POLITECNICO DI MILANO

Facoltà di Ingegneria
Scuola di Ingegneria Industriale e dell'Informazione
Dipartimento di Elettronica, Informazione e Bioingegneria

Corso di Laurea Magistrale in
Ingegneria Informatica

Un modello per tesi di laurea magistrale al DEIB

Relatore:

PROF. MATTEO MATTEUCCI

Correlatore:

????

Tesi di Laurea Magistrale di:

GIOVANNI BERI
Matricola n. 852984

Anno Accademico 2016-2017

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

GRAPE	Ground Robot for vineyArd Monitoring and ProtEction
ROS	Robot Operating System
LIDAR	Laser Imaging Detection and Ranging
UGV	Unmanned Ground Vehicle
ICC	Instantaneous Center of Curvature
IMU	Inertial Measurement Unit
ECHORD++	European Clearing House for Open Robotics Development Plus Plus
PCL	Point Cloud Library
FSA	Finite State Automata
API	Application Programming Interface
EKF	Extended Kalman Filter
SLAM	Simultaneous Localization And Mapping
AMCL	Adaptive Monte Carlo Localization

1 BACKGROUND AND TOOLS

In this chapter we are going to describe the general concepts this thesis deals with, together with the main tools we used to address the project. Since this thesis is in the frame of GRAPE project (see Chapter 2), most of them are typical of the robotic field and, more specifically, of the agricultural robotics. This last field should be seen in the wider context of the so-called *E-agriculture*; to give a precise definition of this term, we make reference to the FAO (Food and Agriculture Organization) definition[1]:

> *E-agriculture, or ICTs in agriculture, is about designing, developing and applying innovative ways to use ICTs with a primary focus on agriculture. E-agriculture offers a wide range of solutions to agricultural challenges and has great potential in promoting sustainable agriculture while protecting the environment.*

The GRAPE project, that will be described with further details in chapter 2, is about the design and realization of an UGV with control and operative task in a vineyard environment, so in this chapter we'll deal with topics concerning software development in robotics, estimation of the state of a robot, autonomous navigation

ROBOT OPERATING SYSTEM

ROS is the *robotic middleware* we used to develop the software components of the system described in this thesis. We decide to use it because of its great modularity, the availability of a very large number of packages, well documented Application Programming Interface (API)s and an active community. Moreover, ROS is a very widespread system, so its power and versatility are well known in the field of software development for robotics. Citing words from its official website[2], these are ROS main features:

> *It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers.*

1 http://www.fao.org/fileadmin/templates/rap/files/uploads/E-agriculture_Solutions_Forum.pdf

2 http://wiki.ros.org/ROS/Introduction

ROS is actually a *meta-operating system*, that is, it's not an operating system in the traditional sense (it requires to be run on top of an another operating system; currently, the only officially supported OS is Linux Ubuntu), but it provides a peer-to-peer network that processes can use to create and process data together. This network is implemented through TCP, and it's called *Computation Graph*. In this section, we're going to describe ROS with more detail, with particular emphasis on the different techniques that nodes can use to communicate among them. Keep in mind that, even if these techniques differs a lot, they all are strongly typed *i.e.* in order to define a channel (with *channel* we now mean one of the technique that we are going to describe. It's not the name of a specific communication tool) you also need to define the types of message that are going to be exchanged through it. ROS already defines a lot of useful message types (*e.g. LaserScan.msg*, *PoseWithCovarianceStamped.msg*), grouped by domain (*e.g.*, *Sensor_msgs*, *Geometry_msgs*). However a simple message definition language is provided, and users are encouraged to define their own message types to make them as self-explanatory as possible.

ROS MASTER Even if the Computation Graph is a peer-to-peer network, a central process, called **ROS Master**, is required to exist, to provide naming and registration services to all the user processes In this. Once the processes have located each other through the services offered by the Master, they can communicate peer-to-peer without involving a central entity;

NODES The processes that are in the Computation Graph are called **nodes**, and they are the atomic units of the computational graph. The ROS API are available in C++, Python and Lisp, but C++ is the most widely used. One of the aims of ROS is to be modular at a fine-grained scale, so a complex task should be achieved through cooperation of several different nodes, each with quite narrow tasks, rather than one large node that include all the functionalities. Nodes can use different techniques for communication, depending whether the message is a part of data stream or it is a request message (*i.e.* a response message is expected) and, in this last case, on the (expected) duration and complexity of the computation of the response.

TOPICS Topics implements a *publish-subscribe* paradigm, are they the easiest way that nodes can use to communicate with each other, and basically are named channels, characterized by the type of the messages that are sent through it. When a node *publish* a message on a certain topic, the message is read from all the nodes that previously *subscribed* to that topic, interfacing with the Master. Note that:

- this technique leads to a strong decoupling between publishers and subscribers to a topic, because a publisher node

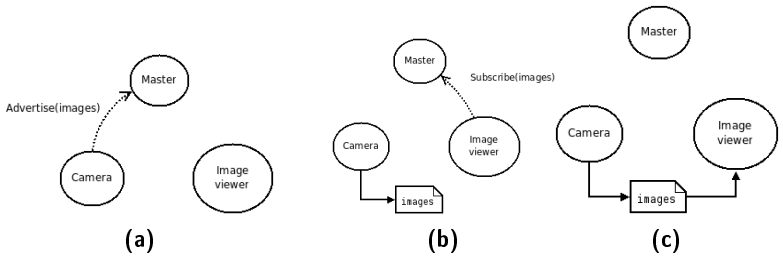


Figure 1.1: *Three phases of the setup of communication of nodes throught topics.*

is, from an high-level perspective[3], not even aware of the presence of subscribers, and vice versa.

- the relationship between publishers and subscribers is *many-to-many*, *i.e.* multiple nodes can publish on a topic, and multiple nodes can subscribe to a topic.

We can easily conclude that this method is very suitable for passing streams of data (*e.g.* the handler of a LIDAR streams its measurements over the network, or a node publish the velocities commands for the wheels of a robot), but there is no notion of a *response* to a message, so it's not suitable for *request-response* communication.

