des_right_velocity.twist.linear.z  =
        des_right_velocity[2] * scaleFactor;
    out_des_right_velocity.twist.angular.x =
        des_right_velocity[3] * scaleFactor;
    out_des_right_velocity.twist.angular.y =
        des_right_velocity[4] * scaleFactor;
    out_des_right_velocity.twist.angular.z =
        des_right_velocity[5] * scaleFactor;

    out_des_left_velocity.header.stamp    = ros::Time::now();
    out_des_left_velocity.twist.linear.x  =
        des_left_velocity[0] * scaleFactor;
    out_des_left_velocity.twist.linear.y  =
        des_left_velocity[1] * scaleFactor;
    out_des_left_velocity.twist.linear.z  =
        des_left_velocity[2] * scaleFactor;
    out_des_left_velocity.twist.angular.x =
        des_left_velocity[3] * scaleFactor;
    out_des_left_velocity.twist.angular.y =
        des_left_velocity[4] * scaleFactor;
    out_des_left_velocity.twist.angular.z =
        des_left_velocity[5] * scaleFactor;

    des_right_velocity_pub.publish(out_des_right_velocity);
    des_left_velocity_pub.publish(out_des_left_velocity);
```

For the slave set-points a scaling factor is added in order to fit with the master workspace.

5.5 velocity_integrator node

First, the velocity integrator waits at least 10 seconds for the initial slave pose. Then the main loop can start, otherwise not.

```
while((meas_init_pose_new_data_flag==false)&&
      (t_new-t_old<10.0)){ros::spinOnce(); t_new=ros::Time::now().toSec();}

        if(meas_init_pose_new_data_flag==true){ //Integrator can start
                std::cout<<"Integrator:␣waiting␣start␣pose␣from␣robot
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣succeed"<<std::endl;

                in_start_pose_sub.shutdown();//Stop receiving initial pose

                while ( public_node_handler.ok()){      //Main loop
            ...
            }
        }
        else{   //Integrator can't start
                std::cout<<"Integrator:␣waiting␣start␣pose␣from␣robot
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣failed␣after␣10␣second␣waiting"<<std::endl;
        }
```

The pose of the slave robot is acquired using the following callback function:

```
void in_start_pose_callback(
    const geometry_msgs::PoseStamped::ConstPtr& msg){

        ROS_INFO_STREAM("in_start_pose_callback");

        pose_kdl.p[0]=msg->pose.position.x;
        pose_kdl.p[1]=msg->pose.position.y;
        pose_kdl.p[2]=msg->pose.position.z;

        pose_kdl.M=KDL::Rotation::Quaternion(
            msg->pose.orientation.x, msg->pose.orientation.y,
            msg->pose.orientation.z, msg->pose.orientation.w);

        meas_init_pose_new_data_flag=true;
}
```

Similarly the desired velocity to be applied to the slave robot are acquired.

```
void in_twist_callback(const geometry_msgs::TwistStamped::ConstPtr& msg){

        // ROS_INFO_STREAM("in_twist_callback");

        meas_twist[0]=msg->twist.linear.x;
        meas_twist[1]=msg->twist.linear.y;
        meas_twist[2]=msg->twist.linear.z;
        meas_twist[3]=msg->twist.angular.x;
        meas_twist[4]=msg->twist.angular.y;
        meas_twist[5]=msg->twist.angular.z;
}
```

In the main loop we first compute the slave pose set-point by integrating the desired velocity, starting from the initial pose of the robot.

```
t_new=ros::Time::now().toSec();
pose_kdl = addDelta(pose_kdl, meas_twist, t_new-t_old);
t_old=t_new;
```

The pose is then stored into a geometry_msgs/PoseStamped message and then published.

```
out_pose_msg.header.stamp=ros::Time::now();

out_pose_msg.header.frame_id = published_tf_pose_reference_name;

out_pose_msg.pose.position.x=pose_kdl.p[0];
out_pose_msg.pose.position.y=pose_kdl.p[1];
out_pose_msg.pose.position.z=pose_kdl.p[2];

pose_kdl.M.GetQuaternion(out_pose_msg.pose.orientation.x,
    out_pose_msg.pose.orientation.y, out_pose_msg.pose.orientation.z,
    out_pose_msg.pose.orientation.w);

out_pose.publish(out_pose_msg);
```

The same pose is also published as a `tf_transform` as a debugging information.

```
out_pose_transform.setOrigin(
    tf::Vector3( out_pose_msg.pose.position.x,
    out_pose_msg.pose.position.y, out_pose_msg.pose.position.z ) );

out_pose_transform.setRotation( tf::Quaternion(
        out_pose_msg.pose.orientation.x, out_pose_msg.pose.orientation.y,
        out_pose_msg.pose.orientation.z, out_pose_msg.pose.orientation.w ));

out_pose_broadcaster.sendTransform( tf::StampedTransform(
    out_pose_transform, ros::Time::now(),
    published_tf_pose_reference_name, published_tf_pose_name ) );
```

6 Collision detection

The goal is to compute the distance between the tools and the forbidden regions in order to give the user useful information about the operative scenario. The concepts behind the implementation of the collision detection node are explained in the ”Colliding region identification for multi-master/multi-slave platform” section of the deliverable D3.1 – Multi-modal human-robot interfaces and architecture for the MULTIROBOTS-SURGERY platform. As a brief overview, the algorithm for collision detection receives in input:

- the poses of the tools,
- the positions of the trocars, and
- the vertices of the parallelepipeds of the forbidden regions.

This node computes the distances between the tools using a capsule-to-capsule distance computation function and between the tools and the pelvic bones using a capsule-to-paralellelepiped distance computation function. For each tool, only the minimum distance is kept; at the end of each cycle, these distances will be presented as output.

6.1 Collision detection ROS implementation

The collision detection algorithm is implemented in a ROS node using C++, ROS and Eigen libraries and data types.

The node starts fetching the transformation between the world reference frame and the robots reference frames. Since the poses of the robots refer to their base reference frames, a transformation is needed for computing the distances in a common reference frame.

