Control architecture for docking UAVs with a 7-DOF manipulator

G.B. (Giuseppe) Barbieri

MSc Report

Committee:
Prof.dr.ir. S. Stramigioli
Dr. R. Carloni
Dr.ir. R.G.K.M. Aarts
M. Reiling, MSc

October 2016

045RAM2016
Robotics and Mechatronics
EE-Math-CS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Abstract

The SHERPA-project (www.sherpa-project.eu) focuses on developing a mixed ground/aerial robotic platform to improve human rescuers activities. In this collaboration, the human should be able to use the robots with ease when necessary, while the robot should be autonomous when control is not demanded by the operator.

The robotic platform consists of a robotic arm integrated with a ground rover, the task of the system will be to autonomosly localize, pick up and dock a UAV[1] in its neighbourhood to achieve a battery swapping. The MSc research focuses on designing and implementing a control architecture that integrates motion planning and control techniques to execute all the necessary steps of grasping and docking a UAV. During execution the architecture supervises the feasability of the given tasks to ensure correct execution.

[1] Unmanned Aerial Vehicle

Contents

1 Introduction

1.1 Context

1.1.1 SHERPA project

The introduction of robotic platforms offer a promising solution for the improvement of search and rescue activities in hostile environments, such as in the alpine scenario. This thesis has been developed under the project named "Smart collaboration between Humans and ground-aErial Robots for imProving rescuing activities in Alpine environments" (SHERPA) (Marconi et al., 2012), launched and funded by the European Union. The project aims to provide a robotic team to support human rescuers, with the purpose of increasing the operators' awereness, facilitating rescuing activities by reducing their work load.
The robotic team is composed of the following elements:

- *SHERPA rover*: the SHERPA rover acts as an "intelligent donkey" that carries load. It is equipped with a recharging station for small-scale UAVs, referred to as SHERPA box, and it is conceived as an hardware station with computational and communications capabilities, as well as a high-degree of autonomy and long endurance. It's autonomy capabilities are improved through a multifunctional robotic arm installed on it, which will be described in further detail in the next sections.

- *Small-scale UAVs*: the small-scale UAVs act as trained "wasps" and are small rotary-wing UAVs. They are used to support rescuing surveillance activity by enlarging the controlled area through the use of small cameras and various sensors.

- *Large glider UAV*: the large UAV is a long-endurance, high-altitude and high-payload aerial vehicle with complementary features with respect to the small-scale UAVs. It is referred to as the "Patrolling hawks" and it is in charge of creating a 3D map of the rescue area. Those maps inform the platforms in case critical terrain morphologies changes are identified.

The research focuses on how the human operator and the mixed aerial and ground robot platform collaborate with each other, toward the achievement of a common goal.
Although the ease and simplicity for the operator to coordinate and communicate with the platform, rescuing activities require great implications so that constant supervision of the platform seem hardly achievable by a single person. Therefore, the autonomous performances of the robot system represent a core part of this project.
The current development regards the implementation of a robotic arm integrated on the ground rover. The task of this system will be to autonomously localize, pick up and dock a small-scale UAV detected in the platform neighbourhood.
The scope of this master thesis project contributes in the increase of autonomy of the robotic system, specifically of the SHERPA arm. A control architecture is presented for controlling the SHERPA arm, leading towards the achievement of grasping and docking operation of the UAVs.

1.1.2 Problem statement

Small-scale UAVs, equipped with small cameras and various sensors, are excellent for increasing the surroundings awareness of the operator. However, they are characterized by a short battery life. In order to increase the UAVs autonomy, and overcome their limited battery life, they have to be recharged directly on field.
For this purpose, a mobile ground-vehicle equipped with a robotic arm works as a mobile recharging station for small-scale UAVs. The robotic manipulator has to be capable of grasping

the UAVs and docking them in the SHERPA box mounted on the ground-rover.
The execution of novel applications such as the one devised for the SHERPA arm, requires the introduction of safer control architectures that can cope with the need of operating in hostile environments. For this reason the supervision of given tasks is desirable at different level of abstraction(N.Xi et al., June 1996), (Yildirim and T.Tunali, 1999). Direct feedback of the world model should be provided to the task level of the robot software, which gathers the necessary informations to decide whether a certain task will be completed or not. Moreover, the path planner should generates free-collision trajectories and be able to cope with situations in which the arm has to interact with unknown environments, while supervising correct trajectory tracking.

1.2 Report Organization

A description of the SHERPA arm and a discussion of the software abstraction levels applicable to the robot will be found in Chapter 2. The paper concept, presenting the proposed control architecture, is then given in Chapter 3. The paper will also describe the methodology and implementation details of the software, the hardware used for realization, as well as the experiment results. In Chapter 4, additional information regarding methodology and software implementation are given. Finally, conclusions of the achieved work are drawn and recommendations for future works proposed in Chapter 5.

In addition, appendices describing the software's installation, Appendix A and Appendix B, and use, Appendix C, can be found at the end of this thesis.

2 Background and Analysis

In this chapter some background information on the system are given. Also, the approach for designing the control architecture is presented and discussed. Following, the requirements to be fulfilled for the present work are listed.

2.1 The SHERPA arm

The SHERPA arm is a light weight compliant manipulator (Barrett et al., 2016). The arm is designed such that it is extremely resilient against disturbances and shocks. This is achieved by introducing compliant elements which allows decoupling of the robot's structure and drives from rigid impacts. Moreover, the arm is equipped with two Variable Stiffness Actuators (VSA) (Vanderborgh et al., 2013), allowing for the adjustement of the mechanical stiffness of the robot's joints, adapting its dynamics behavior to different tasks.
In addition to its variable compliant DOF, the SHERPA arm has 7 active DOF: 3 DOF in the shoulder, 1 DOF in the elbow and 3 DOF in the wrist.
The arm's shoulder is a 3 DOF joint, in which the second and third DOF are actuated by two differentially coupled motors. The wrist is a 3-DOF joint, the last two of which are differentially coupled as well. The elbow joint consists of two axes connected by an intermediate link, driven by a single actuator. This allows the arm to fully fold into its transport configuration and to enlarge its workspace. The SHERPA arm can be observed in Figure 2.1.

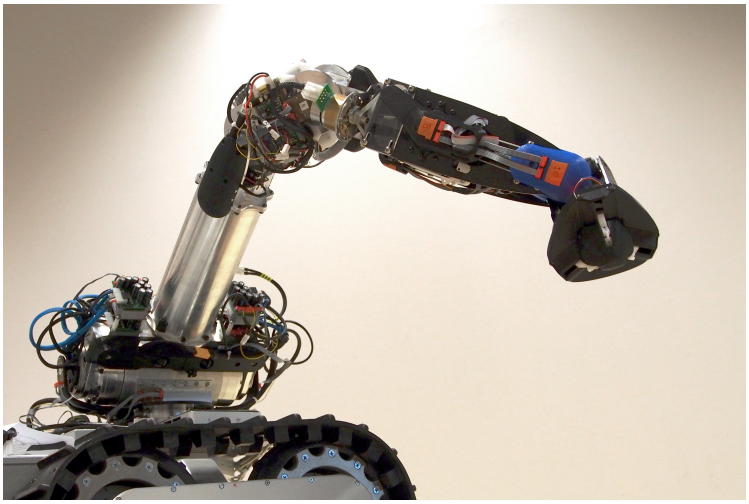


Figure 2.1: The SHERPA arm

2.1.1 Electronics

The SHERPA arm currently contains a total of eleven ELMO Whistle miniature digital servo drives[1] that locally control the actuators in position, velocity or current control. The ELMOs use feedback from the incremental motor encoders. In addition to the incremental encoders, every degree of freedom is equipped with 14-bit absolute magnetic encoders. The arm is also equipped with mechanical limit switches that are directly connected to the ELMO drives.
The communication is realized by connecting the motor controllers via the standard industrial CAN bus, while the encoders are connected through a separate SPI bus. The high-level control is run on an Intel NUC running Ubuntu 14.04.

[1] Elmo motion control ltd. - http://www.elmomc.com/products/whistl-digita-servo-drive-main.htm

2.1.2 Gripper

The SHERPA arm is equipped with a custom gripper (E.Barrett et al., 2016) with integrated actuation and electronics. The gripper ensures grasping of the UAV by latching onto an interface installed on the UAV as depicted in Figure 2.2. Its shape facilitates the grasping, which is easily drivable into the interface, ensuring simple pick-up operation. The gripper actuator is controlled with an Atmel ATmega328 microcontroller and Allegro A4953 motor driver.

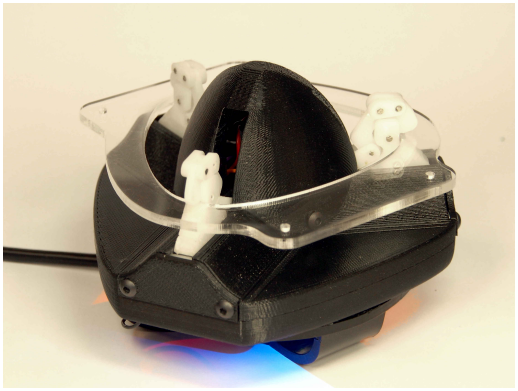


Figure 2.2: Gripper latched into the interface

2.1.3 SHERPA box

The ground-rover is equipped with the SHERPA box - Figure 2.3. This box provides communications capabilities in order to inform when a UAV has been successfully docked. It is equipped with a mechanism that autonomously replace the UAV's battery.

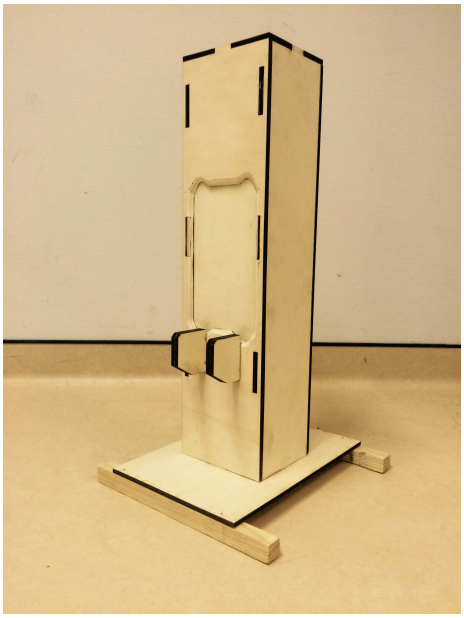


Figure 2.3: Dummy Sherpa box

With reference to the figure, the two tongues sticking out from the box are used to correctly position the drone on the box and guide it to its final docking position. For the scope of the project a dummy box has been used, demonstrating the feasibility of the docking operation.

2.2 Analysis

2.2.1 Current system control architecture

The current control architecture for the SHERPA arm is depicted in Figure 2.4 and provides the implementation of the Joint Controllers and a State Observer which interface with the hardware. The Joint Controller provides setpoints to the local drives placed on the arm every sampling period, which track them in hard-real time. The State Observer retrieves position feedback from the encoders on the arm.
The system works as follows: the user interfaces with the system via a fader panel in which set-points can be sent to each joint controller placed on the arm.
The current implementation includes the option of sending setpoints through a ROS topic as well, although is not used yet. A monitor is used for running a visualization of the arm in ROS (Rviz[2]).

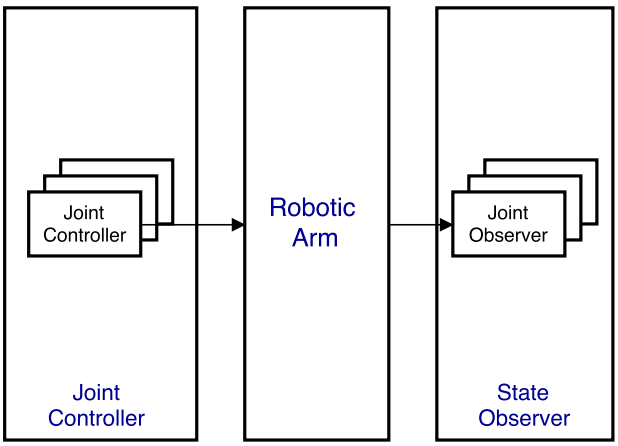


Figure 2.4: Current System Control Architecture

In the current state, all functionality are readily accessible through an interface for direct control. In order to increase the autonomy of the SHERPA arm, a higher-level control architecture has to be integrated with the current implementation of the joint controllers.

2.2.2 Layered control structure

For the realization of robotic systems that are capable of planning and executing tasks, a multi-level approach to robotic software architectures can be used. Hence, the overall control architecture can be designed by studying the layered controller structure proposed by (Broenink and Hilderink, 2001) and depicted in Figure 2.5.

According to the the proposed controller structure, the control processes should be divided over the range of hard and soft real-time. With reference to Figure 2.5, the embedded control software part is structured in the following layers:

- *Supervisory Control*: typically includes control tasks such as task planning, vision algorithms and environment mapping. It performs calculations in order to determine the tasks to be sent to the sequence controller.
- *Sequential Control*: enables and feeds the loop controller with setpoints and necessary parameters. It is implemented in soft real-time.

[2]ROS.org - Ros Visualizator http://wiki.ros.org/rviz

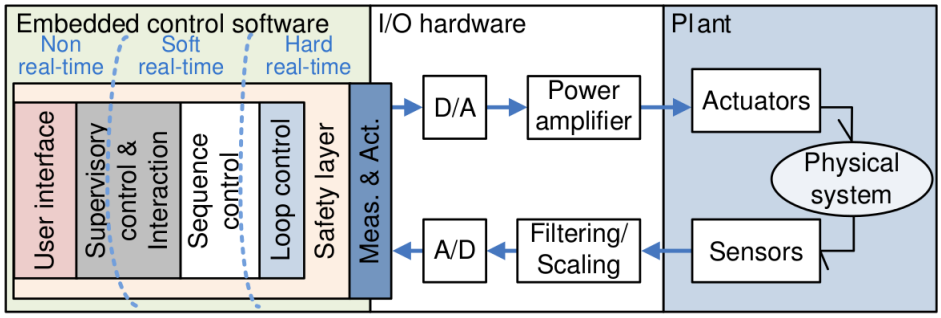


Figure 2.5: Control Architecture

- *Loop control*: contains the control algorithms for controlling the actuators. Those require a setpoint update every sampling period, hence it is implemented as hard real-time.
- *Safety Layer*: is the layer that examines for safety issues on all control levels.

The autonomous control function required to be implemented for the SHERPA arm can be mapped into the layered controller structure. This results in a model as depticted in Figure 2.6.