SERVICES Services are defined by a name, and a couple of message types that describe the *request* type and the *response* type. Each service is offered by a Service server to any Service client that perform a call. So, Services implement an inter-node communication that is very similar to traditional function calling in most common programming languages (*e.g.* C++, Java), in the sense that:

- Service calls are blocking
- using Services, the inter-node communication is *one-to-one*

These properties make Services suitable for punctual (in opposition to data stream) inter-node communication, such as: request of parameters values to another node, ask a node that handles a camera to take a picture, ask a node that perform navigation task to clear the current map.

ACTIONS While Services, with their resemblance to traditional function calls, can address pretty well the problem of *one-to-one* inter-node communication, they can be quite unsatisfying if the computation required to produce the response is demanding in term of execution time (*e.g.*, navigation of a robot from one point to

3 Actually, publisher nodes always know the list of nodes subscribed to their topics. But this is only used in connection phase, and to avoid a situation where a node publish on a topic with no subscribers, for the sake of efficiency.

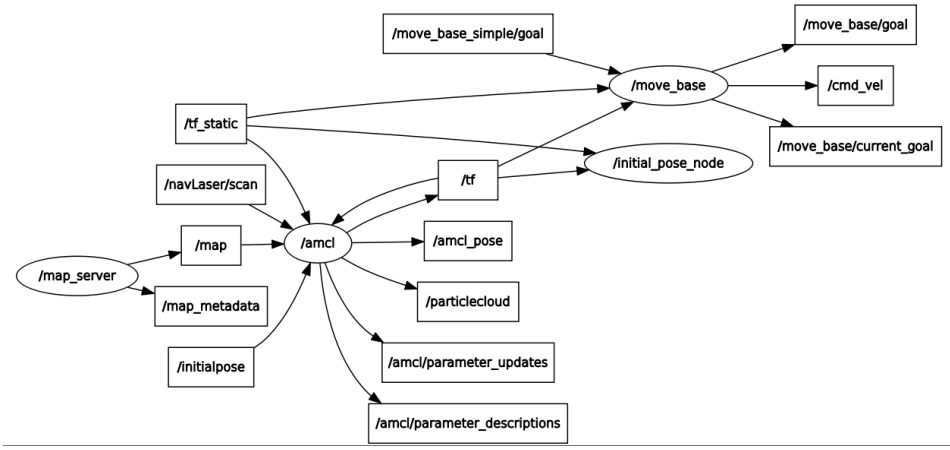


Figure 1.2: *An example of the* ROS *Computation Graph, visualized with* rqt*: nodes are represented with circles, rectangles represent topics, and arrows go from a node to a topic it publishes on, or from a topic to a node subscribed to it. It's easy to recognize the* many-to-many *relationship.*

another in an environment), because the caller is stuck at the line with the Service invocation until the end of the procedure. Services show their weaknesses also in situations where it could be useful to observe the intermediate results of the computation triggered by the request (*e.g.*, a very complex manipulation procedure). **Actions** are very suitable in this context because, at the cost of a more complex implementation, provide an asynchronous and fully preemptable remote procedure call, with the possibility of monitoring intermediate results if needed, and a native exit status to check the state (active, rejected, preempted, succeeded, aborted...) of the execution. Differently from Topics and Services, Actions are not native in ROS, and their functionalities are built on top of the other ROS messaging systems (See figure 1.4). Asynchronicity is provided by the use of callbacks.

A final consideration about the ROS mechanism for recording, and playing back, data published on topics. This software module is call *rosbag*, and takes advantage of the fact that ROS is aware of all the messages exchanged through its messaging system. This tool gets very useful in a project like GRAPE, where is not trivial at all to simulate or replicate in a laboratory environment the physical context of a vineyard, and of course meaningful data are required in order to correctly validate algorithms and tools. The paradigm is very simple: in every moment you can invoke a *rosbag record* command, that records every single message exchanged on the ROS topics; data are logged in a single file with *.bag* extension, that can be lately filtered and/or compressed. Bag files can be played back in a very easy way invoking *rosbag play* command; the playing of the bag is implemented with a single node that streams messages on all the topics existing in the bag (see Figure 1.3). The module also provides an option to use *simulated*

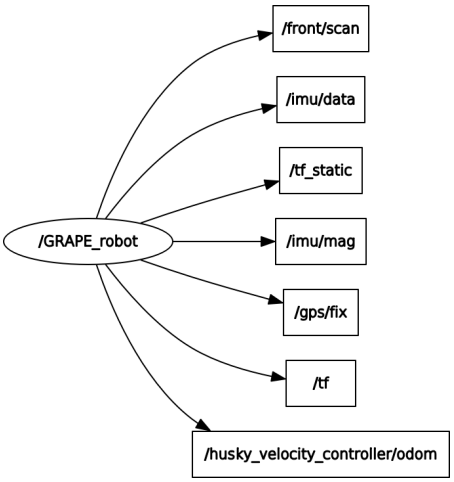


Figure 1.3: *Visualization of the ROS Computation Graph with* rqt *in the context of a data bag being played: there is a single node* (GRAPE_robot) *that simultaneously plays recorded topics messages.*

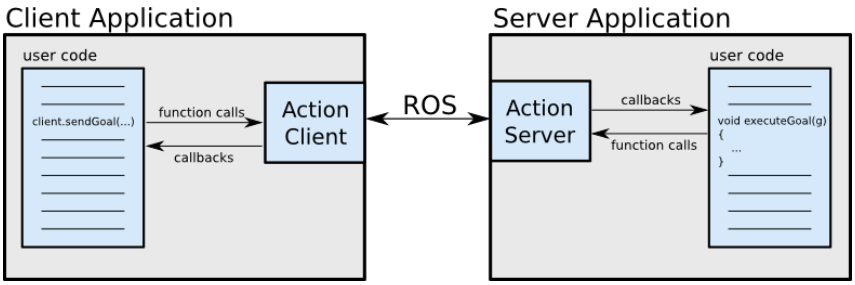


Figure 1.4: *Sketch of the implementation of actionlib, through ROS topics and a callback system.*

time, *i.e.* playing the bag exactly with the same message timing as in the real recorded session. In this situation, the decoupling induced by *publish-subscribe* is a strong advantage, since you can substitute all publishing nodes with a single one, with no consequences[4].