```
t_bones = tfBuffer.lookupTransform("world", "bones_marker_frame",
ros::Time(0), ros::Duration(3000.0));

tr_sbs_link_l = tfBuffer.lookupTransform("world", "mesa_left/sbs_world",
ros::Time(0), ros::Duration(3.0));

tr_sbs_link_r = tfBuffer.lookupTransform("world", "mesa_right/sbs_world",
ros::Time(0), ros::Duration(3.0));

transform_l = tfBuffer.lookupTransform("world", "mesa_left/rcm",
ros::Time(0), ros::Duration(3.0));

transform_r = tfBuffer.lookupTransform("world", "mesa_right/rcm",
ros::Time(0), ros::Duration(3.0));

transform_psm1 = tfBuffer.lookupTransform("world", "PSM1",
ros::Time(0), ros::Duration(3.0));

transform_psm2 = tfBuffer.lookupTransform("world", "PSM2",
ros::Time(0), ros::Duration(3.0));
```

The information about orientation and translation of the transforms are then stored into Eigen matrixes.

```
Eigen::Quaternion<double> q_temp=
Eigen::Quaternion<double>(tr_sbs_link_l.transform.rotation.w,
tr_sbs_link_l.transform.rotation.x,tr_sbs_link_l.transform.rotation.y,
tr_sbs_link_l.transform.rotation.z);

Eigen::Matrix3d rot = q_temp.toRotationMatrix();
sbs_l <<rot(0, 0),rot(0, 1),rot(0, 2),tr_sbs_link_l.transform.translation.x,
rot(1, 0), rot(1, 1), rot(1, 2), tr_sbs_link_l.transform.translation.y,
rot(2, 0), rot(2, 1), rot(2, 2), tr_sbs_link_l.transform.translation.z,
0, 0, 0, 1;

q_temp = Eigen::Quaternion<double>(tr_sbs_link_r.transform.rotation.w,
tr_sbs_link_r.transform.rotation.x, tr_sbs_link_r.transform.rotation.y,
tr_sbs_link_r.transform.rotation.z);
rot = q_temp.toRotationMatrix();

sbs_r <<rot(0, 0),rot(0, 1),rot(0, 2),tr_sbs_link_r.transform.translation.x,
rot(1, 0), rot(1, 1), rot(1, 2), tr_sbs_link_r.transform.translation.y,
rot(2, 0), rot(2, 1), rot(2, 2), tr_sbs_link_r.transform.translation.z,
0, 0, 0, 1;

q_temp = Eigen::Quaternion<double>(transform_psm1.transform.rotation.w,
transform_psm1.transform.rotation.x, transform_psm1.transform.rotation.y,
transform_psm1.transform.rotation.z);

rot = q_temp.toRotationMatrix();
m_psm1<<rot(0,0),rot(0,1),rot(0,2),transform_psm1.transform.translation.x,
rot(1, 0), rot(1, 1), rot(1, 2), transform_psm1.transform.translation.y,
rot(2, 0), rot(2, 1), rot(2, 2), transform_psm1.transform.translation.z,
0, 0, 0, 1;

q_temp = Eigen::Quaternion<double>(transform_psm2.transform.rotation.w,
transform_psm2.transform.rotation.x, transform_psm2.transform.rotation.y,
transform_psm2.transform.rotation.z);

rot = q_temp.toRotationMatrix();
m_psm2<<rot(0,0),rot(0,1),rot(0,2),transform_psm2.transform.translation.x,
rot(1, 0), rot(1, 1), rot(1, 2), transform_psm2.transform.translation.y,
rot(2, 0), rot(2, 1), rot(2, 2), transform_psm2.transform.translation.z,
0, 0, 0, 1;

q_temp = Eigen::Quaternion<double>(t_bones.transform.rotation.w,
t_bones.transform.rotation.x, t_bones.transform.rotation.y,
t_bones.transform.rotation.z);
rot = q_temp.toRotationMatrix();
bones << rot(0, 0), rot(0, 1), rot(0, 2), t_bones.transform.translation.x
+ 0.02,rot(1, 0), rot(1, 1), rot(1, 2), t_bones.transform.translation.y,
rot(2, 0), rot(2, 1), rot(2, 2), t_bones.transform.translation.z,
0, 0, 0, 1;
```

From the transformations we obtain the coordinates of the trocars (which are stored into four Eigen::Vector3d.

```
trocar_ml[0] = transform_l.transform.translation.x;
trocar_ml[1] = transform_l.transform.translation.y;
trocar_ml[2] = transform_l.transform.translation.z;
```

```

trocar_mr[0] = transform_r.transform.translation.x;
trocar_mr[1] = transform_r.transform.translation.y;
trocar_mr[2] = transform_r.transform.translation.z;

trocar_psm1[0] = transform_psm1.transform.translation.x;
trocar_psm1[1] = transform_psm1.transform.translation.y;
trocar_psm1[2] = transform_psm1.transform.translation.z;

trocar_psm2[0] = transform_psm2.transform.translation.x;
trocar_psm2[1] = transform_psm2.transform.translation.y;
trocar_psm2[2] = transform_psm2.transform.translation.z;
```

Subsequently, the node collects information about the forbidden regions by reading a file containing the number NUM of parallelepipeds used for modeling the pelvic bones and the coordinates of their vertices, one vertex per line, listed with the convention in Figure 11.

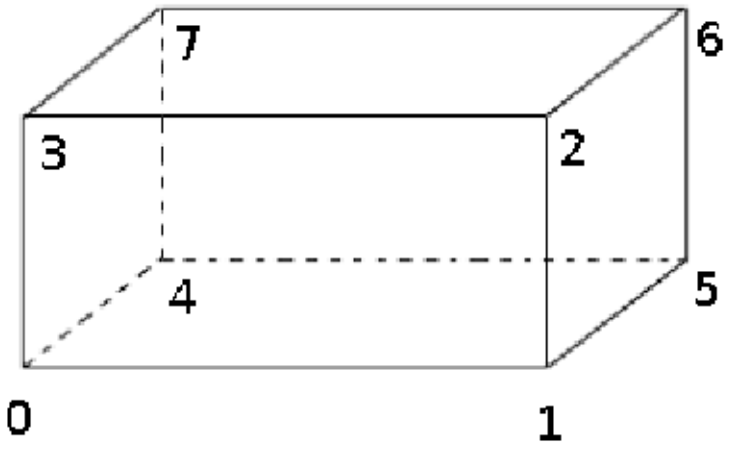


Figure 11: Naming convention for parallelepipeds vertices