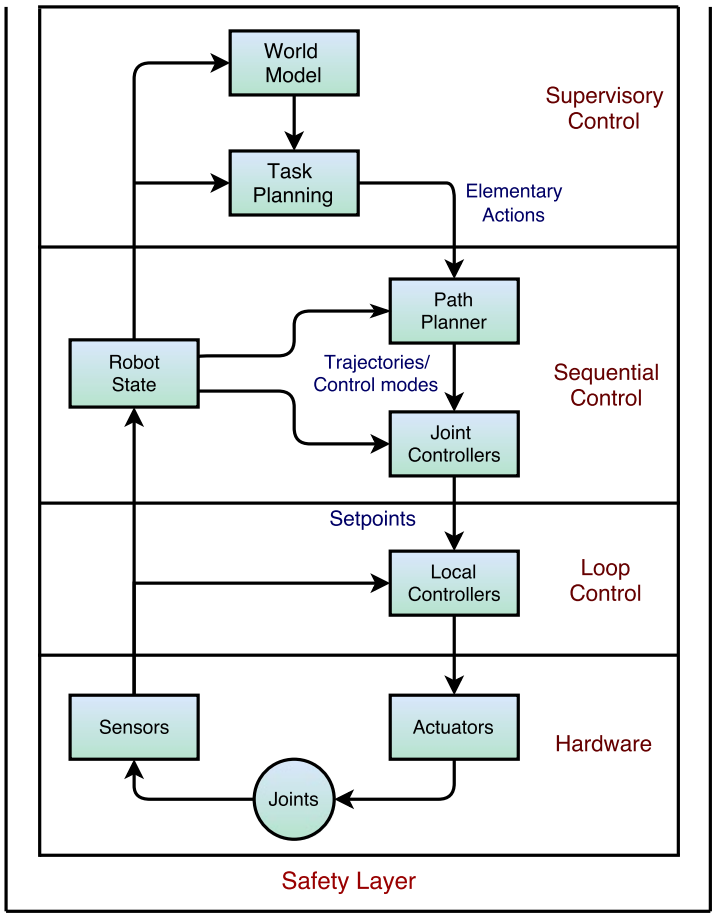


Figure 2.6: Control Architecture

The *Supervisory Control* contains the block *Task Planning* and *World Model*. *World Model* is the geometrical representation of the world, while *Task Planning* contains the sequence of operations allowing the accomplishment of given missions.
The *Sequential Control* contains three blocks: the *Robot State* observer, the *Path Planner* and the *Joint Controllers*. The *Path Planner* outputs the trajectories set-points and sends them to the *Joint Controllers* together with information regarding the required control mode. The *State Observer* provides feedback both to the *Path Planner* and the *Joint Controllers* given the obtained measurements from the sensors. This layer is implemented in soft real-time as the acquired results are useful even after the deadline is passed.
The *Loop Control* and its safety layer are hard-real time, as missed deadlines could results in system failures or unstable control actions for the joints. In this level the *Local Controllers* is the only functional block. The *Local Controllers* are in charge of tracking the set-points sent by the *Joint Controllers* nodes implemented in ROS, using the current joint positions coming from the motor's incremental encoders.
The safety layer covers all the software layers since each one of them include their own safety methods, preventing possible damages to the robot.

For the scope of this thesis the Task Planner is designed as the composition of four pre-defined Elementary Actions that are run in sequence. In the future, nevertheless, it will be possible to implement different more complicated tasks for the SHERPA arm.
The analyzed possible structure consists of several function blocks which are explained in further details in Section 3.

2.3 Approach

As previously mentioned, our software is tested in the scenario where a small-scale UAV is picked up and docked. When its battery is running out, the drone will land in proximity of the robotic ground platform. It will then be possible to pick it up and dock it onto the Sherpa box, which handles the battery replacement.
The whole procedure is accomplished in different phases. The first phase consists of reaching the drone with the robotic arm and executing a grasping operation. Once the drone is grasped, it has to be placed in proximity of the SHERPA box and subsequently docked.

With reference to the diagram depicted in Figure 2.6, the different phases are supervised and executed by a Task Planner which provides target locations to the Path Planner. The Path Planner handles the execution of the task by commanding the Joint Controller and using the feedback that the Robot State obtains from the joint sensors. Therefore, the used methodology in order to accomplish such described task should consists of:

- An *inverse kinematic* control algorithm that takes desired Cartesian poses as input and output robot's joint configurations.
- A *motion planning* algorithm which generates a stream of setpoints given a start and a goal end-effector pose while obeying the arm's kinematic constraints.
- An *impedance controller* to handle situations in which the arm has to interact with an unknown environment. The impedance controller takes desired position and stiffness as input and output torques setpoints for the joints.

2.3.1 Requirements

The requirements for the software development presented in this work are derived from the previous subsections and are divided into technical and functional.

From a technical point of view, the requirements are the following:

- the system must be able to interface with the hardware described in the previous subsection.
- The system has to run on a NUC Intel mounted on the SHERPA rover running Ubuntu 14.04.
- The design of the software has to be scalable such that further autonomous behaviours can be implemented in the future.

The requirements for the desired behaviour are the following:

- The arm needs to reach all the positions required for the specific mission, such that its maneuverability is ensured in its workspace.
- The arm has to execute trajectories by avoiding collisions with obstacles present in the environment.
- The arm needs to interact with the environment in a physically compliant, versatile and robust way.
- Autonomous pick and dock routine is performed upon operator request.
- The robot can autonomously cancel the execution of commands if those result in dangerous behaviour, or if the controllers are faulty.
- When there is the necessity to change the control mode during a task execution, it should be autonomously handled with no need for the operator's intervention.

3 Control architecture for docking UAVs with a 7-DOF Manipulator

Control Architecture for Docking UAVs with a 7-DOF Manipulator

Giuseppe Barbieri, Mark Reiling, Eamon Barrett and Raffaella Carloni

Abstract— **This paper presents the design and implementation of a control architecture to increase the autonomy of a 7-DOF manipulator mounted on a ground rover, used to grasp and dock small-scale unmanned aerial vehicles (UAVs). The overall goal of the controller is to combine efficiency of motion planning algorithms in finding collision-free trajectories and impedance controller to handle situations in which the arm is in contact with unknown environments. The architecture supervises the feasability of the given tasks in order to ensure correct execution.**

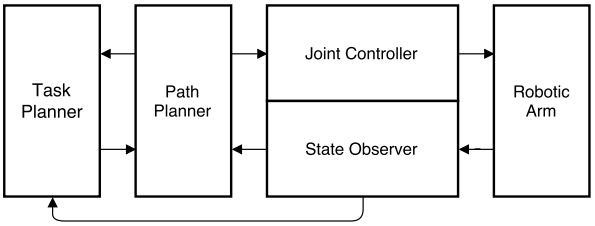


Fig. 1: Overview of the overall control architecture

I. Introduction

The introduction of robotic platforms is seen as a promising solution in improving search and rescue activities in hostile environments as proposed in the european project named SHERPA. The project aims to provide a robotic team which includes a mix of ground and aerial robotic platforms to assist human rescuers [1].

During long rescue operations, small-scale UAVs, equipped with cameras and other sensors, gather data on the field with the goal of enlarging the rescuer's operational environment. However, due to their limited battery life, UAVs cannot guarantee full autonomy during the entire mission. Therefore they have to be autonomosly recharged on field.

For this purpose, a mobile ground-vehicle equipped with a robotic arm behaves as a mobile recharging station for the UAVs. The robotic arm is a 7-DOF lightweight compliant manipulator which should grasp the UAVs and dock them on the ground-rover.

Novel applications such as the tasks of the SHERPA arm require interaction of the robot with unknown environments introducing the necessity for safe control architectures.

Industrial manipulators are generally used in well-defined environments where they are required to perform precise tasks, accomplished with the use of classical control architectures. The ability to operate in unknown environments would considerably increase the applicability, but it also sets new requirements on the robot control system and sensors.

For the realization of robotic systems that are capable of planning and executing tasks, a multi-level approach to robotic software architectures is well recognized and used [2][3]. In some approaches such as [2], various feedback information are sent to the low-level control layers to ensure robust performance. In this case the path planner can be considered as a time-based memory component for storing predefined plans that lacks feedback from the environment and cannot cope with unpredictable changes in the world model. Other works, such as [4], proposes an event-based approach directly integrated into the planner which can take decisions based on sensory measurements. Studies such as [5] propose the introduction of a high level task planner which generates a sequential plan of operations based on the feedback retrieved from sensors and is entrusted with replanning in case unexpected events occur. Those subtasks are then sent to a motion planner which generates the necessary trajectories for the joints but lacks direct feedback.

This paper, introduces a control architecture that combines the approaches described in [5] and in [4] and extends the paradigm with the possibility of executing given subtasks that requires the computation of plans with different behaviours. The Task Planner manages the sequential execution of a set of subtasks defined in Cartesian space and sends them to the Path Planner which sequentially translates them into commands for the joint controllers while monitoring execution. Depending on the task the control mode is autonomously switched and the proper commands are generated. Moreover, the feasability of a specific subtask is managed by the Task Planner which retrieves sensor's feedback during execution. If the environment changes during the operation or a subtask has not been successful, the model is updated and the remaining operations can be stopped or replanned.

The overall architecture of the system is depicted in Figure 1 and includes: the Task Planner, the Path Planner, the Joint Controller, the State Observer for feedback and interfaces to the hardware.

The rest of the paper is organized as follows. First, the general control architecture is analyzed, then the implementation is detailed. Finally, the overall control architecture is validated both in simulation environment and experiments.

This work has been funded by the European Commission's Seventh Framework Programme as part of the project SHERPA under grant no. 600958.

G. Barbieri, M. Reiling, E. Barrett and R. Carloni are with the Faculty of Electrical Engineering, Mathematics and Computer Science, CTIT Institute, University of Twente, The Netherlands (e-mail: {g.barbieri,m.reiling, e.barrett, r.carloni}@utwente.nl).

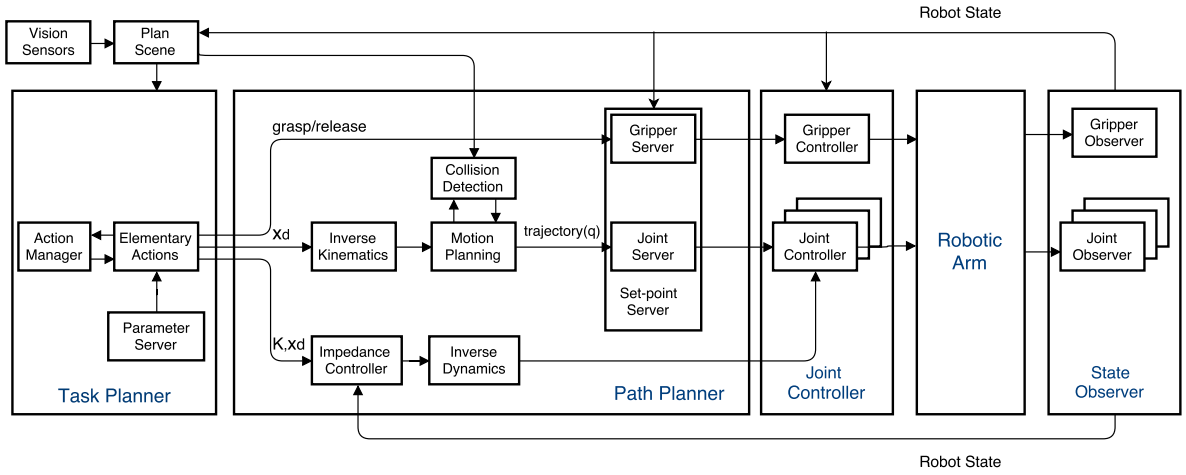


Fig. 2: Detailed scheme of the proposed Control Architecture

II. Control Architecture

This section describes in further detail the control architecture depicted in Figure 1.
The architecture allows to execute the tasks of reaching, grasping and docking a UAV by following point-to-point motions in both Cartesian and Joint space.
The total system consists of five top-level blocks; the Task Planner, the Path Planner, the Joint Controller, the State Observer and the Robotic Arm.
It also contains the Plan Scene which consists of the geometric representation of the environment and is obtained by fusing the data from vision sensors and encoders.

A. Task Planner

The Task Planner handles the execution of a given task, which is divided into subtasks referred to as Elementary Actions. The modules that define the Task Planner are detailed in Figure 2.
In general, the Action Manager receives updates on the world from the Plan Scene and manages the sequential execution of the Elementary Actions when a UAV is detected. By monitoring the state of the robot and of the world during execution, the Task Planner can be aware of changes in the environmental's conditions and decide whether to stop the given mission.
The blocks that compose the Task Planner are described in further detail in the next sections with the exception of the Parameter Server.
The Parameter Server is a shared, multi-variate dictionary which is used to store and retrieve parameters at runtime. The Task Planner uses the Parameter Server to obtain configuration information including robot kinematics, joint limits, planning variables and trajectory constraints.

B. Path Planner

The Path Planner computes the commands that are sent to the Joint Controllers while monitoring execution. It is defined as an interconnection of different modules that are detailed in Figure 2.
In general, once the UAV is detected, the Task Planner sequencially executes the proper subtasks by providing target locations to the Cartesian controllers that constitute the Path Planner, which computes the necessary commands for the Joint Controller. These are then sent to the Joint Controller through the Set-point Server which provides an interface to ensure safe execution.
The blocks that compose the Path Planner and generate the proper commands for the joint controllers are described in further detail in the next sections.

C. Joint Controller

The Joint Controller tracks the trajectories generated by the Path Planner and sends the corresponding set-points to the hardware. It is composed of controllers for each arm's joint and accepts position, velocity or torque set-points. The Joint Controller interfaces with the Path Planner through the Set-point Server.

D. State observer

The State Observer shares feedback on the robot state to the Path Planner. It estimates the internal state of the robot's joint such as position, velocity, acceleration, torque and stiffness by retrieving sensor feedback.

E. Robotic Arm

The Robotic arm is a 7-DOF light weight manipulator, referred to as SHERPA arm [6]. The arm is designed with compliant elements which allows decoupling of the robot's structure and drives from rigid impacts. This makes the arm extremely resilient against disturbances and shocks. Moreover, the arm is equipped with two Variable Stiffness Actuators (VSA)[7], allowing for the adjustment of the mechanical stiffness of the robot's joints.
The arm is equipped with a custom gripper [8] which ensures grasping of the UAV by latching the interface installed on the UAV.

III. Task Planner

This section describes in detail the design choices taken in implementing the Task Planner.

A. Elementary Actions

To realize pick-and-place functionality, the problem is divided into subtasks referred to as Elementary Actions.
The Elementary Actions are run in sequence and are defined in the following way:

- *Idle*: In this stage the arm is inactive. The UAV is not detected within the arm's workspace.
- *Reaching*: This phase starts when the UAV is localized into the robot's workspace. The UAV pose is provided to the controller which computes the free-collision trajectory to place the manipulator's end-effector in proximity of the object.
- *Grasping*: At this point, the manipulator's end-effector is controlled to safely reach the grasping position. Desired position and desired stiffness are provided to the Path Planner which computes safe Cartesian trajectories to handle interaction between the manipulator and the UAV. Once the grasp position is reached, the gripper is latched into the UAV interface.
- *Placing*: This action requires the planner to a compute collision-free trajectory from the grasp pose to a location in proximity of the docking position. In this phase, the UAV is attached to the arm.
- *Docking*: In this phase the drone is safely docked on the SHERPA box. The arm's compliance is controlled to handle the interaction between the UAV and the SHERPA box. Once the UAV has been succesfully docked, the gripper releases the UAV.