TF: THE TRANSFORM LIBRARY

tf is a ROS library, which task is very important to understand in order not to get lost in the next sections and chapters. The goal of *tf* is:

> *" [...] provide a standard way to keep track of coordinate frames and transform data within an entire system such that individual component users can be confident that the data is in the coordinate frame that they want without requiring knowledge of all the coordinate frames in the system" (Foote, 2013)*

4 Except in some pathological cases: for example, a node can require the list of active nodes in the computational graph, and the response would be different when the data are recorded and when the corresponding bag is played.

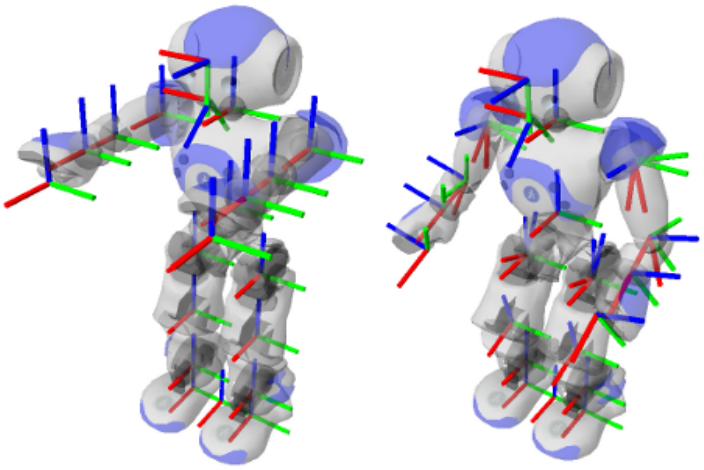


Figure 1.5: *A robot in 2 different positions, with* tf *frames in evidence: x-axis is red, y-axis is green, z-axis is blue. The frames are the same in both configuration, but the transformations* (i.e. *rototranslations) between them are different.*

The utility of such a component is straightforward, even in quite simple robotic systems. We'll describe here a situation we stumbled upon exactly in the development of the GRAPE project, where of the utility of *tf* is very easy to understand; you'll be able to better contextualize this example after you've read Chapter 5. In this example a LIDAR, mounted on top of the final joint of a robotic arm, acquires data while the arm is moving in order to create a point cloud that will be processed later. To get a meaningful point cloud, it's mandatory to keep track of the movement of the LIDAR with respect to a point with speed equal to zero (*e.g.* the base link of the arm, or the base link of the whole robot), and this gets even more difficult because of the multiple (6 in our specific case) joints of the arm; but this problem can be easily addressed by means of *tf*.
Note that frames are very useful for two main tasks:

- represent the configuration of the robot, by assigning frames to the significant physical elements of the robot (*e.g.* joints, sensors, wheels). Since *tf* graph is a tree, a root element of the robot should exists; in a typical configuration, the frame name is *base_link*, it's placed in the geometrical center of the robot, and all the other frames linked to physical components of the robot belong to the subtree with root *base_link*.

- represent the position of the robot in an environment. A typical example is the frame *odom*, that is typically centered in the point where the robot is located when it's switched on. Another example is the frame *map*, that is used as a reference in case the robot is localized not only with respect to its initial point, but in a given map.

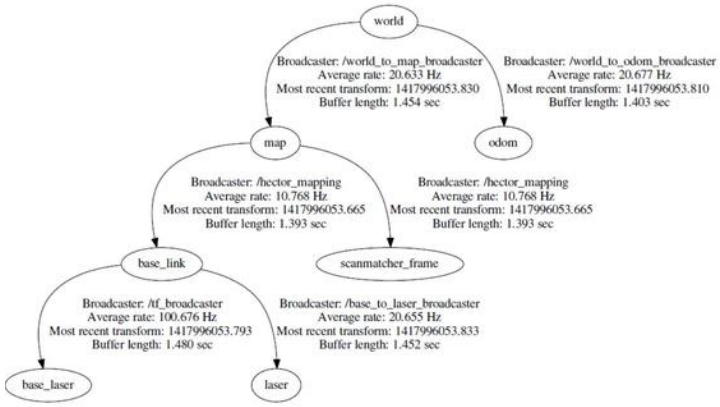


Figure 1.6: *An example of* tf *tree*

We are now giving a a bit more techical detail about *tf*. *tf* implementation relies on ROS topics (see Section 1.1) and achieves the goal mentioned before by building an oriented graph where vertices are reference frames, and edges are transformations (rototranslations) between frames. *tf* does not assume a constant structure and, if a path exists between two reference frames in the graph, the direct transformation between them can be computed by composition of transformation. Since, in general, multiple paths between 2 vertices can exists in a directed graph and this could lead to ambiguity in computing the transformation between two reference frames, the graph is forced to be acyclic. Disconnected subgraphs are allowed, but of course transformation between vertices that belong to different subgraphs cannot be computed. The main components of the library are:

- ***tf* broadcasters**: they are simple software components, that publish a transformation between two reference frames every time an update is available. Different broacasters does not sync together the publishing phase
- ***tf* listeners**: they are more complex components, because they take into account that broadcasters are not synced. Since both transformations and queries to *tf* graph are stamped, listeners make use of queues to store the most recent transformations, and they interpolate old values using SLERP (Spherical Linear intERPolation) to return a transformation for which there is no measured value at the requested timestamp.

In figure 1.5 you can see a graphical representation of the reference frames tracked in a Nao Robot, while figure 1.6 shows an example of *tf* graph visualized with visualization framework *rqt*.

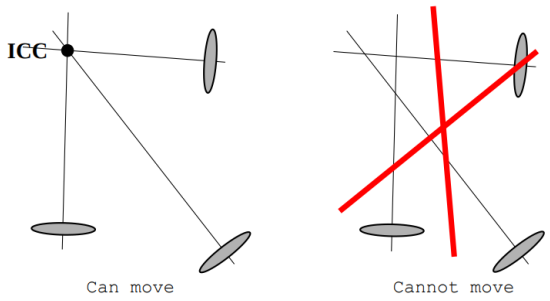


Figure 1.7: *If the axis of all the wheels intersect in a single point, it's called ICC and the robot can move without slipping*

ODOMETRY

The problem of odometry, *i.e.* estimation of the position of a robot in an environment is harder than it could seem. Formally, odometry estimation is the problem of estimating over time the tuple:

$$< x, y, z, \theta, \dot{x}, \dot{y}, \dot{z}, \dot{\theta} > \tag{1.1}$$

given the measurement of some motion sensors. To better understand the complexity of the problem, let's analyze an extremely simple model: a robot with a single, freely rotating wheel. In this frame, assuming rotary encoders on the wheel, we can think about measuring directly the wheel speed and integrate these measurement to get the traveled distance, and measure the variation in the orientation of the wheel to get the position. But actually there are a lot of imperfection that can lead to error, for example:

- wheel can be non perfectly perpendicular to the ground
- the friction between the floor and the wheel might not be enough to avoid slippage (expecially)
- there is no such thing as a perfect sensor, so the use of encoders introduce an error

Even if all these concurrent causes seem negligible, you have to take into account that the errors sum up over time, so an error of a few millimeters per meter might become significant over time.