```
FILE *f = fopen("vertices.txt", "r");
fscanf(f, "%d", &NUM);
boxes = Eigen::MatrixXd(NUM, 24);
Eigen::Matrix4d init_trans;
init_trans << 0.0, -1.0, 0.0, 0.0,
0.0, 0.0, 1.0, 0.0,
-1.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 1.0;
Eigen::Vector4d support_tmp;
for (int i = 0; i < NUM; i++)
{
        Eigen::Vector3d tmp;
        for (int c = 0; c < 8; c++)
        {
        fscanf(f, "%lf␣%lf␣%lf", &tmp[0], &tmp[1], &tmp[2]);
        support_tmp[0] = tmp[0] * SCALING;
        support_tmp[1] = tmp[1] * SCALING;
        support_tmp[2] = tmp[2] * SCALING;
        support_tmp[3] = 1.0;
        support_tmp = bones.inverse() * support_tmp;
        support_tmp = (init_trans.inverse() * support_tmp);
        boxes(i, c * 3) = (support_tmp[0]);
        boxes(i, c * 3 + 1) = (support_tmp[1]);
        boxes(i, c * 3 + 2) = (support_tmp[2]);
                }
        }
        fclose(f);
```

The vertices are stored in a $NUM * 24$ Eigen matrix, where each row represents a parallelepiped and the columns are the concatenation of coordinates (x, y, z) of each vertex of the parallelepiped.

After that, the node publishes in ROS a topic called `/collision/distances` with the result of the computation:

```
pub_dist=
nh.advertise<std_msgs::Float64MultiArray>("/collision/distances",100);
```

where

```
ros::Publisher pub_dist;
```

which is defined as a global variable. Since the node computes four distances (one for each tool), the `std_msgs::Float64MultiArray` data type is chosen as the data type for the topic. This will be filled in a pre-determined order: the minimum distance for the left SARAS tool; the minimum distance for the right SARAS tool; the minimum distance for the first daVinci tool; the minimum distance for the second daVinci tool.

In order to acquire the tool poses, the node subscribes to the following topics:

```
ros::Subscriber sub_ml = nh.subscribe("/mesa_left/ee_pose", 100,callback1);

ros::Subscriber sub_mr = nh.subscribe("/mesa_right/ee_pose", 100,callback2);

ros::Subscriber sub_psm1 =
nh.subscribe("/dvrk/PSM1/position_cartesian_current", 100, callback3);

ros::Subscriber sub_psm2 =
nh.subscribe("/dvrk/PSM2/position_cartesian_current", 100, callback4);
```

These topics are published by the other ROS nodes in the system. Each subscriber registers a different callback to manage the arrival of new data; these, however, have the same function: they update the pose of the tool in order to obtain the current scenario. The following snippet shows the structure of one of the callbacks (which is representative also for the other callbacks).

```
void callback1(const geometry_msgs::PoseStamped::ConstPtr &msg)
{
        ml_ee[0] = msg->pose.position.x;
        ml_ee[1] = msg->pose.position.y;
        ml_ee[2] = msg->pose.position.z;
        Eigen::Vector4d temp;
        temp << ml_ee[0], ml_ee[1], ml_ee[2], 1;
        temp = sbs_l * temp;
        ml_ee << temp[0], temp[1], temp[2];
}
```

The callback simply reads the current Cartesian position of the tool tip, stores it in a vector and performs the transformation into the common reference frame. Reading only the position (and not the orientation) of the tool-tip is enough because for modeling the tool as a capsule only the coordinates of the end-points (which are the tool tip and the trocar) are required.

Finally the node cycles periodically throwing the callbacks that are waiting in the queue and then calling the distance computation function.

```
while (ros::ok())
{
    ros::getGlobalCallbackQueue()->callAvailable(ros::WallDuration(0.05));
    compute_distances();
}
```

With this architecture the node updates all the poses and then computes the minimum distance of each tool on the current scenario. The cycling period has been chosen in order to have an updating frequency for the distance computations compatible with the dynamics of the operation while keeping its impact on the overall system as light as possible.

The function `compute_distances` is defined as follows:

```
void compute_distances();
```

It does not take any input parameter because all the variables needed for the computations are defined as global and also it does not have any return type because it publishes the results directly on the topic before returning the control to the main function of the node. The function checks the distance between each couple of tools using the *dist_capsule_to_capsule* function and if the new distance is lower than the distances already saved for the tools taken in exam it updates those values.

```
double distance_tl = 10000, distance_tr = 10000,
distance_psm1 = 10000, distance_psm2 = 10000;
double dist_temp1, dist_temp2, dist_temp3,
dist_temp4, dist_temp5, dist_temp6;
int ret, ret_tl, ret_tr, ret_psm1, ret_psm2;
Eigen::Vector3d point_tl, point_tr, point_psm1, point_psm2;
Eigen::Vector3d temp1, temp2;

ret = dist_capsule_to_capsule(ml_ee, trocar_ml, mr_ee, trocar_mr,
r_saras_tool, r_saras_tool, dist_temp1, temp1, temp2);
distance_tl = dist_temp1;
distance_tr = dist_temp1;
point_tl = temp1;
point_tr = temp2;
ret_tl = ret;
ret_tr = ret;

ret=dist_capsule_to_capsule(PSM1_ee,trocar_psm1,PSM2_ee,trocar_psm2,
r_davinci_tool, r_davinci_tool, dist_temp2, temp1, temp2);
distance_psm1 = dist_temp2;
distance_psm2 = dist_temp2;
point_psm1 = temp1;
point_psm2 = temp2;
ret_psm1 = ret;
ret_psm2 = ret;
```