B. Action Manager

The Action Manager handles the execution of the state-machine schematically represented in Figure 3. It keeps track of the Elementary Actions being executed and it is designed to facilitate the retrieval of feedback information during task execution.
The Action Manager was designed in order to coordinate separate systems that need to work sequentially.
It consists of a set of global variables, and it works as a repository of messages that can be accessed by separate processes such as the Elementary Actions.
This design choice was made because even though the robot status is constantly received from the Plan Scene, it is not necessary to continuously update it and notify the Elementary Actions. In this way, the separate processes can request the robot state in order to validate whether or not a certain state has been achieved.

IV. Path Planner

This section describes in detail the design choices taken in implementing the path planner.

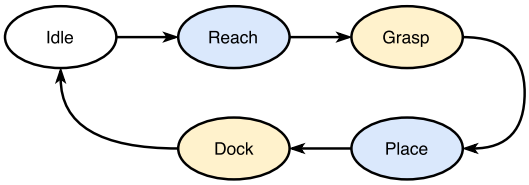


Fig. 3: Internal state machine diagram of the Action Manager. In blue: actions executed in free-space motions. In yellow: actions that requires constrained motions

A. Inverse Kinematics

The execution of the elementary actions requires direct motions in the robot's Cartesian workspace.
As previously mentioned, the arm used in this work is a 7-DOF manipulator operating in a six dimensional workspace. The forward kinematics of the serial manipulator can be described in the form:

$$x = f(q)$$

where $x \in R^6$ denotes the position and orientation of the end-effector in operational space, $q \in R^7$ denotes the position of the joints, and $f(q)$ the forward kinematic model of the robot.
The inverse kinematic is then described as:

$$q = f^{-1}(x)$$

Finding a closed-form solution to the inverse kinematic's problem poses more than a challenge for redundant manipulators since infinite solutions may exist.
For direct workspace motions, Jacobian inverse-kinematics is a well studied approach [9]. The manipulator Jacobian is used to map desired cartesian motions to joint-level commands for the robot.
The typical inverse kinematic method is a closed-loop scheme in which the desired pose x_d is taken as a reference and compared to the actual state x_e. The error e is then used to compute the joint positions to add to the current robot's state.
The selection of the kinematic controller was based on robustness when dealing with joint limits, convergence issues and singularities. Based on these considerations, the damped least squares method was proposed among the control based kinematic solvers [9].
This method consists of finding the solution Δq as the value that minimizes:

$$||J\Delta\mathbf{q} - \mathbf{e}||^2 + \lambda||\Delta\mathbf{q}|| \tag{1}$$

with $\lambda \in R$ being the non-zero damping constant. As described in [10], this results in:

$$\Delta\mathbf{q} = (J^T J + \lambda^2 I)^{-1} J^T \mathbf{e} \tag{2}$$

It can be seen that when $\lambda = 0$, equation (2) reduces to the classic pseudo-inverse scheme described by $\Delta\mathbf{q} = J^{\dagger}\mathbf{e}$.
In order show robustness near singularities, solution (2) was analyzed with the method of singular value decomposition. According to [9], this results in:

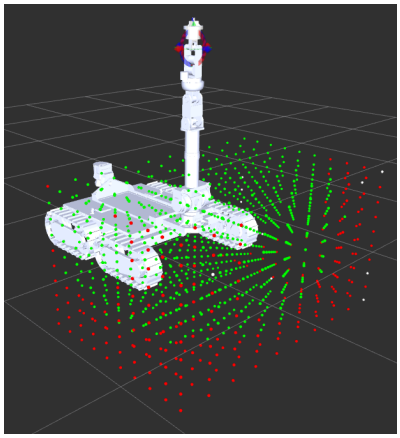


Fig. 4: Visualization in Rviz of kinematic solver benchmarking. Test run on 1000 poses each with 6 end-effector orientations

$$J = \sum_{i=1}^{r} \frac{\sigma_i^2 + \lambda^2}{\sigma_i} \mathbf{u}_i \mathbf{v}_i^T \tag{3}$$

as a result the pseudo-inverse jacobian can be written as:

$$J^{\dagger} = \sum_{i=1}^{r} \frac{\sigma_i}{\sigma_i^2 + \lambda^2} \mathbf{v}_i \mathbf{u}_i^T \tag{4}$$

This expression shows that the introduction of a damping constant allows the damping of the inverse kinematic solution near singularities. This method implies considerations on the value of the damping factor. For small λ, it behaves as the ordinary pseudo-inverse since $\sigma_i^2 >> \lambda^2$, resulting in accurate solutions but low robustness. For large values of λ, we obtain $\sigma_i \to 0$, hence the limits tends to 0 instead of infinite. However this results in low tracking accuracy.

In this context, homogeneous transformations are used to calculate the Cartesian coordinates for the end-effector position. Those are defined in the arm's base frame of reference in which the direct kinematics of the manipulator are expressed. The UAV location is then defined as the matrix that describes the transformation between the base and the object frame: H_o^b. This can be expressed as a combination of the current end-effector pose and the object pose in the end-effector frame $H_o^b = H_e^b H_o^e$. After the object pose has been defined, a translation is computed in order to place the end-effector frame on top of the UAV. The quality of the used solution was tested in comparison with a standard pseudo-inverse method and the results are reported in Table I. Those results allowed to indirectly analyze the reachable workspace of the arm. Figure 4 shows that the chosen kinematic solver allows for the reach of several positions in which the UAV could be located for different end-effector orientations.

TABLE I: Behaviour of studied methods concerned with joint limits for 6000 samples with JP classical pseudo-inverse control scheme and JD damped least square scheme

Method	solved (%)	time solution (ms)
JP	75.71	1.46
JD	93.13	0.91

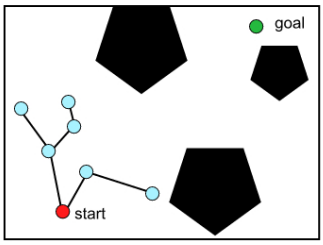


Fig. 5: Inizialization of RRT*

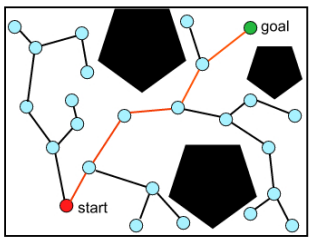


Fig. 6: The goal is included in the tree concluding the search

B. Motion Plannner

Consider $W \in R^6$ as the space in which the robot and obstacles are geometrically described, and $C \in R^7$ as the set of all the possible robot's configurations.
The motion planning block samples the configuration space and approximates its connectivity with a graph structure. Each sample is then checked in order to compute the free configuration space, referred to as C_{free}. This represents the subset of robot's configurations that are not in a self-collision or in contact with environmental obstacles.
With $x_d \in W$ being the pose of the UAV in the Cartesian workspace, and $x_e \in W$ the current pose of the robot's end-effector, the final end-effector pose of the arm can be found by using the kinematic controller previously described.
Once the inverse kinematic solution is found by taking into account joint limits, the sample is added into the motion planning path. The collision avoidance algorithm ensures that the computed states are enclosed in C_{free} by returning a list of contact data that identify collision points for the selected sample. This is done by representing volumes through dynamic boxes which are recomputed each time the robot's state or the environment change.
At this point, the motion planning block proceeds in finding a suitable collision-free path, $p(xe, xd)$, between start and goal state by using the *sampling-based* motion planning algorithm RRT*[11]. This algorithm creates a tree structure of C_{free}, rooted in the start configuration of the robot as shown in Figure 5. The tree is then heuristically expanded by the planner toward the goal configuration, choosing the resulting feasible path that minimizes a given cost function $c(x)$, such as the length of the path. When the goal configuration is reached, the search can be considered complete as depicted in Figure 6.

The solution path is generated as a function $[0, 1] \to C$, and does not prescribe how this path should be followed by the controllers. Therefore, a post-processing routine is applied to represent the path as a time parametrized function $[0, T] \to C$, where T is the planning horizon.
Finally, the trajectory is generated. The trajectory consists of a set of waypoints. A waypoint is a joint configuration, described by the tuple (p, v, a, t) where $p \in R^7$ are the positions, $v \in R^7$ the velocities and $a \in R^7$ the accelerations at time t.
Since the selected algorithm is probabilistically complete

[11], in case a target position is unreachable or an inverse kinematic solution is not enclosed in C_{free}, the motion planning algorithm will conclude that it is not possible to find a solution to the motion planning problem. In this scenario, the planner reports that the target is unreachable.

C. Impedance Controller

In order to handle interaction with unknown environments, a simple impedance controller has been integrated into the Path Planner.
Impedance control provides a common control approach to cope with contact between robotic arms and environment, as well as to maintain interaction forces within some desired level [12]. The idea is to describe the desired stiffness of the end-effector in Cartesian space by enforcing the relation between the desired force response $F \in R^6$ and the deviation Δx from the desired position. This is achieved by establishing a mass-damper-spring relationship between the Cartesian position x_d and the Cartesian force F such as:

$$K\Delta x + D\Delta\dot{x} + M\Delta\ddot{x} = F \tag{5}$$

where M, D and K are diagonal and positive definite matrices representing desired inertia, damping and stiffness of the end-effector respectively. The desired inertia is defined as the intrinsic intertia of the robot in its current state. In this way, the control objective described in Equation 5 can be achieved without the need of force-feedback loop with the following control:

$$K\Delta x + D\Delta\dot{x} = F \tag{6}$$

When speed is essential an high stiffness K is desired, while a small K is advisable when interaction forces should be compensated. The current end-effector position $x = f(q)$ is calculated with the use of forward kinematics. Adopting the transposed manipulator Jacobian $J^T(q)$, the Cartesian force F is transformed into desired joint torques:

$$F = K\Delta x + D\Delta\dot{x} \rightarrow \tau_j = J^T(q)F \tag{7}$$

The joint torque controller, referred to as T, generates the corresponding motor torque commands $\tau_m = T\tau_j$.
The commanded torques are combined with a feedforward command τ_f containing gravity and friction compensations torques.
The Impedance Controller receives from the Task Planner the desired Cartesian trajectories defined in end-effector frame and the desired Cartesian stiffnesses for executing grasp and dock actions. The Impedance Controller contains a server that handles the Cartesian trajectories sent by the Task Planner and ensure proper tracking. The set-points are then translated into proper torque commands by computing Inverse Dynamics as described in (7).
In this context, the object position is described with respect to the end-effector frame by using the homogenous matrix H_o^e. This allows for the simplification of calculation as, once the drone is reached or placed, movements along one of the end-effector reference frame's axis can be executed, as depicted in Figure 7 and 8.

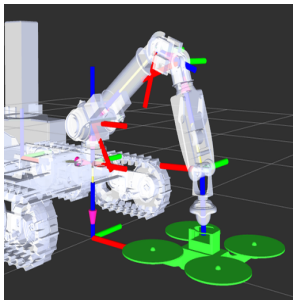


Fig. 7: Grasp

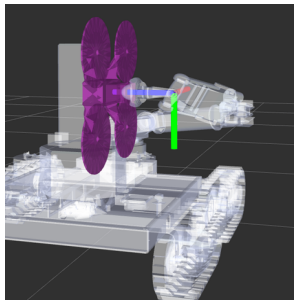


Fig. 8: Dock

The server handles the conversion of the given trajectories from end-effector reference frame to the arm base frame.

D. Set-point Server

The Set-point Server provides an interface to ensure correct trajectory execution.
With reference to Figure 2, it is composed of two blocks: Gripper Server and Joint Server.
The Gripper Server handles trajectories sent to the gripper, while the Joint Server manages joint trajectories. The Gripper Server receives pre-defined goal set-points which command grasp and release operations.
The Joint Server receives the entire joint trajectory output necessary to move the arm from its initial pose to the desired pose. Subsequently, it publishes the trajectory to the controller.
The trajectory tracking is ensured by mantaining a timer which guarantees that the duration value associated with each point of the trajectory is reached by the controller.
The server also enforces constraints on the trajectory, such as position and timing constraints. It allows to abort trajectories in case constraints are violated or whenever the controller suddenly stops responding.

V. Implementation

This section covers the implementation of the design described in Section 2.

A. Software Implementation

The proposed architecture was developed within the ROS-based MoveIt! framework [13][14], which provides a platform for developing robotic applications and incorporates tools for motion planning, manipulation and control.
The motion planning algorithm and the kinematic solver have been integrated as plugins in the MoveIt! framework.
The central part of the framework is the `move_group` node. This node is used as an integrator which pulls all the plugins present in the MoveIt! environment. The `move_group` node is also in charge of creating and maintaining the geometrical model of the environment referred to as Plan Scene. In this model it is possible to explicitly include environmental obstacles or other objects within the robot workspace via corrensponding topics (CollisionObject or

PlanningSceneDiff). In some cases, such as grasping or docking, collision between the manipulator and the object is intended. For this purpose, it is possible to modify the Allowed Collision Matrix within the Plan Scene in which objects can be dynamically included or excluded.
To ensure time parametrization of the computed paths MoveIt! offers an iterative hyperbolic time parameterization algorithm which translate the planned path into a trajectory for the controllers, while respecting imposed velocity and acceleration limits. The kinematic description of the robot is defined in the URDF format [15], a markup language designed to describe serial-chain robots.

The Set-point Server is implemented by using the *actionlib* stack, which provides tools to create client-server models to interface with preemptable tasks in ROS. The client part is in charge of sending goals or cancel requests. The server handles the goal execution, and provides the client with status of the goals present in the system, feedback on the goal execution and result about completion of a goal.
This model is also used in the implementation of the Impedance Controller. The client provides to the controller Cartesian trajectories while the server controls that the right torque commands are sent and executed by the Joint Controller.
In the servers, watchdog timers are implemented in order to detect faults and abort a commanded trajectory in case something goes wrong.

The sequential functionality of the Action Manager has been implemented by using ROS services. This is based on the need of getting the robot state information at specific times during execution. This communication pattern is done via a *service*, which is defined by a pair of messages: one for the request and one for the reply. A providing ROS node offers a service under a string name, and the client then calls the service by sending the request message and waiting for the reply. The Action Manager answers by sending the current state of the action being executed. In this way, each action can be executed only once the previous one was completed in the right way. Custom service messages were implemented to represent the status of a certain action. Figure 9 shows the communication diagram between the client classes and the Action Manager.

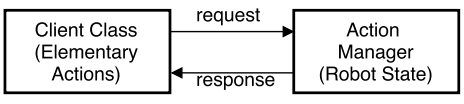


Fig. 9: Communication between elementary actions and Action manager

B. Planning algorithm and Cartesian controllers

The motion planning algorithm is based on the Open Motion Planning Library [16], and the main pipeline architecture is inspired by MoveIt!, however cartesian constraint and state machine for sequential tasks were added. OMPL is an open source motion planning library that contains most of the state of the art sample based algorithms such as RRT [17], PRM [18]. For collision detection, Flexible Collision Library (FCL) [19] has been used. This library offers different types of proximity queries on geometric models composed of triangles.
Kinematics and dynamics library (KDL) [20] is a library for computing forward and inverse kinematic queries with numerical solution. It was used to implement and integrate the aforementioned schemes for kinematic and Cartesian Impedance Controller.