Moreover, the probability of slippage gets higher in systems with more than one wheel, because, because for a system with multiple wheels to move without slippage, a point must exists around which all the wheels can move along a circular path. This point is called Instantaneous Center of Curvature (ICC) (see Figure 1.7), and can be easily identified by looking for the intersection of the axis of all wheels. If the intersection exists in a single point, it's called the ICC. But even if the odometry estimated from the wheels is not a good

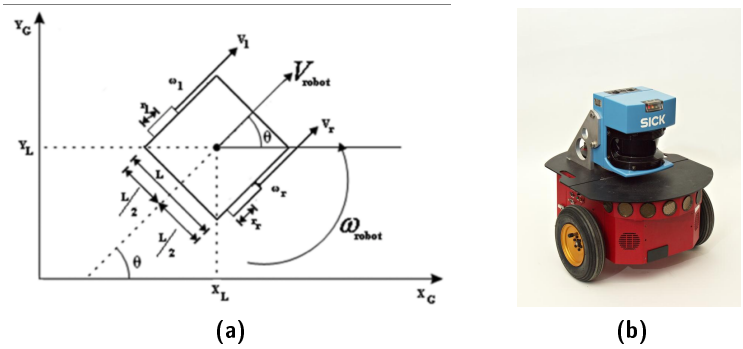


Figure 1.8: *On figure 1.8a, the scheme of a differential drive motion model; in figure 1.8b, an example of a differential drive robot (Pioneer 3DX).*

solution if used alone, it can be used as a starting point for other, more complex, method. For this reason we are now going to describe how to estimate the odometry starting from the wheels encoders in our specific robot. This computation is different according to the **motion model** of the considered robot. As we'll see in Section 3.1, the robot we used is the Husky platform (see Figure 1.10) from Clearpath Robotics, that moves with a *skid steering* kinematics, that is a derivative of *differential drive* kinematics. Thus, we're going to describe these two motion model with more detail.

Differential drive robot

In a differential drive system, the movement of the robot is only based on two separately driven wheels, placed on either side of the robot, on the same axis (see Figure 1.8), and optionally a central, non-actuated caster wheel for stability. The two side wheels are not steerable, so the changes of direction are realized through application of different speed to the two wheels. For example, intuitively, if the wheels move at the same speed and in the same direction, the robot will move straight; if the wheels move at the same speed but different directions, the robot rotates in place. By recalling the definition of ICC, we observe that, if the wheels are correctly aligned, a differential drive robot always have a well-defined ICC and the slippage of the wheels is not very accentuated.

At each instant in time, since the ICC is well-defined, both the left and right wheel follow a path that moves around ICC at the same angular speed ω, and thus:

$$\begin{cases} \omega(R + \frac{L}{2}) = v_r & (1.2) \\ \omega(R - \frac{L}{2}) = v_l & (1.3) \end{cases}$$

where L is the distance between the center of the two wheels, v_r and v_l are, respectively, the linear velocity of the right and left wheel, R is the signed distance between the ICC and the midpoint of the wheels. Note that the only parameter constant through time is L, since it's a physical property of the robot structure, while all other parameter evolve during the movement.
By combining 1.2 and 1.3, we get:

$$R = \frac{L}{2}\frac{(v_r + v_l)}{(v_r - v_l)}, \qquad \omega = \frac{(v_r - v_l)}{L} \tag{1.4}$$

Observing these results we can validate the intuitive impressions about particular cases made a few lines above:

- if $v_r = v_l$, the curvature radius is infinite, because the robot is moving straight.
- if $v_r = -v_l$, the robot is moving around the midpoint of the wheels

We give now some details about odometry computation. Let's assume, in a certain moment $t = t_0$, that the robot pose is (x, y, θ). We assume that in the time interval $t_0 \rightarrow (t_0 + \delta t)$ the values v_r and v_l are constant; if we observe figure 1.9 under these condition, we have:

$$ICC = (x - R\sin\theta, y + R\cos\theta) \tag{1.5}$$

Write now the expressions for (x', y', θ'):

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos(\omega\delta t) & -\sin(\omega\delta t) & 0 \\ \sin(\omega\delta t) & \cos(\omega\delta t) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x - ICC_x \\ y - ICC_y \\ \theta \end{bmatrix} + \begin{bmatrix} ICC_x \\ ICC_y \\ \omega\delta t \end{bmatrix}$$

With this procedure we have identified 3 of the elements of the target tuple (see expression 1.1), but we still have to retrieve x,y and θ. For this reason we consider that by assuming an initial pose (x_0, y_0, θ_0), knowing that:

$$V(t) = \frac{(v_r + v_l)}{2} \tag{1.6}$$

where $V(t)$ represent the overall speed of the robot, and assuming to know the functions $v_r(t)$ and $v_l(t)$ *i.e.* the linear speed of the wheel in time, we can calculate (x, y, θ) by integrating the speed of the robot over time, that is:

$$x(t) = \int_0^t V(t)\cos(\theta(t))dt \tag{1.7}$$

$$y(t) = \int_0^t V(t)\sin(\theta(t))dt \tag{1.8}$$

$$\theta(t) = \int_0^t \omega(t)d\omega \tag{1.9}$$

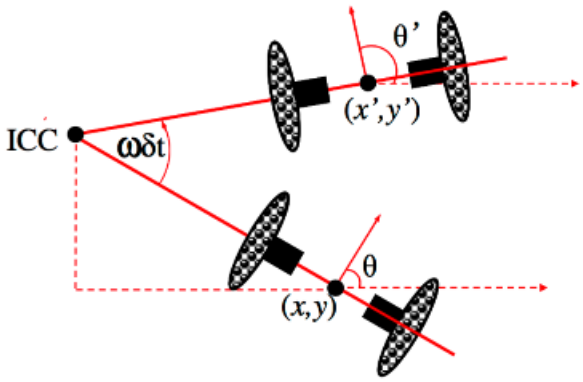


Figure 1.9: *Differential drive:* $(x, y, \theta) \rightarrow (x', y', \theta')$

and, in our specific case we can write it as function of v_l and v_r, that are the quantities that are directly measured on the wheels.