```

ret = dist_capsule_to_capsule(mr_ee,trocar_mr,PSM1_ee, trocar_psm1,
r_saras_tool, r_davinci_tool, dist_temp3, temp1, temp2);
if (dist_temp3 < distance_tr)
{
        distance_tr = dist_temp3;
        point_tr = temp1;
        ret_tr = ret;
}
if (dist_temp3 < distance_psm1)
{
        distance_psm1 = dist_temp3;
        point_psm1 = temp2;
        ret_psm1 = ret;
}

ret = dist_capsule_to_capsule(mr_ee,trocar_mr,PSM2_ee, trocar_psm2,
r_saras_tool, r_davinci_tool, dist_temp4, temp1, temp2);

if (dist_temp4 < distance_tr)
{
        distance_tr = dist_temp4;
        point_tr = temp1;
        ret_tr = ret;
}
if (dist_temp4 < distance_psm2)
{
        distance_psm2 = dist_temp4;
        point_psm2 = temp2;
        ret_psm2 = ret;
}

ret = dist_capsule_to_capsule(ml_ee,trocar_ml,PSM1_ee, trocar_psm1,
r_saras_tool, r_davinci_tool, dist_temp5, temp1, temp2);

if (dist_temp5 < distance_tl)
{
        distance_tl = dist_temp5;
        point_tl = temp1;
        ret_tl = ret;
}
if (dist_temp5 < distance_psm1)
{
        distance_psm1 = dist_temp5;
        point_psm1 = temp2;
        ret_psm1 = ret;
}

ret = dist_capsule_to_capsule(ml_ee,trocar_ml,PSM2_ee, trocar_psm2,
r_saras_tool, r_davinci_tool, dist_temp6, temp1, temp2);

if (dist_temp6 < distance_tl)
{
        distance_tl = dist_temp6;
        point_tl = temp1;
        ret_tl = ret;
}
if (dist_temp6 < distance_psm2)
{
        distance_psm2 = dist_temp6;
        point_psm2 = temp2;
        ret_psm2 = ret;
}
```

After that, the function cycles for each parallelepiped of the forbidden region computing the distance between each tool and the parallelepiped taken in exam calling the `dist_capsule_to_parallelepiped` function and again updating the distance values if a new lower value is computed:

```
for (int i = 0; i < NUM; i++)
{
        Eigen::VectorXd tmp(24);
        tmp << boxes(i, 0), boxes(i, 1), boxes(i, 2),
        boxes(i, 3),boxes(i, 4), boxes(i, 5),
        boxes(i, 6),boxes(i, 7), boxes(i, 8),
        boxes(i, 9),boxes(i, 10), boxes(i, 11),
        boxes(i, 12),boxes(i, 13), boxes(i, 14),
        boxes(i, 15),boxes(i, 16), boxes(i, 17),
        boxes(i, 18),boxes(i, 19), boxes(i, 20),
        boxes(i, 21),boxes(i, 22), boxes(i, 23);
        Eigen::Vector3d vertices[8];
        vertices[0] << tmp[0], tmp[1], tmp[2];
        vertices[1] << tmp[3], tmp[4], tmp[5];
        vertices[2] << tmp[6], tmp[7], tmp[8];
        vertices[3] << tmp[9], tmp[10], tmp[11];
        vertices[4] << tmp[12], tmp[13], tmp[14];
        vertices[5] << tmp[15], tmp[16], tmp[17];
        vertices[6] << tmp[18], tmp[19], tmp[20];
        vertices[7] << tmp[21], tmp[22], tmp[23];

        ret = dist_capsule_to_parallelepiped(ml_ee, trocar_ml, vertices,
        r_saras_tool,dist_temp1, temp1, temp2);

        if (dist_temp1 < distance_tl)
        {
                distance_tl = dist_temp1;
                point_tl = temp1;
                ret_tl = ret;
        }

        ret = dist_capsule_to_parallelepiped(mr_ee, trocar_mr, vertices,
        r_saras_tool, dist_temp2, temp1, temp2);

        if (dist_temp2 < distance_tr)
        {
                distance_tr = dist_temp1;
                point_tr = temp1;
                ret_tr = ret;
        }
        ret = dist_capsule_to_parallelepiped(PSM1_ee, trocar_psm1, vertices,
        r_davinci_tool, dist_temp3, temp1, temp2);

        if (dist_temp3 < distance_psm1)
        {
                distance_psm1 = dist_temp3;
                point_psm1 = temp1;
                ret_psm1 = ret;
        }
        ret = dist_capsule_to_parallelepiped(PSM2_ee, trocar_psm2, vertices,
        r_davinci_tool, dist_temp4, temp1, temp2);

        if (dist_temp4 < distance_psm2)
        {
                distance_psm2 = dist_temp4;
                point_psm2 = temp1;
                ret_psm2 = ret;
        }
}
```

Finally the function publishes the minimum distances following the order already mentioned.

```
std_msgs::Float64MultiArray msg;
msg.data.resize(4);
msg.data[0] = distance_tl;
msg.data[1] = distance_tr;
msg.data[2] = distance_psm1;
msg.data[3] = distance_psm2;
pub_dist.publish(msg);
```

The function `dist_capsule_to_capsule` is defined as follows:

```
int dist_capsule_to_capsule(
    const Eigen::Vector3d& P1, const Eigen::Vector3d& P2,
    const Eigen::Vector3d& P3, const Eigen::Vector3d& P4,
    double r1, double r2,
    double& distance,Eigen::Vector3d& point_on_capsule1,
    Eigen::Vector3d& point_on_capsule2);
```

The function takes the coordinates of the two end-points of the first capsule, the coordinates of the two end-points of the second capsule, the radius of the first capsule, the radius of the second capsule as proper inputs, while uses three variables passed by reference as inputs to store the results of computation: (1) the distance between the capsules, (2) the closest point on the first capsule, and (3) the closest point on the second capsule. The function returns an integer value that can have these values:

- **-1**: if the two capsules intersect; in this case the distance is computed and is negative (instead of zero, to give the user an idea of how much the capsules are intersecting) while the closest points are both set to $(0, 0, 0)$ by default;
- **0**: if the two capsules are tangent; the distance is zero and the closest points are coincident and computed properly;
- **1**: if the two capsules do not intersect; the distance and the closest points are computed properly;
- **2**: if the two capsules are parallel; the distance is computed properly while the closest point are both set to $(0, 0, 0)$ by default;