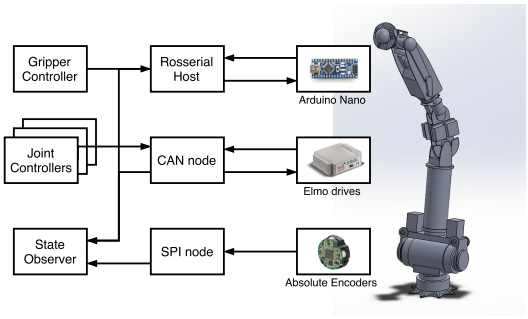


Fig. 10: Communication between ROS and hardware

C. Communication between hardware and ROS

The communication between ROS and the hardware is realized by the implementation of three ROS nodes that handle the information flowing to the drives and converging from the encoders.
The connections are schematically represented in Figure 10. The actuators on the arm are controlled locally with ELMO Whistle motor drives [21] that use feedback from incremental motor encoders. The motor controllers are connected via standard industrial Controller Area Network (CAN) bus. In addition to the incremental motor encoders, every DOF is equipped with 14-bit absolute magnetic encoder. These are connected via a separate Serial Peripheral Interface (SPI) bus. The gripper is controlled through an Arduino Nano[22], which communicates with ROS through *Rosserial* package which is a protocol for wrapping standard ROS serialized messages. The ROS side works as a serial server while a client library is installed on the Arduino. Parameters relative to the drives, such as transmission ratios and calibration of the encoders, are stored in configuration files and initialized at initialization.

VI. Experiments

This section presents some preliminary experiments. The experiments show that the performance of the task mission described throughout this paper is indeed achieved by the designed controller.
In the experiments, the task given to the robotic arm was to reach, grasp, place and dock the UAV. The UAV has been placed in a pre-defined Cartesian position, namely $x = 0.6\ m$, $y = 0.0\ m$ and $z = 0.0\ m$. For now, the

impedance controller has only been tested in simulation as no appropriate dynamic compensation has been implemented in the State Observer. For contact operations, a Cartesian stiffness has been set manually to safely execute grasping and docking.
Vision sensors are not yet implemented in the setup, hence the location of the UAV is assumed to be known.

A. *Hardware Setup*

The SHERPA arm mounted on the SHERPA rover is used for the experiments.
The control architecture runs on an Intel NUC running Ubuntu 14.04 which is placed on the rover.
An Optitrack Motion-Capture System [23] with 10 cameras is used to get absolute state feedback on the arm. The data is streamed from a dedicated computer over network towards another Ubuntu computer. Visual feedback is provided via a monitor connected to the NUC running Rviz[1].

B. *Experiment results*

During the experiments the controller ran at a rate of 30 Hz. The state of the robot was updated at a rate of 100 Hz. Note that the state observer was not complete during the execution of those experiments, hence deviations from the desired trajectory are due to the fact that no dynamic compensation is integrated in the arm.
Figure 11 shows a successfull trial where the arm completes the whole sequence. Starting from the idle state, it reaches the UAV (1) and proceed with the grasping (2). When the grasping has been achieved the arm proceeds in placing the UAV in proximity of the box (3) and docks it (4).

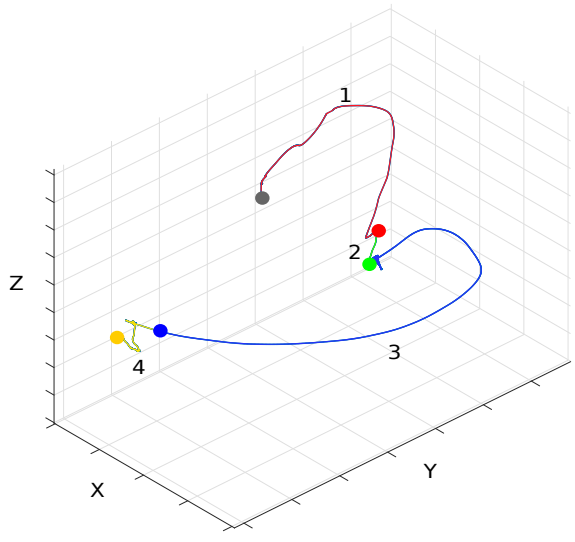


Fig. 11: The end-effector trajectory during the docking maneuver. The arm moves from the transport to the reach position, where it grasps the UAV. Subsequently the UAV is placed in proximity of the SHERPA box, where it will be docked.

[1]Ros Visualizator

Figure 12 shows the executed trajectory in comparison with the desired trajectories computed by the Path Planner. These plots show, respectively, the x, y, z vs $time$.
All plots initiate at the moment in which a new task request is sent by the user. As can be seen in the plots, the path is generated after a certain amount of time ($0.18s$).
A significant amount of deviation error between the desired and measured positions is visible in the plots. This can be interpreted as the absence of dynamic compensation in the arm, imprecision of measurements with the OptiTrack and rough calibration of the position sensors. However, it is observable that the arm follows the computed path in all stages.
During docking the deviation from the desired trajectory increases since disturbing factors such as gravity and play of the shoulder further affects the arm's performances.
In order to allow execution of the complete sequence, the constraint limits were relaxed to 0.25 m.

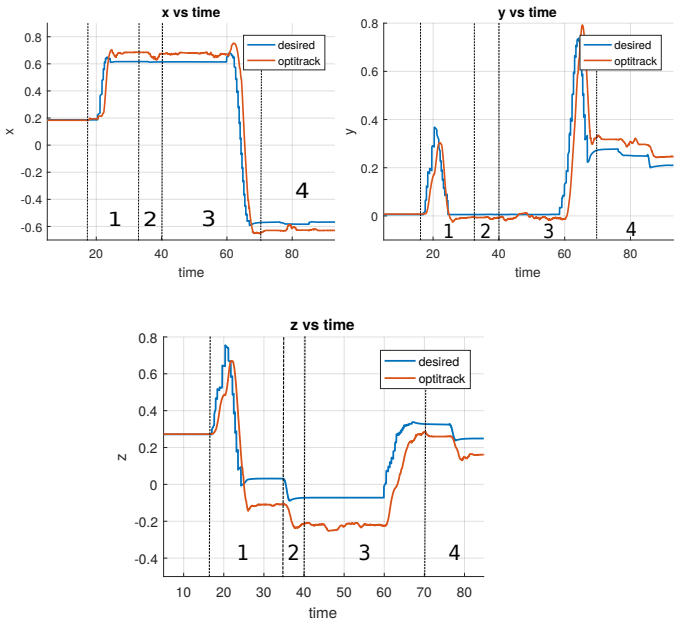


Fig. 12: Result plots of one of the trials from the experiment in compliance with the 3D plot in Figure 11

In this context repeatability and precision of the computed paths in reaching the UAV has been measured.
The mean values and standard deviations are listed in Table II .

TABLE II: Measured precision of position and orientation of the end-effector for 10 reaching trials

	$x(m)$	$y(m)$	$z(m)$	$r(rad)$	$p(rad)$	$y(rad)$
$\bar{x}$	0.5176	-1.018	0.0344	0.586	0.0936	-0.0231
σ	0.0104	0.0153	0.0058	0.0053	0.0315	0.012

From the values listed in Table II it can be seen that the maximum deviation from the mean value is about 1.5 *cm* in the x-direction, 1 *cm* in the y-direction and 0.5 *cm* in the

z-direction.
Regarding orientation, the max deviation is about $0.03\ rad$. From the mean values of the orientation, it can be seen that the arm indeed reaches the desired orientation ($roll = 0.57$ *rad*, $pitch = 0$ *rad*, $yaw = 0$ *rad*). This results shows that the computed path is reliable and allows the arm to reach the desired position each time.

More experiments were run to show behaviours of the arm in presence of obstacles between the start and end pose.
The end-effector trajectory performed by the arm, and measured by the OptiTrack system, is shown in Figure 13. In this experiment, a box of dimension $x = 0.118\ m$, $y = 0.130\ m$ and $z = 0.578\ m$ is inserted between the arm's initial state and the UAV location. The plot shows two trials. Reaching and grasping have been executed with and without the box. The plot shows that the computed path in presence of the obstacle allows the arm to smoothly avoid the box.

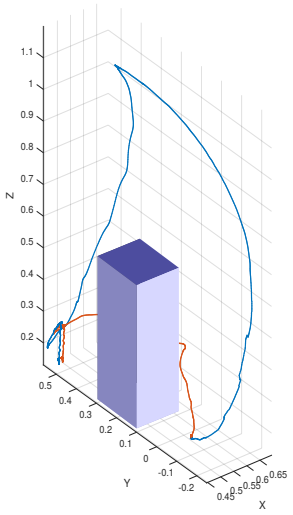


Fig. 13: Obstacle avoidance plot. In red the end-effector trajectory without obstacle, in blue the end-effector trajectory computed once the obstacle has been introduced in the workspace

VII. Conclusion and Future Works

This paper has presented a control architecture for docking UAVs with a robotic manipulator. The advantage of the proposed method lies in supplying feedback information about the world model to the task level of the robot software which can evaluate unexpected changes. Moreover, the trajectory execution is supervised by the Path Planner through the Set-point Server. This reduces the cognitive workload of the higher-level Task Planner and offers a framework that facilitates execution of more complex state-machines. The approach was implemented for our specific scenario and important aspects of the implementation were discussed. An RRT* motion planning algorithm in combination with a kinematic controller was used to generate collision-free paths for executing free-space motions, while a simple Impedance Controller was integrated to handle interaction with unknown environments. The advantage of this approach lies in the possibility of planning paths for the controllers with different behaviours.
Experiments showed the execution of the given task also in presence of obstacles. Future works will focus on the integration of vision sensors and the implementation of proper dynamic compensation, in order to reduce the amount of error between the measured and desired trajectories.

References

[1] L. Marconi, C. Melchiorri, M. Beetz, D. Pangercic, R. Siegwart, S. Leutenegger, R. Carloni, S. Stramigioli, H. Bruyninckx, P. Doherty, A. Kleiner, V. Lippiello, A. Finzi, B. Siciliano, A. Sala, and N. Tomatis, "The SHERPA project: Smart collaboration between humans and ground-aerial robots for improving rescuing activities in alpine environments," in *Proceedings of the IEEE International Symposium on Safety, Security, and Rescue Robotics*, 2012, pp. 1–4.

[2] M.Brady, J. Hollerbach, T. Johnson, T. Lozano-Perez, and M.T.Mason, *Robot Motion: Planning and Control*. MIT Press, 1986.

[3] J. F. Broenink and G. H. Hilderink, "A structured approach to embedded control systems implementation," in *Proceedings of the 2001 IEEE International Conference on Control Applications*, M. W. Spong, D. W. Repperger, and J. M. I. Zannatha, Eds. IEEE, 2001, pp. 761–766. [Online]. Available: http://www.ce.utwente.nl/rtweb/publications/2001/pdf-files/042R2001.pdf

[4] N.Xi, T.Tarn, and A.Bejczy, "Intellingent planning and control for multirobot coordination: An event-based approach," in *IEEE Transaction on Robotics and Automation*, vol. 12, no. 3, June 1996.

[5] S. Yildirim and T.Tunali, "A new methodology for dealing with uncertainty in robotic tasks," in *International Symposyum on Computer and Information Sciences*, Bornova, Turkey, 1999.

[6] E. Barrett, M.Realing, G.Barbieri, M.Fumagalli, and R.Carloni, "Mechatronic design of the sherpa robotic arm," University of Twente, 2016.

[7] B. Vanderborgh, A. Albu-Schaffer, A.Bicchi, E.Burdet, D.G.Caldwell, R.Carloni, M.G.Catalano, O.Eiberger, W.Friedl, G.Ganesh, M.Garabini, M.Grebenstein, G.Grioli, S.Haddanin, H.Hoppner, A.Jafari, M.Laffranchi, D.Lefeber, F.Petit, S.Stramigioli, N.G.Tsagarakis, M.V.Damme, R.V.Ham, L.C.Visser, and S.Wolf, "Variable impedance actuators: A review," in *Robotics and Autonomous Systems*, vol. 61, no. 12, 2013, pp. 1601–1614.

[8] E.Barrett, M.Fumagalli, and R.Carloni, "The sherpa gripper: Grasping of small-scale uavs," in *Proceedings of the IEEE International Symposium on Safety, Security and Rescue Robotics*, 2016.

[9] S. R. Buss, "Introduction to inverse kinematics with jacobian transpose, pseudoinverse and damped least squares methods," *IEEE Journal of Robotics and Automation*, vol. 17, no. 1-19, p. 16, 2004.

[10] B.Siciliano, L.Sciavicco, L.Villani, and G.Oriolo, *Robotics, Modelling, Planning and Control*. Springer, 2009.

[11] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *The International Journal of Robotics Research*, vol. 30, no. 7, pp. 846–894, 2011.

[12] N.Hogan, "Impedance control: An approach to manipulation," in *Journal of Dynamic Systems, Measurement and Control*, 1985, pp. 8–15.

[13] "Ros: an open-source robot operating system," http:www.ros.org/.

[14] "Moveit," http://moveit.ros.org/.

[15] "Urdf, unified robot description format," http://wiki.ros.org/urdf.

[16] "Ompl," http://ompl.kavrakilab.org/.

[17] S. M. LaValle, *Planning Algorithms*. Cambridge, U.K.: Cambridge University Press, 2006.

[18] L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, 1996.

[19] "Fcl, flexible collision library," "https://flexible-collision-library.github.io/".

[20] "Kdl, kinematics and dynamics lybrary," http://www.orocos.org/kdl.

[21] "Elmo motion control ltd." http:www.elmomc.com/products/whistle-digital-servo-drive-main.htm.

[22] "Arduino," http://www.arduino.cc.

[23] "Optitrack, natural point," http://www.optitrack.com.

4 Algorithms and Software Implementation

In the next sections, design choices and explanations of used algorithms are given. The specific implementation has been presented in the paper, but some details were omitted due to limited space.

4.1 Overview

The overview of the control architecture, which was previously presented in the paper in Section 3, is reported again in Figure 4.1. The software architecture is mainly composed of the Task Planner which handle the sequantial execution of the the Elementary Actions and the Path Planner which generates the appropriate set-points for the joint controllers. The Joint Controller communicates with the Robotic Arm through CAN bus while the sensors are interfaced through separate SPI bus.