$$x(t) = \frac{1}{2}\int_0^t (v_r(t) + v_l(t))\cos(\theta(t))dt \tag{1.10}$$

$$y(t) = \frac{1}{2}\int_0^t (v_r(t) + v_l(t))\sin(\theta(t))dt \tag{1.11}$$

$$\theta(t) = \frac{1}{L}\int_0^t (v_r(t) - v_l(t))(t) \tag{1.12}$$

So the tuple required by the odometry calculation is now complete.

Skid-steering

However, observing image 1.10, it's very easy to contest that the Husky platform used in GRAPE project is not similar to the model described in previous section, because the number of actuated wheels is four instead of two. However, the motion model of the Husky is more similar to the differential drive because:

- wheels are not steerable
- being $v_{f\star}$ the speeds of the front wheels, $v_{r\star}$ the speeds of the rear wheels, $v_{\star r}$ the speeds of the right wheels, $v_{\star l}$ the speeds of the left wheels, we always have:

$$v_{fr} = v_{rr} \tag{1.13}$$

$$v_{fl} = v_{rl} \tag{1.14}$$

This type of motion model is called *skid steering*, and is often use in real-world applications (see figure 1.11b) because of its simple and robust mechanical structure that leaving more room in the vehicle for the mission equipment. In addition, it has good mobility on a variety of terrains, which makes it suitable for all-terrain missions.

Figure 1.10: *Husky platform from Clearpath Robotics is the platform used for the development of GRAPE project.*

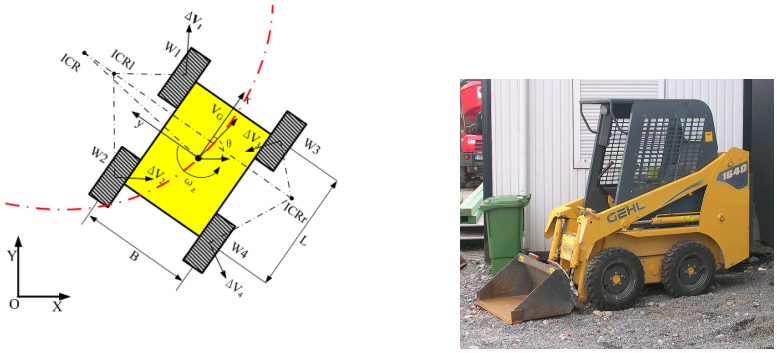


Figure 1.11: *On figure 1.11a, the scheme of a skid steering motion model; in figure 1.11b, an example of a skid steering vehicle.*

But, of course, it also present a variety of problems and weakness. For example, if you recall what we told about ICC in Section 1.3, it's clear that the ICC of such a model will always be at the infinite, since wheels are organized in 2 parallel rows that cannot steer. Thus, skid steering robots can't turn without slipping of the wheels! From a theoretical point of view, the main consequence is a a complexity in obtaining an accurate kinematics and dynamic model of *skid steering* (Yi et al., 2007). On the other hand, in the context of GRAPE project this leads to two major practical problems:

- the slippage of the robot leads to higher power consumption of the electric engines with respect to the a system with explicit steering (Shamah, 1999). This is something to be taken into account in the sizing phase of the power system of the robot.
- the estimation of the odometry is going to be much more imprecise than in the differential drive case, for the error introduced by the slipping of the wheels.

Actually, there is a quite simple way (Wang et al., 2015) to model with an acceptable approximation the skid steering as an equivalent

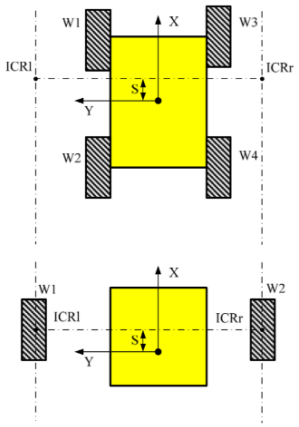


Figure 1.12: *The equivalence of differential drive motion model and skid steering model according to Wang et al., 2015*

differential drive system, where the distance between the wheels is obtained multiplying the original distance for a factor χ that is function of the physical structure of the robot This approximation relies on some assumptions:

- the mass center of the robot located at the geometric center of the body frame.
- the two wheels of each side rotate at the same speed.
- **the robot is running on a firm ground surface, and four wheels are always in contact with the ground surface.**

We cannot guarantee that conditions 1) and 2) will hold in our context, but we are almost sure that condition 3) is **not** going to be verified, given the condition in which our robot is going to operate (vineyard terrain), so this is another reason for the wheel odometry not to be very precise. Thus, the integration of several sensors beyond the wheels encoders is essential to correct the wheel odometry.

SENSOR FUSION

First of all, clarify what we mean for sensor fusion, following the definition of Elmenreich, 2002:

> *"Sensor fusion is the combining of sensory data or data derived from sensory data in order to produce enhanced data in form of an internal representation of the process environment. The achievements of sensor fusion are robustness, extended spatial and temporal coverage, increased confidence, reduced ambiguity and uncertainty, and improved resolution."*

Let's analyze this last list, element by element, recalling the analysis of Remagnino, Monekosso, and Jain, 2011:

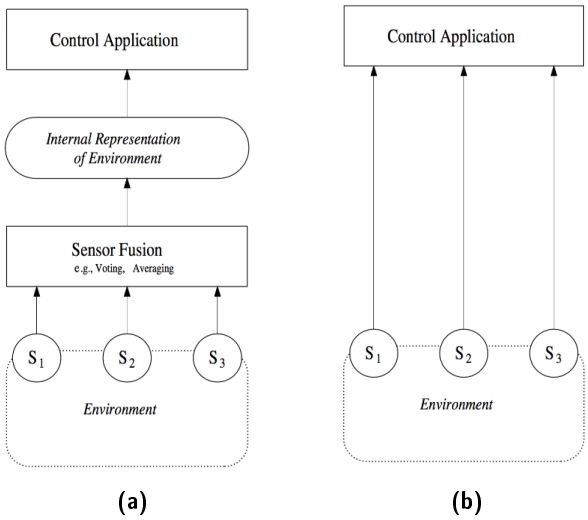


Figure 1.13: *Sensor fusion 1.13a vs multi-sensor integration 1.13b*

- *robustness*: redundancy generated by the presence of multiple sensors make the robot more resistant to partial failures
- *extended spatial and temporal coverage*: disseminate sensors are more likely to measure the dimension concerned with temporal continuity and from several positions
- *increased confidence*: measures confirmed by more that one sensor are given a larger weight
- *reduced ambiguity and uncertainty*: joint information reduces the set of ambiguous interpretations of the measured value
- *robustness against interference*: by increasing the dimensionality of the measurement space (*e.g.*, measuring the desired quantity with optical sensors and ultrasonic sensors) the system becomes less vulnerable against interference.

Even if the name could be misleading, sensor fusion is different from *multi-sensor integration* in the sense that multi-sensor integration only consists in the simultaneous use of disparate sensor sources in order to accomplish a goal task. Sensor fusion techniques make a step further, and aim to the construction of a single common representation, using all the available sensor sources. The difference between multi-sensor integration and sensor fusion is described graphically in image 1.13, to underline that sensor fusion actually provides one single representation of the state of the environment that is used as a single input stream by the control application, while in multi-sensor integration the application uses each of the sensor data streams as direct input. In this context, we are assuming the definition of environment, from Thrun, 2002: pose of the robot, velocity of the robot,

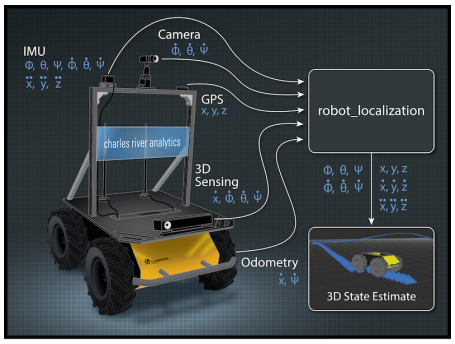


Figure 1.14: *A scheme of Robot Localization senor fusion*

and velocity of the joints, configuration of actuators, location and features of surrounding objects, location and velocity of moving object and people.

In ROS ecosystem, several open source solutions exist that implements *sensor fusion-based* odometry; we tried two of them, which gave proof of good functioning in the past: **ROAMFREE** (Cucci and Matteucci, 2013, Calabrese, 2014, Cucci and Matteucci, 2014), a graph-based algorithm, and **Robot Localization** (Moore and Stouch, 2014, Dudek, Szynkiewicz, and Winiarski, 2016, Mohanty et al., 2016), based on Kalman filters.

After the experimental campaign in Casciano Terme, ROAMFREE turned out to be harder to configure for our specific problem, while Robot Localization had some specific documentation about the its usage in the required context[5], so we opted for this last one. Further details about our usage or Robot Localization will be given in Chapter 4, while we are now giving a few hints about its general functioning.

Robot Localization

Citing Robot Localization documentation[6]:

> *"Robot_localization is a collection of state estimation nodes, each of which is an implementation of a nonlinear state estimator for robots moving in 3D space. It contains two state estimation nodes, ekf_localization_node and ukf_localization_node". In addition, robot_localization provides navsat_transform_node, which aids in the integration of GPS data."*

The Robot Localization algorithm is implemented specifically for ROS, and it's widely adopted by ROS community for its documentation, and its easiness of use and configuration. It fuses an unlimited number of sensor sources, for each of which you can specify a configuration vector given in the frame id of the input message *i.e.* you

5 http://docs.ros.org/kinetic/api/robot_localization/html/integrating_gps.html

6 http://docs.ros.org/kinetic/api/robot_localization/html/

can specify the message fields to be fused in the global estimate. For example, since GPS output is not so precise about altitude estimation, you can specify that only latitude and longitude parameters to be fused in the global estimate.

The state variables considered by Robot Localization are:

$$< x, y, z, \psi, \theta, \phi, \dot{x}, \dot{y}, \dot{z}, \dot{\psi}, \dot{\theta}, \dot{\phi}, \ddot{x}, \ddot{y}, \ddot{z} > \tag{1.15}$$

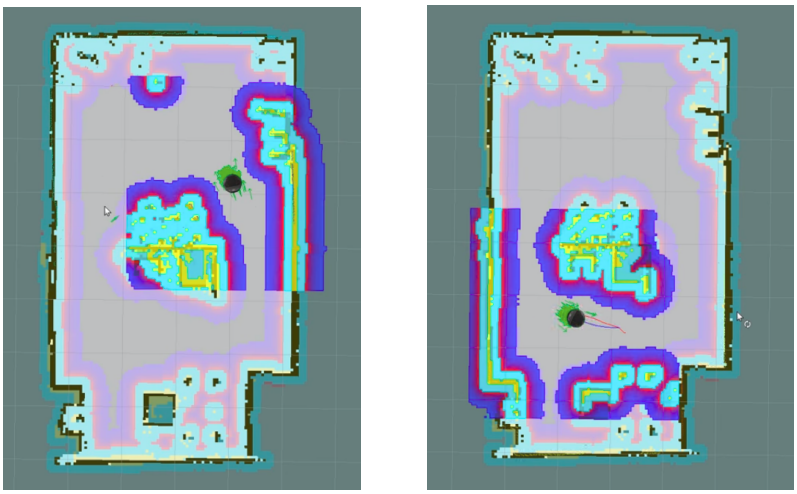
where ψ, θ, ϕ correspond to the Euler angles *roll, pitch, yaw*. The node accepts as inputs several types of ROS messages:

- *nav_msgs/Odometry*: the odometry estimation output by another node can be used entirely as input; this is the case of the odometry estimated only using wheels encoders, and it also allows for cascaded localization nodes. This will be useful in the GRAPE project, as we'll se in Chapter 4.
- *sensor_msgs/Imu*: the output of an IMU sensor *i.e.* a device composed by accelerometer, gyroscope, and magnetometer
- *geometry_msgs/PoseWithCovarianceStamped*: an estimate of the position and orientation of the robot, together with the timestamp of the measure and its covariance matrix
- *geometry_msgs/TwistWithCovarianceStamped*: an estimate of the linear and angular velocities of the robot, together with the timestamp of the measure and its covariance matrix.