```
int dist_capsule_to_capsule(
    const Eigen::Vector3d& P1, const Eigen::Vector3d& P2,
    const Eigen::Vector3d& P3, const Eigen::Vector3d& P4,
    double r1,double r2,
    double& distance, Eigen::Vector3d& point_on_capsule1,
    Eigen::Vector3d& point_on_capsule2)
{
    point_on_capsule1<<0,0,0;
    point_on_capsule2<<0,0,0;
    Eigen::Vector3d di1  = P2 - P1;
    Eigen::Vector3d di2  = P4 - P3;
    Eigen::Vector3d di12 = P3 - P1;
    double D1  = di1.dot(di1);
    double D2  = di2.dot(di2);
    double S1  = di1.dot(di12);
    double S2  = di2.dot(di12);
    double R   = di1.dot(di2);
    double D = D1*D2-R*R;
    double u,t,u_t;
    bool parallel=false;
    if ((D1<SMALL_NUM && D1>-SMALL_NUM) || (D2<SMALL_NUM && D2>-SMALL_NUM))
    {
        if (D1 >SMALL_NUM || D1<-SMALL_NUM)
        {
            u =0;
            t =S1/D1;
            if(t<SMALL_NUM)
               t=0;
            else if (t>1)
                t=1;
        }
    else if (D2 >SMALL_NUM || D2<-SMALL_NUM )
    {
        t = 0;
        u = -S2/D2;

        if (u<SMALL_NUM)
            u=0;
        else if(u>1)
            u=1;
    }
    else
    { using
        t = 0;
        u = 0;
        parallel=true;
    }
}
else if (D <SMALL_NUM && D>-SMALL_NUM)
{
    t = 0;
    u = -S2/D2;
     if (u<SMALL_NUM)
         u_t=0;
     else if(u>1)
         u_t=1;
     else
        u_t=u;


    if (std::abs(u_t - u)>SMALL_NUM )
    {    t = (u_t*R+S1)/D1;
         if (t<SMALL_NUM)
            t=0;
        else if(t>1)
            t=1;

        u = u_t;
   }
}
else
  {

    t = (S1*D2-S2*R)/D;

    if (t<SMALL_NUM)
        t=0;
    else if(t>1)
        t=1;

    u = (t*R-S2)/D2;
    if (u<SMALL_NUM)
         u_t=0;

     else if(u>1)
         u_t=1;
     else
        u_t=u;
    if (std::abs(u_t - u)>SMALL_NUM )
    {
        t = (u_t*R+S1)/D1;
        if (t<SMALL_NUM )
            t=0;
        else if(t>1)
            t=1;

        u = u_t;
    }
}

double dist_temp = (di1*t-di2*u-di12).norm();
distance=dist_temp-r1-r2;
Eigen::Vector3d point_on_ax1,point_on_ax2;
if(distance<-SMALL_NUM)
    return -1;
else

{
    if(!parallel)
    {
        point_on_ax1=P1 + di1*t;
        point_on_ax2=P3+di2*u;
        Eigen::Vector3d dir_1_to_2=(point_on_ax2-point_on_ax1).normalized();
        point_on_capsule1=point_on_ax1+r1*dir_1_to_2;
        point_on_capsule2=point_on_ax2-r2*dir_1_to_2;
        if((distance<SMALL_NUM && distance >-SMALL_NUM))
        {
           distance=0;
           return 0;
        }
        else
        {
            return 1;
        }
        }
        else
        {
            return 2;
        }

    }
}
```

As explained in details in the deliverable D3.1 – Multi-modal human-robot interfaces and architecture for the MULTIROBOTS-SURGERY platform, the `dist_capsule_to_capsule` function implements an algorithm for segment-to-segment distance computation by:

- parameterizing the lines containing the segments,
- solving a minimization problem and, finally,
- extending the computation to the capsule-to-capsule case by subtracting to the distance the radii of the capsules.

The closest points are then computed by projecting the closest points returned by the segment-to-segment distance algorithm (lying on the segment by definition) on the capsule surface. Mathematically the problem is:

$$\overline{point\_on\_capsule1} = \overline{point\_on\_segment1} + r1 * \overline{v1}$$
$$\overline{point\_on\_capsule2} = \overline{point\_on\_segment2} + r2 * \overline{v2}$$

where

$$\overline{v1} = \frac{\overline{point\_on\_segment2} - \overline{point\_on\_segment1}}{normusing(\overline{point\_on\_segment2} - \overline{point\_on\_segment1})}$$
$$\overline{v2} = \frac{\overline{point\_on\_segment1} - \overline{point\_on\_segment2}}{norm(\overline{point\_on\_segment1} - \overline{point\_on\_segment2})}$$

The `dist_capsule_to_paralellepipeds` function is defined as follows:

```
int dist_capsule_to_parallelepiped(const Eigen::Vector3d& p1,
const Eigen::Vector3d& p2,const Eigen::Vector3d vertices[],
double r,double& distance,Eigen::Vector3d& point_on_capsule,
Eigen::Vector3d& point_on_parall)
```

The function takes the coordinates of the two end-points of the capsule, the coordinates of the vertices of the parallelepiped, the radius of the capsule as proper inputs, while uses three variables passed by reference as inputs to store the results of computation in: (1) the distance between the capsule and the parallelepiped, (2) the closest point on the capsule, and (3) the closest point on the parallelepiped. The function returns an integer value that can have these values:

- **-1**: if the two objects intersects; in this case the distance is computed and is negative (to give the user an idea of how much the objects are intersecting). The closest points are both set to (0,0,0) by default;
- **0**: if the two objects are tangent; the distance is zero and the closest points are coincident and computed properly;
- **1**: if the two objects do not intersect; the distance and the closest points are computed properly.