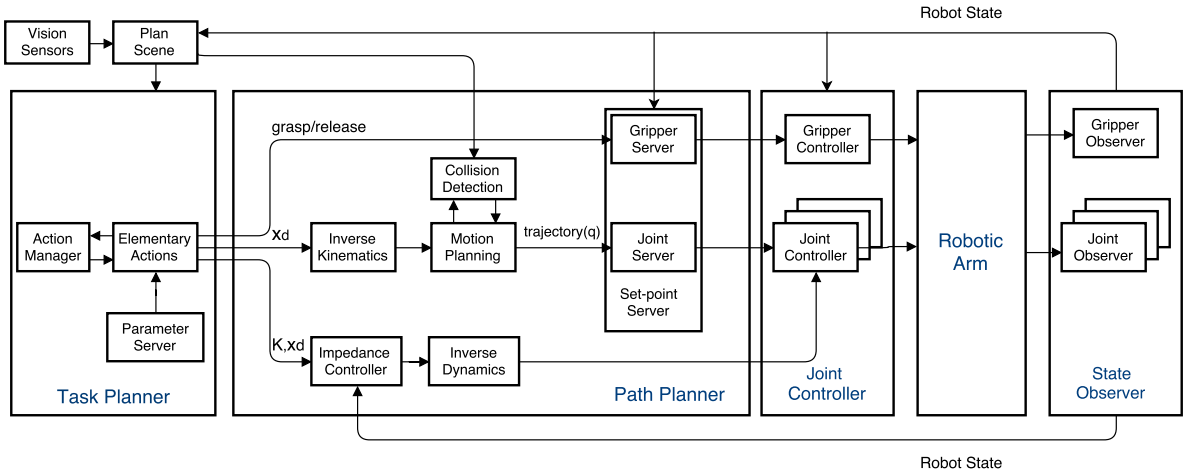


Figure 4.1: Control Architecture

The motion planning algorithm consist of a sampling-based algorithm, which finds collision-free path in the environment. The Set-point Server sends the computed trajectories to the Joint Controller for execution. In the Joint Controller a simple Proportional-Integral feedback controller is implemented.
The Impedance Controller is used to execute constrained Cartesian trajectories and to handle operations in which contact between the arm and the environment is necessary.
The State Obsterver output feedback on the robot's state by reporting measurements from the sensors on the arm. The feedback can be visualize through the ROS visualizator Rviz.
Inputs regarding the target locations and the control modes are handled by the Action Manager which manages the execution of the actions necessary to complete the planned task.
The geometrical representation of the environment is handled by the Plan Scene, while parameters such as constraints and robot model are stored and retrieved at run-time from the Parameter Server.

4.2 Motion Planning

In order to solve the complex motion planning problem, different approaches have been proposed (LaValle, 2006), such as search-based algorithms, potential functions, combinatorial algorithms or sampling-based methods. Out of the four methods, sampling-based planners are the most suitable to practically solve complex problems in high-dimensional spaces.
The fundamental idea of this class of algorithms is to approximate the connectivity of the configuration space with a graph structure. The configuration space is then analyzed and sampled in different ways and the selected samples between start and goal state are connected via a collision-free path. These methods are faster and computationally more efficient in high-dimensional planning problems than other motion planning algorithms since they operate within a finite set of configurations in the state space, instead of inspecting the entire continuous configuration space.

4.3 Sampling-Based Motion Planning algorithms

Nowadays, the most used sampling-based planners are: Probabilistic Roadmaps (PRM) (Kavraki et al., 1996) and Rapidly Exploring Random Trees (RRT) (LaValle, 2006).
Even though the idea of connecting points sampled randomly from the configuration space is common to both approaches, these two algorithms differ in the way they construct a graph connecting these samples.
The PRM is a multiple-query method that builds a roadmap which represents a set of collision-free trajectories, and answers the queries by reporting the one connecting start and goal state. This algorithm has been reported to perform well in high-dimensional configuration space (Kavraki et al., 1996). Moreover, the PRM algorithm is probabilistically complete, hence the probability of failure decays to zero exponentially with the number of samples used to construct the roadmap.
However, our planning problem does not require multiple queries and the environment is not known a priori. Moreover, computing a roadmap a priori may be computationally challenging or even infeasible.
The RRT is a single-query method that avoids the necessity to set the number of samples a priori. RRT has been showed to be probabilistically complete with an exponential rate of decay for the probability of failure (LaValle, 2006).
As introduced in Section 3, the motion plannig algorithm used in our architecture is the RRT* which is an optimal version of the RRT sampling-based algorithm.
This section introduces the theory regarding the basic RRT technique , together with its optimal variation. Collision detection approach will then be discussed.

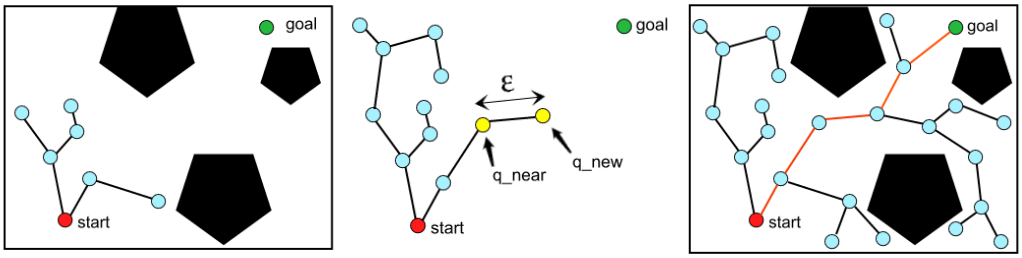
4.3.1 Rapidly-Exploring Random Trees

The RRT technique was introduced by LaValle in 1998. This is a non-deterministic, single-query planning algorithm which offers a method to quickly search non-convex, high-dimensional spaces with kinematic constraints.

Algorithm description

First we introduce some definitions. Let $C \in R^7$ be the configuration space of the manipulator and $W \in R^6$ the workspace in which the arm and the geometrical environment are described. C_{free} is defined as a subset of C which contains all the configurations in which the robot is not in collision. Furthermore, q_{start} and q_{goal} denote, respectively, the start and goal configurations of the robot. The path between start and goal state is denoted by $p(q_{start}, q_{goal})$.
The RRT algorithm generates the collision-free path between start and goal configuration by creating a tree structure of the free space. The tree outsets from the start configuration q_{start} of the robot, and is heuristically expanded by the planner towards the goal state (Figure 4.2a).

One of the advantages of this method is that the configuration space is sampled simultaneously with the construction of the configuration tree.
The basic RRT algorithm working principle is given in Figure 4.3.



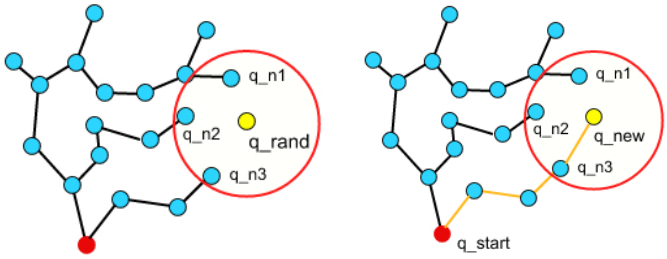
(a) **(b)** **(c)**

Figure 4.2: Working principle of the RRT algorithm

In each iteration a random sample q_{rand} is chosen from the configuration space C. At this point, an heuristic function is used to search for the nearest vertex q_{near} in the tree to the given sample q_{rand}.
The algorithm moves toward the goal configuration q_{goal} with some fixed incremental distance ϵ, starting from q_{near} as depicted in Figure 4.2b. At this point, if q_{near} is reachable in a single step, q_{rand} is directly added to the tree as q_{new} and its edge is checked for collision. If the proposed new edge does not lie in C_{free} then the configuration is rejected from the RRT.
The algorithm will continue performing this operation until the distance between the node q_{new} and q_{goal} is less then a defined distance d_{goal} or in case the maximum number of iterations I_{max} has been executed. In this case the search can be considered complete (Figure 4.2c) and the trajectory is published.

Optimal RRT (RRT*)

The RRT planner belongs to the family of non-optimal planners. There exists two paradigms to compute near-optimal solutions using RRT: construct trees considering some optimal criteria or improve the plans by applying post-optimization.
The RRT* (S.Karam and E.Frazzoli, 2011) belongs to the first category, and its implementation closely resembles that of the standard RRT described in the previous subsection. However, the optimal motion planning problem imposes the additional requirement that the resulting feasible path minimizes a given cost function $c(x)$, such as the length of the path.
The method starts by initializing an empty tree and a node corresponding to the initial state. Like the RRT, the RRT* incrementally builds the tree by sampling a random state q_{rand} from the free configuration space C_{free}. At this point, the RRT* rather than selecting the nearest node q_{near} as parent, will consider all the nodes in a defined neighboorhood and will evaluate the cost of choosing each as the parent as depicted in Figure 4.3a and Figure 4.3b.
This process calculates the total cost as the additive combination of the cost associated with reaching the potential new parent nodes q_{n1}, q_{n2} or q_{n3} and the cost of the entire trajectory to q_{new}. Among those, the node that yields the lowest cost will be selected and will be added to the tree.



(a) **(b)**

Figure 4.3: Working principle of the RRT* algorithm

Collision Detection

Collision detection is a critical component of sampling-based motion planning techniques. Once the algorithm computes the samples, it has to be determined wheter they lie in the free configuration space or not. The collision can be identified either as a self-collision or as an environmental collision.
In this project the flexible collision library (FLC[1]) was used. This algorithm returns a list of contact data that identify collision points for a new random sample chosen by the motion planning algorithm.
In this method, dynamic axis aligned bounding boxes are used. This means that volumes are represented trough dynamic boxes which are recomputed each time the robot's state or the environment changes.
Collision checking was tuned by modification of the Allowed Collision Matrix in order to enable collisions with objects when contact is required by the task.

[1] Flexible Collision Library - https://github.com/flexible-collision-library/fcl

4.4 Kinematic Solver

As introduced in Section 3, control methods are applied to solve inverse kinematics problems. The damped least square method with the singular value decomposition was used in this project. Our choice was based on the robustness of the method in handling joint limits, convergence issues and singularities.
However, other solutions has been explored during this assignment. One of them was the Jacobian Transpose algorithm, which is very interesting as it avoids matrix inversion. This method was first introduced for inverse kinematics by (Wolovich and H.Elliot, 1984).

Jacobian Transpose Method

The control scheme for the Jacobian Transpose method is showed in Figure 4.4.

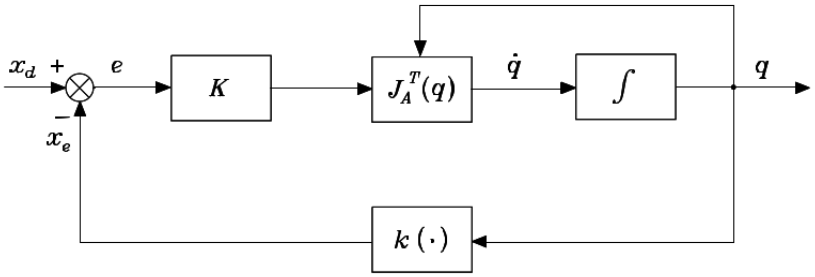


Figure 4.4: Jacobian transpose control scheme (B.Siciliano et al., 2009)

This method uses the transpose Jacobian instead of its inverse. This result in very fast computational time and avoidance of unstable behaviour in the neighborhoods of singularities.
With reference to the scheme depicted in Figure 4.4 the following control rule is defined:

$$\Delta q = K J_a^T e \tag{4.1}$$

where J_a^T is now the transpose of the analytic jacobian of the manipulator.
In order to ensure error convergence to zero, when choosing K, literature (B.Siciliano et al., 2009)(Buss, 2009) recommends to take the value that makes the norm of the change in the task space equal to the norm of the error.
As described in (Buss, 2009), this results in:

$$K = \frac{< JJ^T e, e >}{< JJ^T e, JJ^T e >} \tag{4.2}$$

As already introduced, this solution is computationally very fast, altought it may require more steps than other methods since the Jacobian values are smaller. Moreover, by not being a least-squares solution it can derive in chattering.
The algorithm was first implemented in Matlab to study its feasability.
The simulation was run for a 7-DOF manipulator executing reach motion with joint limits set at 1.57 *rad*. The results are reported in Figure 4.5.

It can be seen that by using this method, joint limits are easily surpassed. This is due to the fact that the algorithm works under the crucial assumption that any commanded joint velocity is achievable (Wolovich and H.Elliot, 1984).
This assumption is broken in presence of unfeasible states in the configuration space such as joint limits and environmental obstacles.
The integration of this algorithm in our architecture resulted in very poor performances. Even

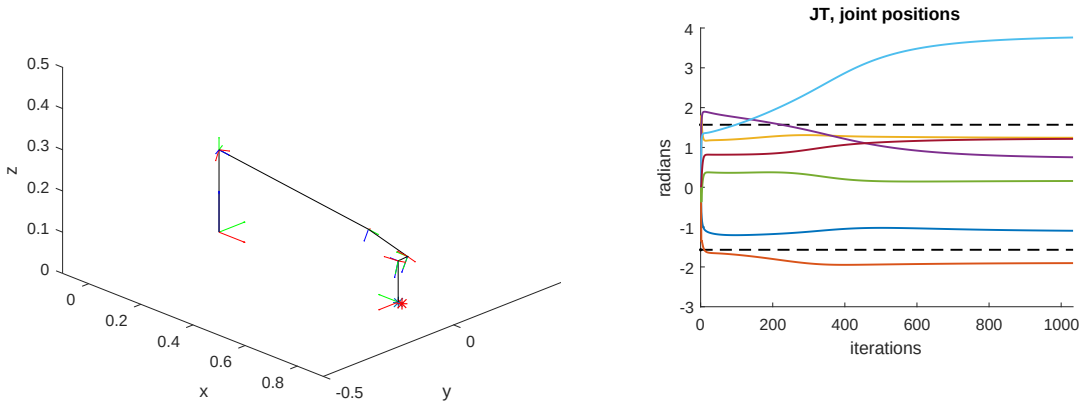


Figure 4.5: Behaviour of the Jacobian Transpose method when the objective vulnerate joint limits

when a solution to the inverse kinematic problem was found, it lied outside the free configuration space C_{free}, causing the motion planning algorithm to fail in finding a valid path between start and goal state.

Moving towards the use of Jacobian Transpose algorithm

The Jacobian transpose algorithm results very attractive from a computational point of view. Indeed, it requires only forward kinematics to be computed.

However, as already described, the problem of introducing joint limits and collision avoidance has to be solved.

One solution to this problem could be to assign the maximum or minimum value for the joint if the limit is surpassed. Nevertheless this will not work well, as blocking a joint will result in blocking other joints movements.

Another solution is proposed in (L.Sciavicco, 1988), which consists in incorporating constraints derived from joint limits and obstacles, on the condition of properly enlarging the task space vector. It is possible to define a set of variables that describe the configuration of the manipulator with respect to the obstacle and the joint limits. In this way, it will be possible to describe those variables in terms of joint variables obtaining augmented direct kinematics:

$$y = f'(q) \tag{4.3}$$

where y is an $((m+v)x1)$ vector completely defined in task space, with $0 \leq v \leq n-m$. Once the forward kinematic is solved, it will also be possible to solve the inverse kinematic problem under both constraints.

Hence, a solution can be found by enlarging the task space and including obstacle avoidance contraints and joint limits. Anyway this approach will present some difficulties regarding activation and deactivation of the constraints (L.Sciavicco, 1988).

4.5 Software Implementation

4.5.1 Executing Elementary Actions

The main goal of this project is to design and implement a control architecture that autonomously manages the execution of different elementary actions to achieve the finalization of a pre-defined mission. Figure 4.6 shows the interaction of the designed classes in a normal flow of executing elementary actions clarifying the behaviour of the Task Planner.
In the end, it is possible to see two outcome states, namely *Succeded* and *Aborted*.