Moreover, the Kalman filter implemented in Robot Localization is capable of *continuous estimation*: if for some reason no data are received from the various sensors for a large enough amount of time, the filter keeps producing odometry estimation exploiting an internal motion model. Remember that Robot Localization can be configured also to publish the *tf* transformation associated to the estimated odometry. This is one of the usage of *tf* described in Section 1.2.

NAVIGATION STACK

In Computer Science, a software stack is a set of programs that collaborate, in order to achieve a common goal. Since we are talking about ROS ecosystem, the programs are ROS nodes, and in the case of Navigation Stack, the common goal is to provide a modular out-of-the-box navigation system for UGVs. It was originally developed for Willow Garage's *PR2* robot, but its usage can be easily extended to differential drive and holonomic wheeled robots. The planner, Move Base, is known to be a very good solution for indoor navigation (Marder-Eppstein et al., 2010), so we were not sure about its performance in an outdoor environment as required in GRAPE project. Luckily, it turned



(a) (b)

Figure 1.15: *Local and global costmap visualized with* RViZ*. As you can see, the local costmap (in brighter colors) is built around the robot, and moves together with it.*

out to be a very good solution even in our situation, so it was integrated in the final navigation system. We are now giving a brief description of the main building blocks of the Navigation Stack (figure 1.16):

ODOMETRY SOURCE A topic from which read the pose estimated from the odometry system *e.g.* simple wheels odometry, output of a sensor fusion node as Robot Localization or ROAMFREE.

SENSOR SOURCE A topic from which read the data coming from the laser sensors (LIDAR) that are probing the environment, for obstacle avoidance, localization and possibly mapping tasks.

AMCL Implementation of a probabilistic localization system, based on the Monte Carlo localization, as described in Fox et al., 1999

MAP SERVER A ROS node where the map is published as a single topic message. This block is not used if Move Base is used in mapless mode

BASE CONTROLLER This is the only output topic of the Navigation Stack, and of course it contains speed commands for the base of the robot.

LOCAL AND GLOBAL COSTMAP They represent the information about the obstacles on the 2D plane of the ground, with a certain inflation radius that represent the size of the robot base. Each cell of the gridmap is associated to a certain cost that measure "how hard is it" to traverse that cell of the gridmap; possible

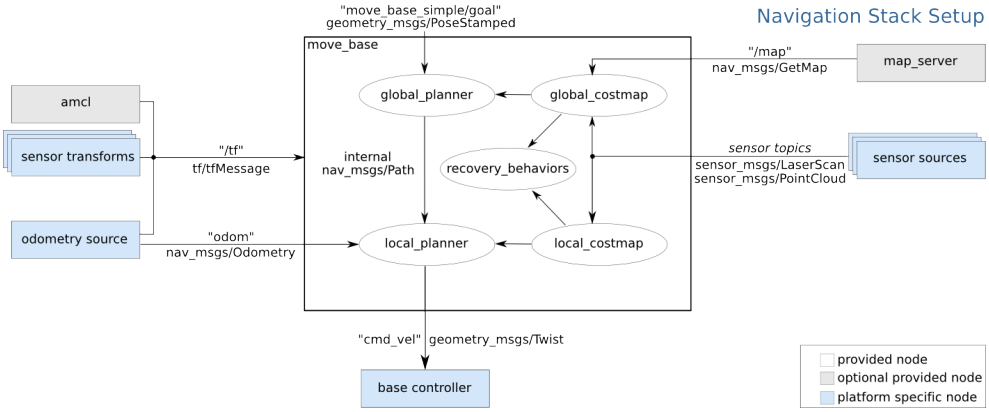


Figure 1.16: *A scheme representation of the Navigation Stack; see the color legend for classification in provided, optional provided, and platform specific nodes.*

values to represent the severity of obstacles are *Lethal obstacle*, *Inscribed*, *Possibly circumscribed*, *Freespace*, *Unknown*. The **global** costmap represent whole environment as is build from a known map, while the **local** costmap is built using only incoming laser scans. Local map is, in general, a scrolling window that moves in the global costmap in relation to robot current pose, and it's continuously updated.

LOCAL AND GLOBAL PLANNERS Global planner takes as input the global costmap, and traces the path with lowest cost from current position of the goal position; the local planners instead takes care of continuously update (if required) the global plan in the light of the incoming laser measurements. This combination is essential in case of unexpected obstacles.

MOTION PLANNING

Motion planning is a term used in robotics for the process of breaking down a desired movement task into discrete motions that satisfy movement constraints, and possibly optimize some aspect of the movement. This process is essential for autonomous systems, that accepts high-level description of tasks and should require no further human intervention; in this way the user can focus on *what* he wants done, instead of *how* to do it (Latombe, 2012). Motion planning can be applied to any kind of mechanical device equipped with actuators and sensors under the control of a computing system (*e.g.* a robotic arm that perform *pick-and-place*, a differential drive robot that has to reach a goal in an environment). Motion planning includes (among other): obstacle avoidance, computation of collision-free paths, building reliable sensory-based motion strategies.

More concisely, the basic problem of motion planning is, given:

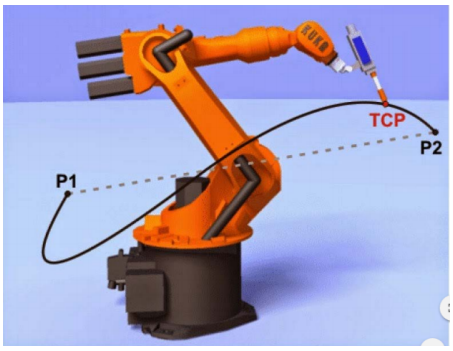


Figure 1.17: *A graphical representation of the trajectory planned for moving a robotic arm from start (P1) to goal pose (P2), satisfying geometrical constraints of the robot.*

- a start pose of the robot
- a desired goal pose (position, orientation)
- a geometric description of the world
- a geometric description of the robot (both as geometric shape, and as kinematic constraints due to kinematic motion model of the robot)

finding a path that moves the robot gradually from start to goal while never touching any obstacle.