```cpp
int dist_capsule_to_parallelepiped(const Eigen::Vector3d& p1,
const Eigen::Vector3d& p2,const Eigen::Vector3d vertices[],
double r,double& distance,Eigen::Vector3d& point_on_capsule,
Eigen::Vector3d& point_on_parall)
{
    double d1,d2,d;
    Eigen::Vector3d point_c,point_p,point_c1,point_p1,point_c2,point_p2;
    point_on_capsule<<0,0,0;
    point_on_parall<<0,0,0;
    int res1,res2;
    res1=dist_sphere_to_parallelepiped(p1,vertices,r,d1,point_c1,point_p1);
    if(res1==-1)
    {
        distance=d1;
        return -1;
    }
    res2=dist_sphere_to_parallelepiped(p2,vertices,r,d2,point_c2,point_p2);
    if(res2==-1)
    {
        distance=d2;
        return -1;
    }
    if(d2<d1)
    {
        d=d2;
        point_c=point_c2;
        point_p=point_p2;
    }using
    else
    {
        d=d1;
        point_c=point_c1;
        point_p=point_p1;
    }
    Eigen::Matrix<double,12,6> lines;
    lines<< vertices[0][0],vertices[0][1],vertices[0][2],
    vertices[1][0],vertices[1][1],vertices[1][2],
    vertices[1][0],vertices[1][1],vertices[1][2],
    vertices[2][0],vertices[2][1],vertices[2][2],
    vertices[2][0],vertices[2][1],vertices[2][2],
    vertices[3][0],vertices[3][1],vertices[3][2],
    vertices[3][0],vertices[3][1],vertices[3][2],
    vertices[0][0],vertices[0][1],vertices[0][2],
    vertices[0][0],vertices[0][1],vertices[0][2],
    vertices[4][0],vertices[4][1],vertices[4][2],
    vertices[4][0],vertices[4][1],vertices[4][2],
    vertices[7][0],vertices[7][1],vertices[7][2],
    vertices[7][0],vertices[7][1],vertices[7][2],
    vertices[3][0],vertices[3][1],vertices[3][2],
    vertices[4][0],vertices[4][1],vertices[4][2],
    vertices[5][0],vertices[5][1],vertices[5][2],
    vertices[5][0],vertices[5][1],vertices[5][2],
    vertices[6][0],vertices[6][1],vertices[6][2],
    vertices[6][0],vertices[6][1],vertices[6][2],
    vertices[7][0],vertices[7][1],vertices[7][2],
    vertices[1][0],vertices[1][1],vertices[1][2],
    vertices[5][0],vertices[5][1],vertices[5][2],
    vertices[2][0],vertices[2][1],vertices[2][2],
    vertices[6][0],vertices[6][1],vertices[6][2];

    for (int i=0;i<12;i++)
    {
        double d_c;
        Eigen::Vector3d cap1,cap2;
        int res=dist_capsule_to_capsule(p1,p2,
        Eigen::Vector3d(lines(i,0),lines(i,1),lines(i,2)),
        Eigen::Vector3d(lines(i,3),lines(i,4),lines(i,5)),
        r,0,d_c,cap1,cap2);
    if(res ==-1)
    {
        distance=d_c;
        point_on_capsule<<0,0,0;
        point_on_parall<<0,0,0;
        return -1;
    }

    if(d_c<d)
     {
        d=d_c;
        point_c=cap1;
        point_p=cap2;
    }
    }
    distance=d;
    point_on_capsule=point_c;
    point_on_parall=point_p;

    if(distance>SMALL_NUM)
     return 1;
    else
        return 0;


}
```

The function checks the distance between the capsule and all the faces and the edges of the parallelepiped; the distance between the capsule and the faces is computed calling the function `dist_sphere_to_parallelepiped` on both the end-point while the distance between the capsule and the edges is computed calling the `dist_capsule_to_capsule` function on each edge, considering an edge as a capsule with the radius equal to 0.

The `dist_sphere_to_parallelepiped` function is defined as follows:

```
int dist_sphere_to_parallelepiped(const Eigen::Vector3d& s,
const Eigen::Vector3d vertices[],double r,double& distance,
Eigen::Vector3d& point_on_sphere,Eigen::Vector3d& point_on_parall);
```

The function takes the coordinates of the center of the sphere, the coordinates of the vertices of the parallelepiped, the radius of the sphere as proper inputs, while uses three variables passed by reference as inputs to store the results of computation in the following order: (1) the distance between the sphere and the parallelepiped, (2) the closest point on the sphere, and (3) the closest point on the parallelepiped. The function returns an integer value with the following meanings:

- **-1**: if the two objects intersects; in this case the distance is computed and is negative (to give the user an idea of how much the objects are intersecting).The closest points are both set to $(0, 0, 0)$ by default;
- **0**: if the two objects are tangent; the distance is zero, the closest points coincide and are computed properly;
- **1**: if the two objects do not intersect; the distance and the closest points are computed properly.

```
int dist_sphere_to_parallelepiped(const Eigen::Vector3d& s,
const Eigen::Vector3d vertices[],double r,
double& distance,Eigen::Vector3d& point_on_sphere,
Eigen::Vector3d& point_on_parall)
{
    point_on_sphere<<0,0,0;
    point_on_parall<<0,0,0;
    Eigen::Vector3d origin,x,y,z;
    double roll,pitch,yaw;

    get_parall_frame(vertices,origin,x,y,z);
    vers2rpy(x,y,z,roll,pitch,yaw);
    Eigen::Matrix3d R=(Eigen::AngleAxisd(roll,Eigen::Vector3d::UnitX())
    *Eigen::AngleAxisd(pitch,Eigen::Vector3d::UnitY())
    *Eigen::AngleAxisd(yaw,Eigen::Vector3d::UnitZ())).toRotationMatrix();
    Eigen::Matrix3d inv=R.inverse();
    Eigen::Vector3d s_t=inv*s-origin;
    Eigen::Vector3d vertices_temp[8];
    for (int i=0;i<8;i++)
    {
        vertices_temp[i]=inv*vertices[i] -origin;
    }
    double max_x,max_y,max_z;
    max_x=vertices_temp[1][0];
    max_y=vertices_temp[4][1];
    max_z=vertices_temp[3][2];

    double dx,dy,dz;
    Eigen::Vector3d temp;
    temp<<-s_t[0],0,s_t[0]-max_x;
    dx=temp[0];
    if(temp[1]>dx)
        dx=temp[1];
    if(temp[2]>dx)
        dx=temp[2];
    temp<<-s_t[1],0,s_t[1]-max_y;
    dy=temp[0];
    if(temp[1]>dy)
        dy=temp[1];
    if(temp[2]>dy)
        dx=temp[2];
    temp<<-s_t[2],0,s_t[2]-max_z;
      dz=temp[0];
    if(temp[1]>dz)
        dy=temp[1];
    if(temp[2]>dz)
        dx=temp[2];
    distance=Eigen::Vector3d(dx,dy,dz).norm()- r;
    if(distance < SMALL_NUM)
        return -1;

    if(s_t[0]>SMALL_NUM)
        dx=-dx;
    if(s_t[1]>SMALL_NUM)
        dy=-dy;
    if(s_t[2]>SMALL_NUM)
        dz=-dz;
    Eigen::Vector3d direz= Eigen::Vector3d(dx,dy,dz).normalized();
    point_on_sphere=s_t+r*direz;


 point_on_parall=point_on_sphere+distance*direz;
 point_on_sphere=R*point_on_sphere+origin;
 point_on_parall=R*point_on_parall+origin;
  if(distance>SMALL_NUM)
    return 1;

    return 0;