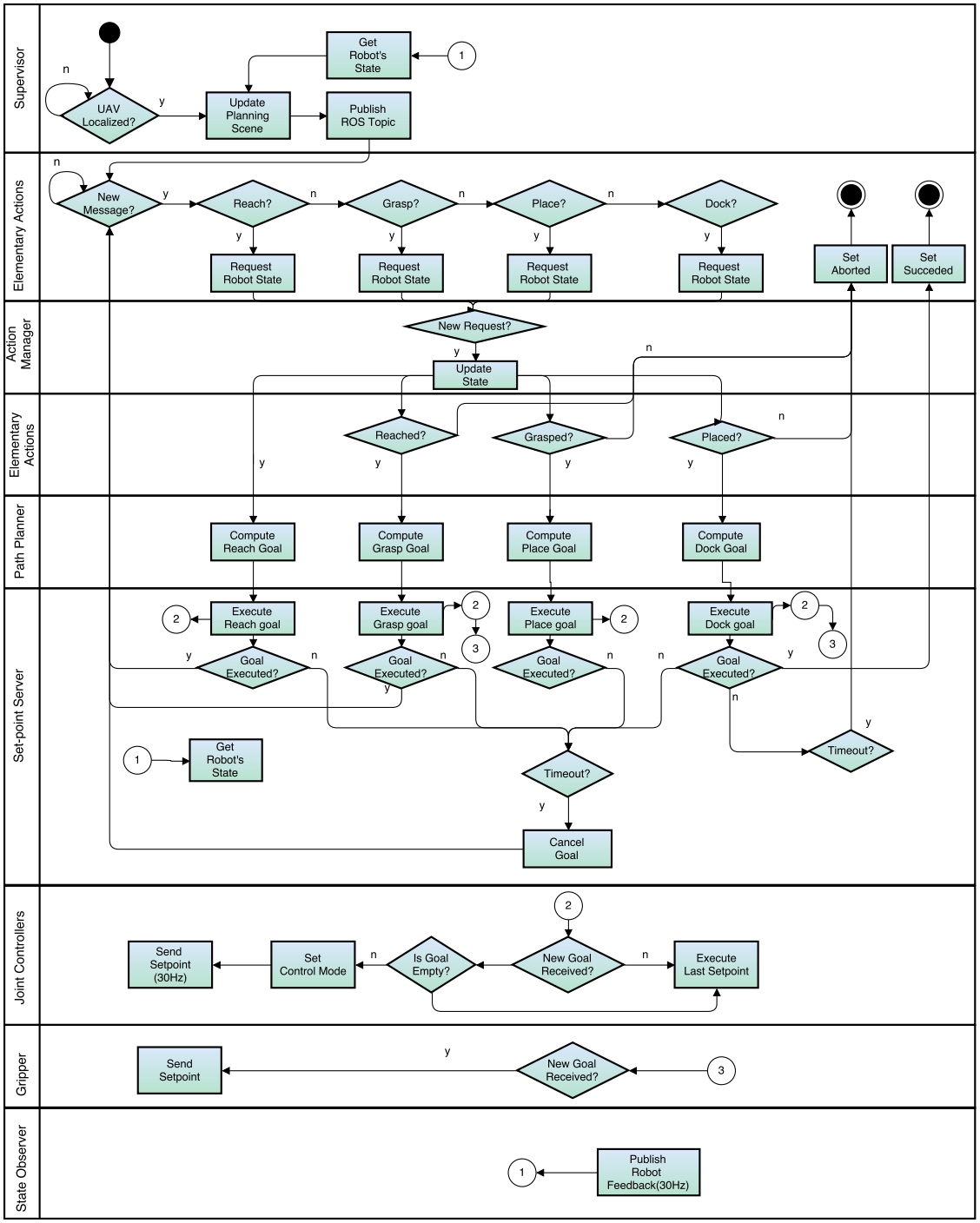
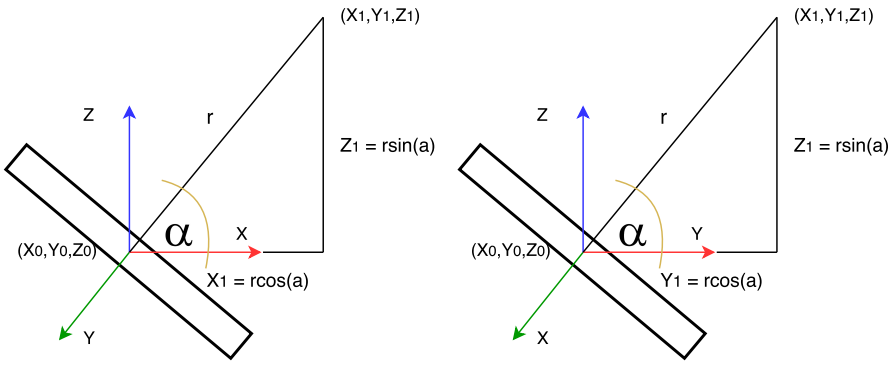


Figure 4.6: Diagram showing execution of Elementary Actions

Following the diagram flow, the starting sequence depends on whether the UAV has been localized or not by the vision sensors. In our case, the UAV is assumed to be localized every time, since no vision system has been implemented yet.
Once the UAV is detected in the arm's workspace the Plan Scene is updated and the new world representation is published to the corresponding topic `/Planning_Scene`. At this point, the elementary actions are initialized. The first action is always *Reach* which sends a request to the Action Manager to retrieve the current state of the robot. Once the state is updated the Elementary Action can be executed and the location of the UAV is sent to the blocks of the Path Planner to generate the task execution orders for the Joint Controller.
The Set-point Server, controls the execution by retrieving feedback from the State Observer. If the execution failed, the goal will be canceled. The subsequent elementary action will be initialized by requesting the current state of the Plan Scene and verifying whether the previous one has been executed. If this is not the case, the mission will be stopped. *Grasp* and *Dock* actions include the commands *Grasp* and *Release* to the gripper.
The mission will be considered successfull if all the elementary actions are executed.

4.5.2 Generating a correct reach position

As introduced in Section 3, before grasping the object, the arm's end-effector has to be placed above the UAV interface. The easiest case scenario is obtained when the interface is parallel to the ground and it is possible to compute a simple translation along z from the object pose. However, due to irregularities of the ground the UAV interface may be tilted.
Hence, once the homogenous transformation that describes the object position with respect to the base frame is known, another translation has to be computed. This translation depends on the orientation of the UAV interface, since it has to be computed along the z-axis of the object frame.
Figure 4.7a and 4.7b shows the transformation that is computed in defining the desired end-effector pose depending on the UAV's orientation. With reference to the figures the goal position for the end-effector is defined as (x_1, y_1, z_1). Parameter $r = 15cm$ represents the distance between the end-effector and the object's center of rotation.



(a) Pitch **(b)** Roll

Figure 4.7: Computation of end-effector position for rotations of the UAV interface

4.5.3 Software Implementation: ROS

As already mentioned in Section 3 the Robotic Operating System (ROS) framework has been used to develop the software. In this framework, different scripts (threads) run along-side and communicate over internal UDP-connections. The version of the framework used for this project is the 8th distribution Indigo Igloo and the operating system that the system is running on is Ubuntu 14.04.
The general software implementation of Figure 4.1 is showed in Figure 4.8. The nodes corresponds to the blocks described in the control architecture analysis and their relation can be seen in the colored legend.
The threads that costitutes the Task Planner are the *Elementary_Actions* nodes and the *Action_Manager_Server* node.
The Path Planner is composed by the following nodes:

- the *move_group* node which handles execution of inverse kinematics and motion planning queries.
- the *Joint_Trajectory_Action_Server* node and the *Gripper_Action_Server* node which compose the Set-point Server
- the *Impedance_Controller_Server* node, and the *Impedance_Controller* node which constitutes the Impedance Controller package.

The *Controller* node contains the implementation of the Joint Controller and the State Observer and communicates to the hardware interfaces through the *Spi_Interface_node*, the *Can_Interface_node* and the *Serial_Interface_node*.

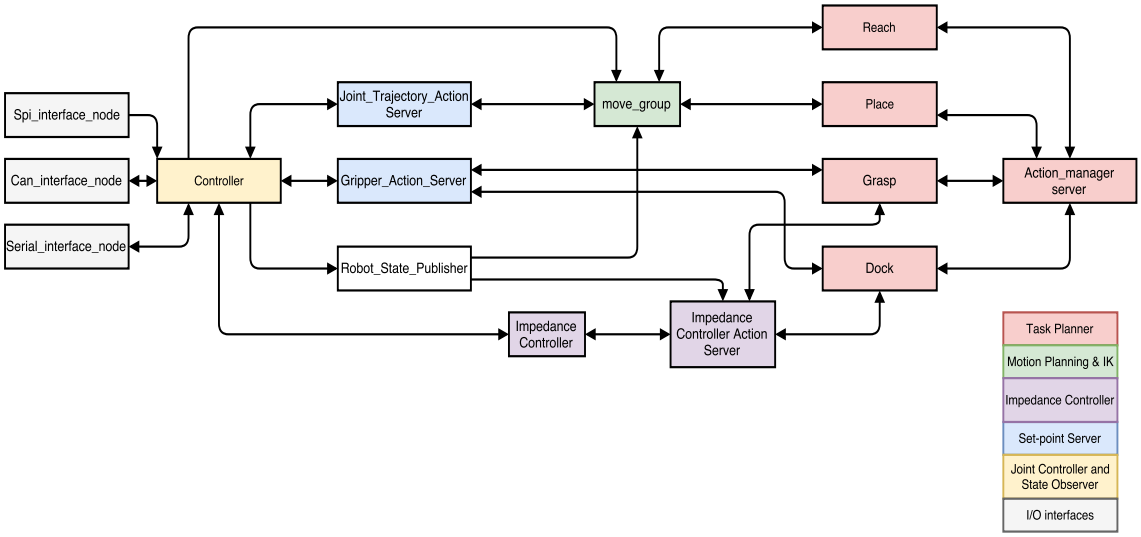


Figure 4.8: ROS node diagram

At the current state of implementation the Impedance Controller package is not used on the real arm. Hence, all the paths necessary to execute the Elementary Actions are generated within the *move_group* node. Further details on the nodes and the communications between them are presented in the next subsections.

4.5.4 MoveIt!

In this project, tasks that involves motion planning are developed within the MoveIt![2] framework. Here, a more detailed overview about MoveIt! is given.
Various parts of MoveIt! are implemented as plugins which means they are replaceable by other components that provide the same interfaces. This makes MoveIt! highly customizable.
The motion planning and kinematic solver used in this project are integrated as plugins in the MoveIt! framework.
The central part of the framework is the *move_group* node, which manages all the plugin present in the Moveit! environment. This node also mantains the Plan Scene, which is created by retrieving informations from the sensors placed on the robot.
The *move_group* node provides a user interface, consisting of various ROS topics and services that allows to access its functionalities.
MoveIt! also provides a plugin for Rviz visualization tool which can be used to do planning requests base on a graphical interface or receive feedback on the plan computed during execution.

Robot Interface

The *move_group* node communicates with the robot through ROS topics and ROS actions.
It requires to listen to the */joint_state* topic to retrieve the current state of the robot. It also monitors informations about the homogenous matrices that describe the position of a robot's link with respect to the reference frame using the ROS TF library[3]. Those informations are provided by the *robot_state_publisher* node.
The *move_group* node communicates with the Joint Controller through the *Joint_Trajectory_Action_Server*.

More details on the implementation and description on how to install and connect MoveIt! with the SHERPA Arm are described in Appendix A.

4.5.5 Set-point server messages

As already mentioned in Section 3, the Set-point Server handles the execution of the trajectory generated by the Path Planner. This was achieved by designing a server-client model using the *actiolib*[4] stack in ROS. In order to communicate with the servers, the clients have to send a message that contains the goal that has to be executed. In general, client and server communicate by exchanging messages as showed in Figure 4.9.

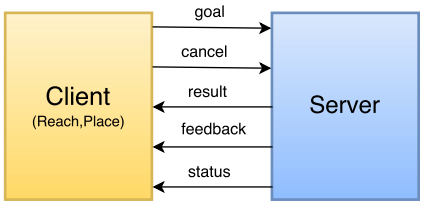


Figure 4.9: Server-Client model with messages

- *Goal*: This topic is used to send an ActionGoal message to the controller. It's a custom message that contains the relevant fields for the server to execute the action. In our case, this message describes a time parametrized trajectory that has to be executed along with

[2] ROS.org - MoveIt! http://moveit.ros.org/
[3] ROS.org - TF library http://wiki.ros.org/tf
[4] ROS.org - actionlib http://wiki.ros.org/actionlib

allowed tolerance values for final joint positions and execution time. Those tolerances specify how precise the controller has to stick to the position, velocity, acceleration and time constraints defined for the trajectory in order to consider the execution to be successful.

- *Feedback*: The controller uses this topic to continuously provide feedback about the current execution status. The feedback message contains information about the trajectory point the controller is currently processing.This result in preventing the execution of another goal while one is currently being executed.
- *Status*: Provides the client with status information about the goal that is currently being tracked by the server. The client uses this information to know if the goal is still active, aborted or pending.
- *Result*: After finishing the trajectory execution, the controller publishes the result to this topic. If the final joint positions lie within the specified tolerance values and the time constraints were met, the trajectory execution is considered to be successful. The message contains a boolean variable *status* that returns true when the terminal state of the state is succeeded, and false when it is terminated in any other state.

The messages used to define the goals for the Joints and the Gripper are structured in the same way. Details on the topics are reported in the Appendix A.

4.5.6 Graphical User Interface

Graphical feedback is provided to the user through the ROS visualization tool (RViz). This is a 3D visualizer for displaying sensor data and state information of the robot's model.
Two models have been included in the visualizator. One of them displays the joint commands generated by the Path Planner coming over a ROS topic, while the second one is used for comparison and displays the actual state of the robot by subscribing position measurements sent by the encoders over ROS.

4.5.7 Impedance Controller implementation

In order to execute actions that require interaction with an unknown environment the Action Manager will send commands to the Impedance Controller which will translate them into torque set-points for the Joint Controller. Figure 4.10 shows the interaction of the designed classes to achieve this goal.

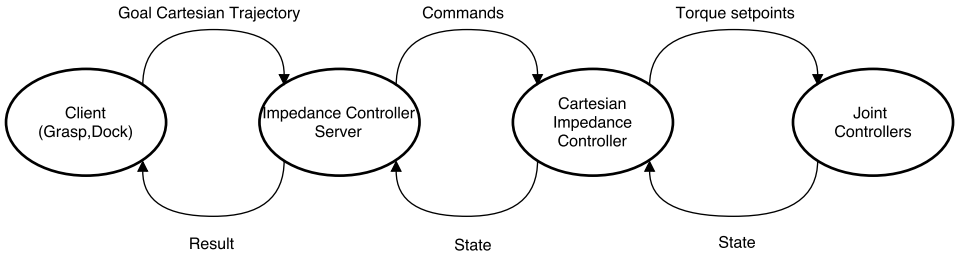


Figure 4.10: Impedance Controller design

Client-Server

As well as for the Set-point Server the Cartesian trajectory execution is handled by a client and a server, implemented through the use of the *actionlib* stack from ROS.
Once *Reach* or *Place* have been executed, the Action Manager initializes respectively *Grasp* and *Dock*, which sends the desired Cartesian Trajectories to the Impedance Controller Server. The

first point of the trajectory has been set as the current position of the end-effector with maximum stiffness, in order to avoid leaps in the arm movements during control switching.
The Impedance Controller Server is in charge of sending the commands to the Impedance Controller while monitoring execution. The desired trajectories are defined in end-effector frame and directly translated by the server to the arm base frame. The detailed scheme of the ROS messages exchanged between client and server is the same as the one depicted in Figure 10, although in this case the goal is defined in a different way.
In ROS, the goal is defined as a custom message, hence a new structure was created. The description of the goal's field is as follows:

- *Pose*: this defines the pose to be achieved by the end-effector of the arm.
- *Stiffness*: this defines the Cartesian stiffness to be achieved in each Cartesian dimension.
- *Time from start*: this field stores the time from start of each point of the trajectory.