}
```

This function computes the distance between an end-point of the axis of the capsule and the faces of the parallelepipeds modeling the point as a sphere, since it is the center of the hemispherical termination of the capsule. Given the chosen naming convention for the vertices of the parallelepipeds, it is easy to align both the sphere and the parallelepiped to a conventional reference frame. This allows to relate the position of the end-point to the parallelepipeds which leads to improved computation efficiency. These closest points, when unique, are computed as follows:

$$\overline{point\_on\_sphere} = \overline{S} + r * \overline{dir};$$

$$\overline{point\_on\_parall} = \overline{point\_on\_sphere} + distance * \overline{dir};$$

where $\overline{S}$ is the center of the sphere,*distance* is the output distance and $\overline{dir}$ is an unit vector whose components are the normalized distance components which are computed in the algorithm. Since both `dist_capsule_to_capsule` and `dist_sphere_to_parallelepiped` functions can compute the closest points on the surfaces, also `dist_capsule_to_parallelepiped` function can compute them.

7 Visual Collision Feedback

The system implements an augmented reality visual collision feedback to convey information regarding the collision events computed in the ROS node presented in Section 6. The basic concepts behind 3D information projection in the 2D plane is presented in the "Colliding Regions Displaying" section of deliverable D3.1 – Multi-modal human-robot interfaces and architecture for the MULTIROBOTS-SURGERY platform. This implementation follows the color-coding convention to represent the information regarding collision events as provided by the collision detection node.

7.1 Visualization Framework ROS Implementation

The described visual feedback functionality is provided by a ROS *nodelet*, which is a high-performance event-driven spin of the standard ROS node. It employs multi-threaded subscribers on image and tool pose topics and acquires the transforms between the latter and the camera frame evaluated in the calibration phase (Section 4). By using a nodelet, the time required for data transfer with the system is reduced thanks to a pre-defined zero-copy policy; consequently, the computation appears transparent to the user's perception.

The main nodelet class is defined as:

```cpp
class VisualFeedback : public nodelet::Nodelet
{
public:
  VisualFeedback();

  virtual ~VisualFeedback();

  ros::NodeHandle *nh;
  image_transport::ImageTransport *it;

private:
  virtual void onInit();

  void onLeftPoseReceived(const geometry_msgs::PoseStamped::ConstPtr &msg);
  void onLeftImageReceived(
    const sensor_msgs::Image::ConstPtr &msg);

  void onRightPoseReceived(
    const geometry_msgs::PoseStamped::ConstPtr &msg);
  void onRightImageReceived(
    const sensor_msgs::Image::ConstPtr &msg);

  void onInfoReceived(
    const std_msgs::Float64MultiArray::ConstPtr &msg);

  void processCameraInfo(
    const sensor_msgs::CameraInfo &cam_info, const int cam);
  // cam = 0 -> LEFT; cam = 1 -> RIGHT
  void drawLines(cv::Mat &image, const int cam);

private:
  // ros workers
  image_transport::Subscriber left_image_sub_;
  image_transport::Subscriber right_image_sub_;

  ros::Subscriber left_pose_sub_;
  ros::Subscriber right_pose_sub_;

  ros::Subscriber info_sub_;

  image_transport::Publisher left_image_pub_;
  image_transport::Publisher right_image_pub_;

  tf2_ros::Buffer tfBuffer;
  tf2_ros::TransformListener tfListener;

  // stored data
  sensor_msgs::CameraInfo left_cam_info_;
  sensor_msgs::CameraInfo right_cam_info_;

  cv_bridge::CvImagePtr left_capture_;
  cv_bridge::CvImagePtr right_capture_;

  cam_params cam_param_[2];
  std::vector<double> collison_distances;
  CvSize sz_[2];
  std::vector<std::vector<cv::Point3f>> points;

  cv::Scalar colors[2];

  // ros time
  ros::Time start_time;
  ros::Time current_time;
  ros::Timer left_pose_timer;
  ros::Timer right_pose_timer;

  // utilities
  std::mutex mut_pose;
  std::mutex mut_collision;
};
```

It declares the subscribers for left and right images, poses, for the collision distances and the transform listener for the tools to camera roto-translation matrix. It declares also two *mutex* for sharing variables between multiple threads. Finally, it declares the image publishers which transmit the left/right augmented reality images towards the virtual reality headset. Information is drawn on the images as sets of circles and lines using the namesake OpenCV drawing functions *circle* and *line* delimited by the projected tool points.

```
void VisualFeedback::drawLines(cv::Mat &image, const int cam)
{
    // point order has to be: [left_ee, right_ee, left_shaft, right_shaft]
    cv::Matx31d trasl;
    trasl(0, 0) = cam_param_[cam].P.col(3)(0, 0) / 1e3;
    trasl(1, 0) = cam_param_[cam].P.col(3)(1, 0) / 1e3;
    trasl(2, 0) = cam_param_[cam].P.col(3)(2, 0) / 1e3;
    cv::Matx31d rvec;
    rvec(0, 0) = 0.0;
    rvec(1, 0) = 0.0;
    rvec(2, 0) = 0.0;
    cv::Matx33d K = cam_param_[cam].P.get_minor<3, 3>(0, 0);
    // cv::Rodrigues(cv::Matx33d::eye(), rvecR);