In order to enforce the constraints on the trajectory execution some parameters has been introduced.

- *constraint/goal/time*: this represents the amount of time that the controller is allowed to be late in executing the goal. If the set time is passed and the controller still hasn't achieved the final position, then the goal is aborted.
- *contraints/goal/pose*: this defines the maximum final error in pose for the trajectory goal to be considered successfull. Negative numbers indicates that there are no constraints.
- *constraint/goal/effort*: this parameter defines the maximum error allowed in the squared difference between the joint efforts. Negative numbers indicates that no constraint is set.

In initialization, the Impedance Controller Server will retrieve those parameters from the Parameter Server.

Controller

As already introduced in Section 3, the controller is an open-loop Jacobian-transpose force controller.
The main function of the Impedance Controller class is to convert at every point on the trajectory the desired stiffness into a Cartesian force vector and ultimately into a joint torque vector using the transpose of the arm Jacobian matrix computed at its actual location.
Since there is no adequate feedback on the arm to determine the applied force, no guarantee is made on the magnitude of the applied force.
In initialization, the Impedance Controller retrieves the arm's kinematics from the Parameter Server creating a serial chain, which is used to compute forward and inverse kinematic. The Impedance Controller Server sends a vector containing the entire Cartesian Trajectory. In order to translate the given set-points into joint commands at the appropriate time defined in the goal, they are are stored in a dynamic queue. The sampling rate of the controller is used to analyse the time passed between the last and next set-point. Once this value corresponds with the one set in the goal, the set-point is removed from the trajectory reducing its size. When the trajectory is empty the last setpoint is hold and sent to the controller each sampling update.
During execution, the State Observer provides the current state of the arm to the Impedance Controller which uses it to calculate forward kinematics. The state is then provided to the Impedance Controller Server, which compares it with the imposed constraints. Based on this comparison, it will keep executing the goal or cancel it and reporting the result to the Action Manager.

The instructions necessary to run the Impedance Controller package has been properly documented in Appendix C.

4.5.8 Testing Impedance Controller on a simulated 3-link arm

Since the control architecture is still missing a full implementation of the state-observer the impedance controller was tested in simulation on a different and simpler arm. However, the code has been written and documented to be immediately adaptable once the proper dynamic compensations on the arm will be implemented.
In Figure 4.11, the Gazebo environment with the simulated arm for testing is depicted.

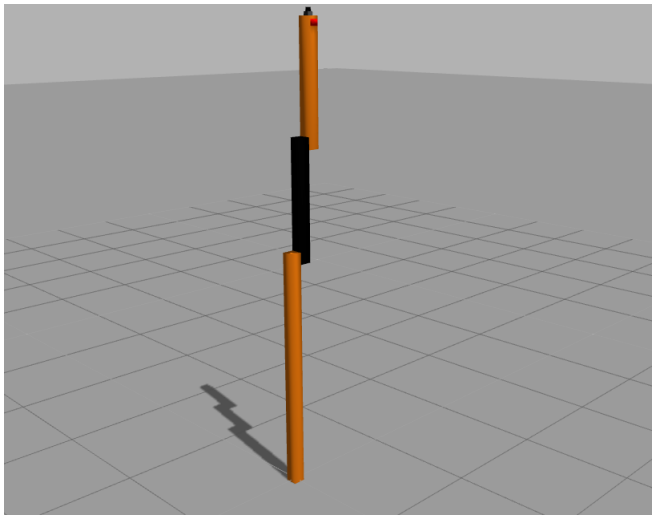


Figure 4.11: Simulated 3-link arm in Gazebo environment

The total lenght of the arm is 4 meters ($link1 = 2m$, $link2 = 1m$, $link3 = 1m$). The inertias are calculated with respect to the center of mass of each link.
In the experiment, the desired position is set at $x = 0.5m$,$y = 0.2m$,$z = 1m$ with a desired cartesian stiffness of $1N/m$ in the x-direction, $10N/m$ in the y-direction and $10N/m$ in the z-direction. In this arm, the end-effector link has only 1-DOF, hence only variation on the x-axis will be evaluated.
The desired position is defined slighty inside a box placed in front of the 3 link arm in the simulation environment as depicted in Figure 4.12.

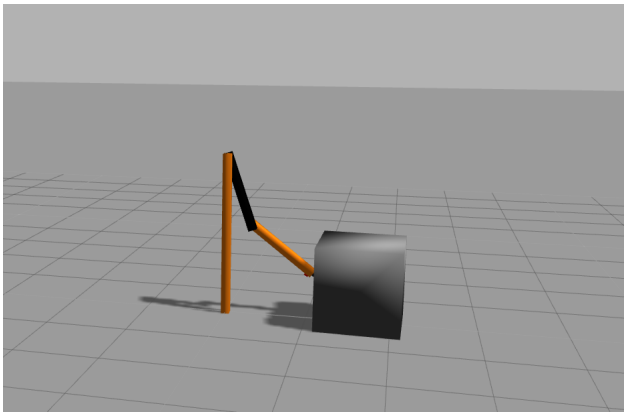


Figure 4.12: Simulated 3-link arm in Gazebo environment in contact with an unknown object

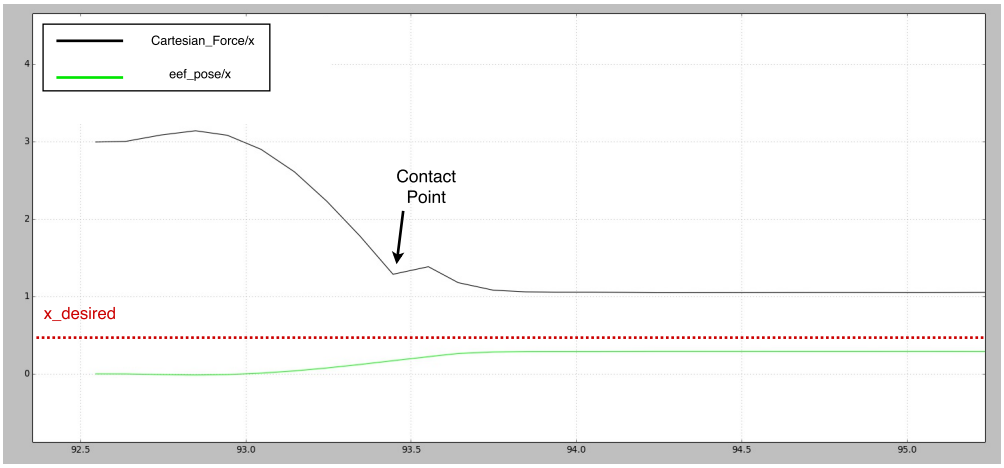


Figure 4.13: Plot of cartesian force and end-effector position in presence of obstacle

The plot of Figure 4.13 shows the response of the system while approaching and contacting the environment.
It is possible to observe that the end-effector position never reaches the desired position, which stabilizes around $0.4m$ once the contact with the box has happened.
The Cartesian force in the x-direction starts from a value of 3N and reduces the closer we get to the desired position. We can notice that at the contact point ($x = 0.4m$) the Cartesian force increases. At this point, the Impedance Controller reacts to redefine the position trajectory that limits the steady-state force to a value of approximately 1N.

5 Conclusions and Recommendations

The aim of this master thesis research was to increase the autonomy of the 7-DOF light weight compliant manipulator of the SHERPA project. This report introduced a control architecture which allows to execute the task of grasping and docking a UAV while monitoring execution.

By means of layered control approach, several levels of autonomy for the SHERPA arm were defined. The proposed method is based on the supply of feedback information, about the world model, to the task planner which can monitor the feasability of a given subtask. In case an action is not successfully executed, the mission is interrupted. In addition, the path planner ensures trajectory tracking through the use of the set-point server.

The previously described approach has been implemented and tested for the validation of the proposed architecture and algorithms. Experiments results show reliability of the system in executing the given task even with the presence of obstacles in the environment. However, significant deviation between the desired and measured trajectory was observed, reducing the success rate of effective grasping and docking of the UAV.

In order to improve the system's performance, it is recommended to use a proximity sensor mounted on the end-effector. This allows, precise detection of contact between the gripper and the UAV, granting a command to be sent to the gripper with higher certainty. Moreover, once vision sensors are integrated in the system, a more accurate position of the UAV can be provided to the software, whilst currently, the drone is placed manually in the desired position. It is also recommended to further develop the state observer by integrating a proper dynamic model of the arm. This will allow the integration of the implemented impedance controller and the improvement of the overall performance of the system. Also, the calibration technique of the sensor should be improved, as currently, existing calibration errors compromise the accuracy of the executed trajectory.

The proposed method was implemented and tested for the specific scenario described in Section 2. At the current state of development, in the event of failure of an elementary action during execution, the mission is ceased. However, the designed software framework lays groundwork for future development, focusing on the implementation of more complex behaviours for the task planner such as the replanning of given subtasks in the case of failure, or the occurence of unexpected events.

Future work should include the integration of the arm and the ground rover's softwares in order to conduct more complex and realistic cooperative missions.

Finally, the devised scenario for the scope of this thesis is the one in which the multirotor lands on the ground in proximity of the rover and is then docked by the robotic arm. In this scenario, the type of surface on which the drone lands is important; a smooth flat surface is preferable. However, this scenario is very unlikely, and a rocky environment is to be expected in alpine areas. The drone could then be lost or damaged when attempting to land.
This thesis introduces and demonstrates the plausibility and feasibility of such operation. In the future, other scenarios could be inspected such as docking the multirotor while is still flying in close proximity to the rover.

A Appendix

A.1 URDF model

The Unified Robot Description Format (URDF) is a markup language, designed to describe robots. The description happens in text files, in a special XML format. The most important elements in the XML specification are:

- *Link*: Describes the properties of a specific robot link. Each link must have a unique name. The visual, inertial and collision details are configured in the corresponding subtags of the link element. The visual part as well as the collision model can either be composed from primitive shapes or from mesh files.

- *Joint*: Describes the properties of a joint. A joint is a flexible connection between two links, having exactly one parent and one child link. Each joint states a new reference frame for its child link and it is positioned relative to its parent frame. The joint actuates its child link relative to its parent link along the joint axis. There are different types of joints available such as fixed and revolute.
  A fixed joint states a rigid connection between parent and child link. A revolute joint is a rotational joint with one degree of freedom. Details like angular joint limits, axis orientation and the dynamic properties of the joint motors can be configured in the corresponding subtags of the joint element.

Those elements are used to form the URDF graph that exactly describes the kinematic chain of the robot components and their placement relative to each other.

The URDF file relative to the SHERPA arm can be found in the folder `SHERPA_ARM_V1.52_rover` together with the meshes generated in SolidWorks.

A.2 MoveIt installation

A.2.1 Creating MoveIt configuration package

After designing the URDF description of our robot setup it was necessary to create a MoveIt configuration package. This was done, using the MoveIt Setup Assistant which is part of the MoveIt distribution.
This software tool provides a graphical user interface that allows to con figure and generate a ROS package that contains all necessary MoveIt configuration files based on an existing URDF model. This section explains the steps that had to be performed and the resulting configuration package.
The Setup Assistant was launched using the following command line statement:

```
roslaunch moveit_setup_assistant setup_assistant.launch
```

This command brings up the Setup Assistant which asks for the file path to the previously created URDF description. After pointing the Setup Assistant to the correct file location, we had to perform the following configuration steps:

- *Computing the self collision matrix*: The self collision matrix consists of pairs of robot links that can safely be excluded from collision checking. Adjacent links of the robot arm for example are in permanent collision. Collisions between other links can never happen because they are simply too far apart. The Setup Assistant can be triggered to compute

this self collision matrix automatically by testing a large number of different robot configurations while tracking for link pairs that are hardly always in collision and pairs that are never in collision. The number of sample configurations to check can be adjusted. We selected a medium density of 50.000 sample configurations to find colliding link pairs. Excluding a large number of link pairs raises performance during motion planning because collision checking is an expensive process. The resulting self collision matrix can be adjusted manually if necessary. The image in figure below shows a screenshot of this configuration step.

- *Defining the planning groups*: MoveIt requires to define so called planning groups for the robot setup. A planning group is a group of links and joints within the model that can be seen as a logical component, e.g. a gripper or a robot arm. Each planning group consists of a unique name and a list of robot links and joints that are part of that group. Additionally can be specified, which IK solver should be used for the planning group.Each planning request in MoveIt is done against one of those defined planning groups. We defined planning groups for arm, gripper and rover. However, motion planning is performed only for the group arm.

- *Defining the end effectors*: Each planning request in MoveIt requires to specify the end effector of the selected planning group. Those end effectors also had to be defined during the setup process. An end effector definition in MoveIt consists of a unique name, the underlying planning group (in our case end effector), the parent planning group (in our case arm) and the name of the last link in the kinematic chain of the parent group (in our case End adaptor).

- *Completing the Setup Assistant*: The last step in the setup process was to generate a MoveIt configuration package based on the previously explained configuration steps. Therefore the Setup Assistant required to specify the desired package name which we set to `sherpa_moveit_config`. After triggering the package generation on the Configuration Files page of the Setup Assistant, the resulting ROS package was created at the specified save location.

The package that was generated during this setup process contains all the configuration and launch files that are necessary to make planning requests for our robot setup though it is not yet connected to the hardware. The configuration can be tested by running it in demo mode. This mode allows to plan and execute trajectories without being connected to the robot hardware. The demo mode is launched with the following command line statement: `roslaunch sherpa_moveit_config demo.launch` This command runs a `move_group` node using the previously created configuration and starts an instance of RViz with the motion planning plugin. There it is possible to switch between planning groups, set start and target configurations, do planning requests and visualize the resulting trajectories based on a graphical user interface.

A.2.2 Launch files and Configuration files

The configuration files withing the config package are described here:

- `controllers.yaml`: The content of this file specifies the available controllers for the (simulated or real) hardware. The file is initially empty and has to be populated manually. This was done after implementing the required controller interface, which is described in section 3.6.