    std::vector<cv::Point3f> left_in_points(2);
    std::vector<cv::Point3f> right_in_points(2);
    if (mut_pose.try_lock())
    {
        left_in_points = points.at(0);
        right_in_points = points.at(1);
        mut_pose.unlock();
    }
    std::vector<cv::Point2f> left_out_points;
    std::vector<cv::Point2f> right_out_points;
    cv::projectPoints(
        left_in_points, rvec, trasl,
        K, cv::noArray(), left_out_points);
    cv::projectPoints(
        right_in_points, rvec, trasl,
        K, cv::noArray(), right_out_points);

    cv::Point2i left_img_shaft_1(left_out_points.at(0));
    cv::Point2i left_img_shaft_2(left_out_points.at(1));
    cv::Point2i right_img_shaft_1(right_out_points.at(0));
    cv::Point2i right_img_shaft_2(right_out_points.at(1));

    bool left_in_borders = check_perimeter(
        left_img_shaft_2,
        cam_param_[cam].width,
        cam_param_[cam].height);
    bool right_in_borders = check_perimeter(
        right_img_shaft_2,
        cam_param_[cam].width,
        cam_param_[cam].height);

    cv::Scalar l_color(130, 255, 0);
    cv::Scalar r_color(130, 255, 0);

    if (mut_collision.try_lock())
    {
        l_color = colors[0];
        r_color = colors[1];
        mut_collision.unlock();
    }
    if (l_color != cv::Scalar(0,0,0))
    {
        if (left_in_borders)
        {
            cv::line(image,
                     left_img_shaft_1, left_img_shaft_2,
                     l_color, 3);
            cv::circle(image, left_img_shaft_1, 1.5, l_color, 1.5);
            cv::circle(image, left_img_shaft_2, 1.5, l_color, 1.5);
        }
        else
        {
            cv::circle(image, left_img_shaft_2, 2, l_color, 2);
        }
    }
    if (r_color != cv::Scalar(0,0,0))
    {
        if (right_in_borders)
        {
            cv::line(image,
                     right_img_shaft_1, right_img_shaft_2,
                     r_color, 3);
            cv::circle(image, right_img_shaft_1, 1.5, r_color, 1.5);
            cv::circle(image, right_img_shaft_2, 1.5, r_color, 1.5);
        }
        else
        {
            cv::circle(image, right_img_shaft_2, 2, r_color, 2);
        }
    }
}
```

Color information is read from the ROS topic message containing the evaluated set of distances and shared with the draw function via mutex; a parametric collision threshold discriminates between the three operational regions.

```
void VisualFeedback::onInfoReceived(
    const std_msgs::Float64MultiArray::ConstPtr &msg)
{
    collison_distances = msg->data;
    if (mut_collision.try_lock())
    {
        // left/right tools
        for (int i = 0; i < 2; ++i)
        {
            if (collison_distances.at(i) <= COLLISION_THRESHOLD)
            {
                // RED
                colors[i] = cv::Scalar(255, 76, 0);
            }
            else if (collison_distances.at(i) > COLLISION_THRESHOLD
                && collison_distances.at(i) < WARNING_THRESHOLD)
            {
                // YELLOW
                colors[i] = cv::Scalar(0, 255, 243);
            }
            else
            {
                // GREEN
                colors[i] = cv::Scalar(130, 255, 0);
            }
        }
        // collision_distance index 2 & 3 are for PSM1 and PSM2
        mut_collision.unlock();
    }
}
```

8 Contact Forces Estimation

As presented in section *Vision based force sensor* of deliverable D3.1 – Multi-modal human-robot interfaces and architecture for the MULTIROBOTS-SURGERY platform, the system evaluates contact forces by analyzing the images taken from the endoscopic stereo camera and reflects them on the user; this allows to avoid using external force sensors on the tooltips of the laparoscopy instruments without removing this useful feedback for the assistant surgeon. Information about the contact forces is transmitted via a custom ROS message defined as

```
float64 depth
geometry_msgs/Vector3 direction
```

where `depth` indicates the amount of estimated contact penetration with vector `direction` to provide the 3D contact location in the environment. With this information, the ROS node evaluates the contact force separately for each laparoscopy instrument as an elastic interaction with constant *Kskin*; the evaluation is computed as follows:

```
vnorm = sqrt(
    direction[0]*direction[0] +
    direction[1]*direction[1] +
    direction[2]*direction[2]);
geometry_msgs::Twist msg;
msg.linear.x = depth*Kskin*direction[0]/vnorm;
msg.linear.y = depth*Kskin*direction[1]/vnorm;
msg.linear.z = depth*Kskin*direction[2]/vnorm;
msg.angular.x = 0.0;
msg.angular.y = 0.0;
msg.angular.z = 0.0;
```

The force is, then, published as a `twist` message at a rate of $60Hz$.

9 Conclusions

This deliverable collects useful information about software and hardware aspects of the MULTIROBOTS-SURGERY platform. Several pieces of code are inserted to explain how the theoretical results described in other deliverables are implemented within the SARAS architecture.

This deliverable will be exploited as a reference for further improvements of the software and the hardware, and as a starting point for the deliverables D7.3 – Software/Hardware architecture for the SOLO-SURGERY platform and D7.4 – Software/Hardware architecture for the LAPARO2.0-SURGERY platform due at months 24 and 36, respectively.

References

[1] Huajian Deng, Hongmin Wu, Cao Yang, Yisheng Guan, Hong Zhang, and Jianling Liu. Base frame calibration for multi-robot coordinated systems. *2015 IEEE International Conference on Robotics and Biomimetics, IEEE-ROBIO 2015*, pages 1489–1494, 2015.

[2] Peter Kazanzides, Zihan Chen, Anton Deguet, Gregory S. Fischer, Russell H. Taylor, and Simon P. DiMaio. An open-source research kit for the da vinci surgical system. In *IEEE Intl. Conf. on Robotics and Auto. (ICRA)*, pages 6434–6439, Hong Kong, China, 2014.

[3] R. Y. Tsai and R. K. Lenz. A new technique for fully autonomous and efficient 3d robotics hand/eye calibration. *IEEE Transactions on Robotics and Automation*, 5(3):345–358, June 1989.

[4] Z. Zhang. A flexible new technique for camera calibration. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(11):1330–1334, Nov 2000.