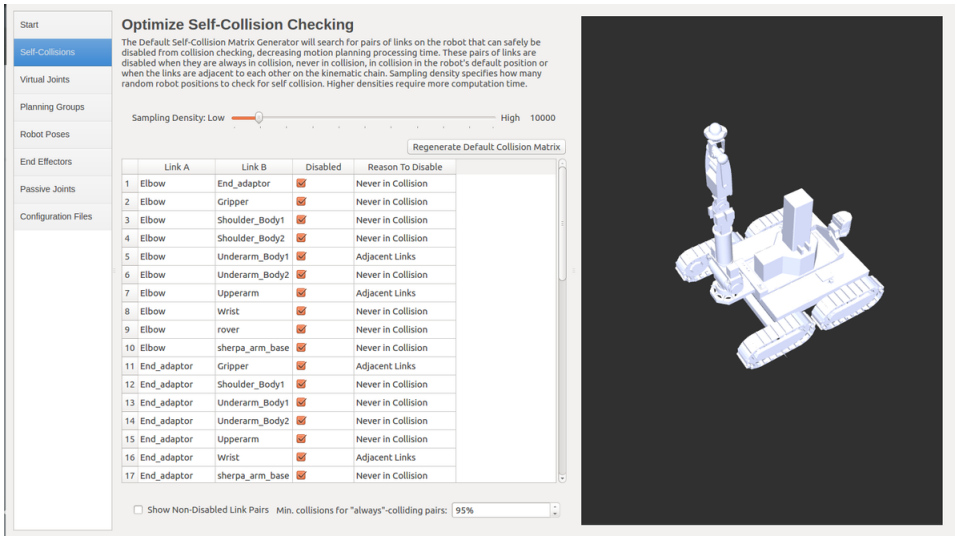


Figure A.1: generating collision matrix with setup assistant

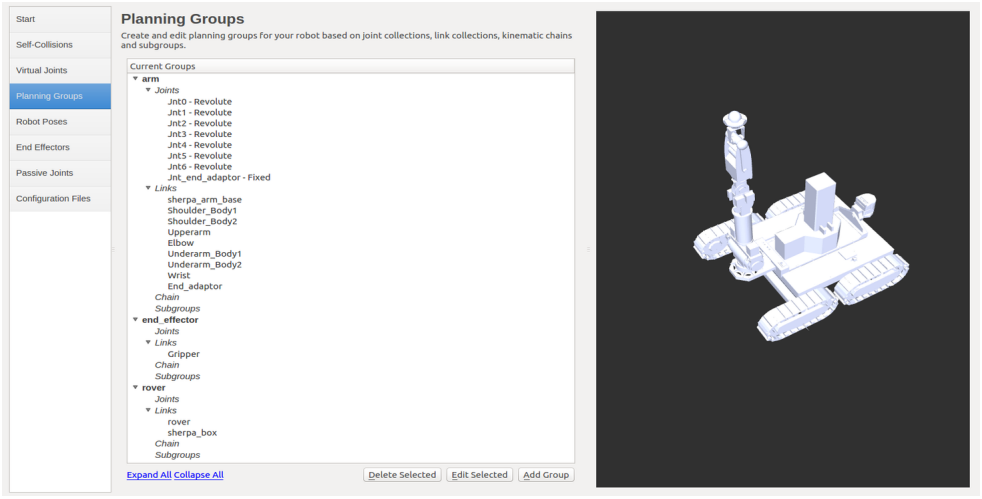


Figure A.2: planning groups setup assistant

- `fake_controllers.yaml`: This file is populated by the Setup Assistant and contains the definitions for the fake controllers. The fake controllers simulate a connection to the hardware by providing the required ROS interface.
- `SHERPA_ARM_V.1.52.srdf`: This file contains all the semantic information about the robot that was configured during the setup process. The content is specified using the SRDF format. It holds the definitions of the configured planning groups, end effectors and the generated self collision matrix entries.
- `joint_limits.yaml`: This file holds the velocity and acceleration limits of all joints contained in the robot description. The Setup Assistant creates the initial values based on the URDF model but this configuration file allows to specify other limits if necessary. MoveIt uses the limits that are configured within this file, not those configured in the URDF description.
- `kinematics.yaml`: This file contains the definition of the IK solvers that should be utilized for the robot arms. This file is currently not used since the kinematic plugin was modified internally.

- `ompl_planning.yaml`: This file defines the allowed planning algorithms for each planning group. It holds OMPL specific configuration parameters, the most relevant of which are:
  - Range: represents the maximum length of a motion to be added in the tree of motions
  - Goal bias: In the process of randomly selecting states in the state space to attempt to go towards, the algorithm may in fact choose the actual goal state, if it knows it, with some probability. This probability is a real number between 0.0 and 1.0; its value should usually be around 0.05 and should not be too large.
- `joint_trajectory_action.yaml`: This file holds the Joint Server configuration. It specifies goal constraints for the joints and for the goal pose.
  Note: If, while executing a trajectory, the following error appears on terminal:

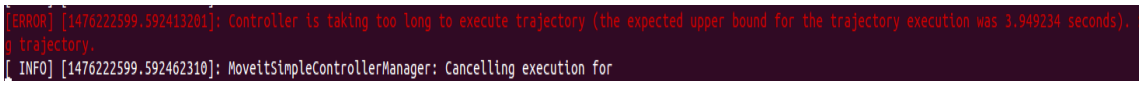


Figure A.3: Feedback screen error message

  increase the goal tolerance's value in this parameter file. This parameter defines the maximum error deviation between the current robot's state and the desired trajectory.

The launch files within the configuration package are responsible for uploading the configuration parameters to the parameter server and start the `move_group` node. A launch file was written to run all the necessary nodes to set-up MoveIt. This can be launched with the following command:

```
roslaunch sherpa_moveit_config moveit_planning_execution.launch
```

This launch file will launch the `move_group` node, the joint trajectory action server, the Rviz visualization, the `robot_state_publisher` the gripper action server and the impedance controller action server. It offers also the possibilty to change the environment from simulation to real hardware. By setting `sim = true` a controller simulator part of the `industrial_simulator_package` will be run. In this way it will be possible to execute the docking procedure in simulation.

A.2.3 Connecting MoveIt to real hardware

MoveIt interfaces with the robot through the FollowJointTrajectory action, which has been integrated in `Joint_Trajectory_Action` node.
In order to enable communication between MoveIt and the Sherpa Arm the Joint Controller has to subscribe the `joint_path_command` topic containing the `trajectory_msgs/JointTrajectory` message.
On the other hand, the State Observer is required to publish the `feedback_states` topic containing the (`control_msgs/FollowJointTrajectoryAction`) message.

B Appendix: Gazebo Model

In order to run Gazebo the following packages are needed:

- *sherpa_arm_control*: It contains configuration files in which is possible to specify simulated controllers and a launch file to call them.
- *sherpa_arm_description*: It contains the description of the arm in URDF format with the addition of Gazebo elements for visualization and simulation.
- *sherpa_arm_gazebo*: It contains the Gazebo world and a launch file that loads the robot description and controllers.

In order to visualize the URDF model in Gazebo the file `sherpa_arm.gazebo` has been created to add <gazebo> elements for each link and joint. This elements allow to: visualize links in gazebo, set damping dynamics for the joints and add actuator control plugins for the joints.
The URDF has been converted into Xacro file which allows to simplify the model and include different files. In the URDF model transmission values have been added. The file `materials.xacro` is used to specify colors and materials of the arm.

Launch Gazebo Model To start the Gazebo model the following command has to be run from terminal:

```
→roslaunch sherpa_arm_gazebo worlds.launch
```

When launching this node, you should be able to visualize the following image on your screen:

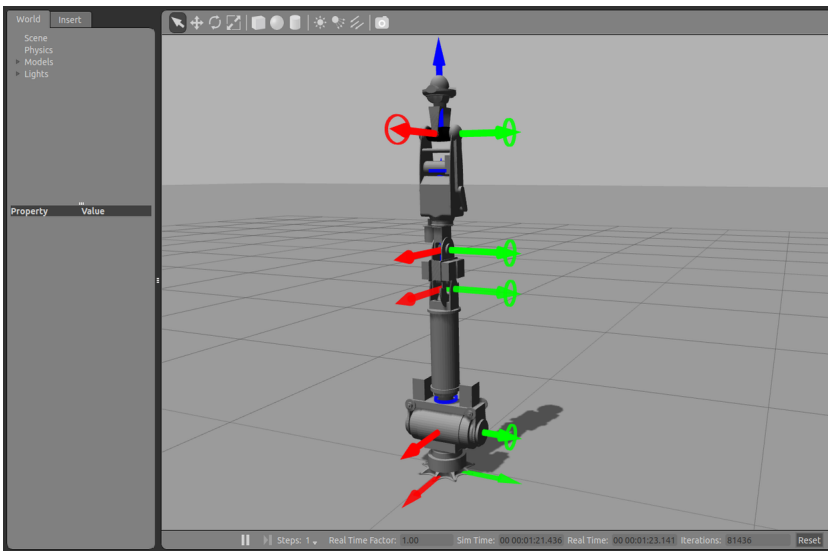


Figure B.1: Sherpa Arm in Gazebo environment

This model has been set up and prepared, however some problems still hold relative to the mimic joint present in the elbow. Since Gazebo does not support mimic joint the model of the model of the arm result inaccurate, hence difficult to run proper simulations. The best way to solve this issue should be by writing a proper Gazebo plugin.

Connect ROS and Gazebo

When running the Gazebo model the following topics will be set-up running:

Figure B.2: Sherpa Arm in Gazebo environment

In order to run a simulation, the `/sherpa_arm/joint_states` has to be subscribed by the controller node. This furnish to the controller the current state of the simulated arm.
The simulated arm is commanded by providing set-point messages to the simulated controllers advertising the topics `/sherpa_arm/Jnt_X_effort_controller/command`

RRbot

In order to run the gazebo model of the 3-link simulated arm the following command has to be run:

```
→ roslaunch rrbot_gazebo rrbot_world.launch
```

The communication with this arm happens as described above.

C Appendix: User Manual

C.1 Instructions to start the software

1. Start `roscore`
2. Load parameters relative to motors and encoders on the arm using the following command:
   →`rosparam load sherpa_arm.yaml`
3. Run the node that enables communication with SPI bus:
   →`rosrun spi_interface spi_interface_node`
4. Run the node that enables communication with CAN bus:
   →`rosrun can_interface can_interface_node`
5. Run the node that contains implementation of joint controllers and state observer:
   →`rosrun controller controller_node`

Once the controller is running and the arm is calibrated the screen should show the following message:

sensors are calibrated

Start MoveIt by inputing the following command to terminal:

→ `roslaunch sherpa_moveit_config moveit_planning_execution.launch`

This will initialize MoveIt and the servers for the Joint Controller (`Joint_Trajectory_Action` node), the gripper(`Gripper_Action_Server` node), the impedance controller(`Impedance_Controller_` node) and the action manager(`Action_Manager_Server` node). It will also launch the RViz GUI for visualization and the `robot_state_publisher` node.

If everything is correct the terminal should output the following message:

```
[ INFO] [1476022702.826913056]: MoveGroup context using planning plugin ompl_int
erface/OMPLPlanner
[ INFO] [1476022702.826950877]: MoveGroup context initialization complete

All is well! Everyone is happy! You can start planning now!
```

Figure C.1: Feedback screen message

To start the docking routine the following launch file has to be run

→ `roslaunch elementary_actions pick_place.launch`

This will initialize the elementary actions and the serial communication with the gripper.

C.1.1 Starting the Impedance Controller

Before starting the impedance controller the parameter file containing trajectory constraint is loaded by the following command:

→ `rosparam load impedance_controller.yaml`

The impedance controller is started by inserting the following line in the terminal:

→ `rosrun impedance_controller impedance_controller`

At the actual state of development the elementary actions *Grasp* and *Dock* are executed without the use of the Impedance Controller package. They are integrated in reach_place node and executed through motion planning functionalities.
In order to test the impedance controller and execute those indipendently from the rest of the software two executable has been created. In order to test them the following command has to be run: Grasp: → `rosrun elementary_action grasp`

Dock: → `rosrun elementary_action dock`

In case you want to test the controller without running MoveIt! and all the other nodes, remember to run the following command:

→ `rosrun impedance_controller impedance_controller_action`

Which will start the action Server.

Bibliography

Barrett, E., M.Realing, G.Barbieri, M.Fumagalli and R.Carloni (2016), Mechatronic Design of the SHERPA Robotic Arm, University of Twente.

Broenink, J. F. and G. H. Hilderink (2001), A structured approach to embedded control systems implementation, in *Proceedings of the 2001 IEEE International Conference on Control Applications*, Eds. M. W. Spong, D. W. Repperger and J. M. I. Zannatha, IEEE, pp. 761–766, ISBN 0-7803-6733-2, doi:10.1109/CCA.2001.973960. http://www.ce.utwente.nl/rtweb/publications/2001/pdf-files/042R2001.pdf

B.Siciliano, L.Sciavicco, L.Villani and G.Oriolo (2009), *Robotics, Modelling, Planning and Control*, Springer.

Buss, S. R. (2009), Introduction to Inverse Kinematics with Jacobian Transpose, Pseudoinverse and Damped Least Squares methods.

E.Barrett, M.Fumagalli and R.Carloni (2016), The SHERPA Gripper: Grasping of Small-Scale UAVs, in *Proceedings of the IEEE International Symposium on Safety, Security and Rescue Robotics*.

Kavraki, L., P. Svestka, J.-C. Latombe and M. Overmars (1996), Probabilistic roadmaps for path planning in high-dimensional configuration spaces, in *IEEE Transactions on Robotics and Automation*.

LaValle, S. M. (2006), *Planning Algorithms*, Cambridge University Press, Cambridge, U.K.

L.Sciavicco, B. (1988), A solution algorithm to the inverse kinematic problem for redundant manipulator, in *IEEE Journal of Robotics and Automation*.

Marconi, L., C. Melchiorri, M. Beetz, D. Pangercic, R. Siegwart, S. Leutenegger, R. Carloni, S. Stramigioli, H. Bruyninckx, P. Doherty, A. Kleiner, V. Lippiello, A. Finzi, B. Siciliano, A. Sala and N. Tomatis (2012), The SHERPA project: Smart collaboration between humans and ground-aerial robots for improving rescuing activities in alpine environments, in *Proceedings of the IEEE International Symposium on Safety, Security, and Rescue Robotics*, pp. 1–4, doi:10.1109/SSRR.2012.6523905.

N.Xi, T.Tarn and A.Bejczy (June 1996), Intellingent Planning and Control for Multirobot Coordination: An Event-Based Approach, in *IEEE Transaction on Robotics and Automation*, volume 12.

S.Karam and E.Frazzoli (2011), Sampling-based algorithms for optimal motion planning, in *The International Journal of Robotics Research*, volume 30, Sage Publications, pp. 846–894.

Vanderborgh, B., A. Albu-Schaffer, A.Bicchi, E.Burdet, D.G.Caldwell, R.Carloni, M.G.Catalano, O.Eiberger, W.Friedl, G.Ganesh, M.Garabini, M.Grebenstein, G.Grioli, S.Haddanin, H.Hoppner, A.Jafari, M.Laffranchi, D.Lefeber, F.Petit, S.Stramigioli, N.G.Tsagarakis, M.V.Damme, R.V.Ham, L.C.Visser and S.Wolf (2013), Variable Impedance actuators: A review, in *Robotics and Autonomous Systems*, volume 61, pp. 1601–1614.

Wolovich, W. and H.Elliot (1984), A computational technique for inverse kinematics, in *Proceedings of the 23rd IEEE Conference on Decision and Control*.

Yildirim, S. and T.Tunali (1999), A new methodology for dealing with uncertainty in Robotic Tasks, in *International Symposyum on Computer and Information Sciences*, Bornova, Turkey.