Even if the movement of the robot has to be executed in the real world, the planning of the motion is typically computed in the *configuration space*, or *C-space* (Lozano-Perez, 1983). Formally, let: the robot A, at a certain position and orientation, be described as a compact subset of $W = \mathbb{R}^n$, $N = 2$ or 3, and the obstacles $B_1, \ldots, B_n$ be closed subsets of W. In addition, let F_a and F_w be Cartesian frames embedded in A and W, respectively. F_a is a moving frame, while F_w is fixed. A **configuration** q of A is a specification of the position T and the orientation Θ of F_a with respect to F_w. The **configuration space** of A is the space C of all the configurations of A. Usually, configuration space is high dimensional; a configuration is expressed as a vector of positions and orientations, so the robot in configuration space is always represented as a point. We distinguish the configuration space from the *workspace*, that is the physical environment in which the robot move. A few examples:

- the robot is a single point and the workspace is a 2-dimensional plane; C is a plane, and a configuration can be represented using two parameters (x, y).
- the robot is a 2D shape that can translate and rotate, the workspace is a 2-dimensional plane; C is 3-dimensional and a configuration can be represented using 3 parameters (x, y, θ).

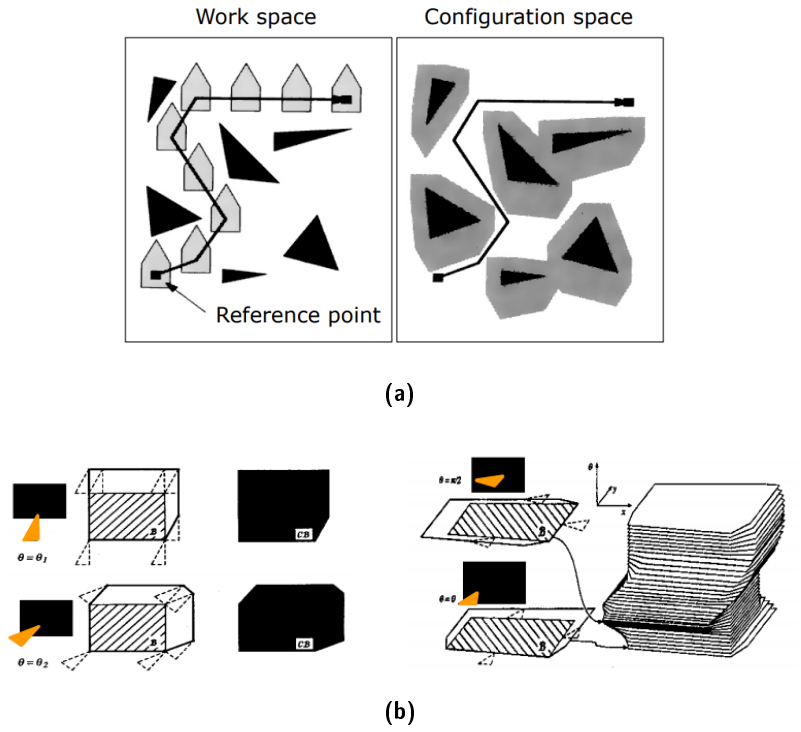


Figure 1.18: *The graphical construction of C-space from workspace, by sliding robot shape along the borders of obstacle regions. In Figure 1.18a you can see the procedure in 2D, in Figure 1.18a in 3D. Note that in 3D you only need to compute 2D procedure for each θ, and then stack all obtained images.*

- the robot is a solid 3D shape that can translate and rotate, the workspace is 3-dimensional; C is 6-dimensional, and a configuration requires 6 parameters: (x, y, z) for translation, and Euler angles (ψ, θ, ϕ).
- the robot is a fixed-base manipulator with N revolute joints and no closed-loops; C is N-dimensional.

Note that, as physical workspace, the configuration space is splitted in free space (C_{free}), and obstacle space (C_{obs}). From a practical point of view, *C-space* can be drawed by virtually sliding the robot shape along the edge of the obstacle regions, inflating them of the area covered by the robot. You can see some graphical examples of this procedure in Figure 1.18.

In simple configuration spaces, *grid-based search* algorithms can be used. With this approach, a grid is overlaid to configuration space, and assume that each configuration is identified with a grid point. With this approach, the problem is reduced to a graph search and exact algorithms like Dijkstra and A*, or suboptimal algorithms like A*, ARA*, AD* can be used.

But this approach gets easily unfeasible (for example, A* has complexity $O(b^d)$ (Hart, Nilsson, and Raphael, 1968), with b branching

factor and d depth of shortest path), so **sample-based approaches** are currently state-of-the-art planning algorithms. Sampling bases approaches, in a nutshell:

- are more efficient in most practical problems but offer weaker guarantees
- are probabilistically complete: increasing computing time, the probability tends to 1 that a solution is found if one exists (otherwise it may still run forever)
- performance degrades in problems with narrow passages

In sample-based planning there isn't a explicit characterizing of C_{free} and C_{obs}, but only a collision detection algorithm that probes C to see whether a certain configuration lies in C_{free} or not. An example of sample-based planning algorithm is **Rapidly exploring random trees** (LaValle, 1998); its pseudocode is shown in Algorithm 1.

Algorithm 1 RapidlyExploringRandomTrees(q_{goal})

```
1:  G.init(q_0)
2:  iterNumber ← 0
3:  N ← 100                                          ▷ for example
4:  repeat
5:      if iterNumber mod N ≠ 0 then
6:          q_rand ← RANDOM_CONFIG(C)
7:      else
8:          q_rand ← q_goal
9:      q_near ← NEAREST(G, q_rand)
10:     G.add_edge(q_near, q_rand)
11:     iterNumber ← iterNumber + 1
12: until q_near = q_goal
```

In the context of GRAPE project, motion planning is an essential component in two different modules:

1. Navigation module: **Move Base**, described in Section 1.5 is a motion planner, that makes use of grid-based planning algorithm, and is used for the navigation of the robotic base in the vineyard environment
2. Manipulation module: **MoveIt!**[7] is a state-of-the-art sample-based planner, and is used to plan the motion of our robotic arm in manipulation tasks (see Figure 1.20 for a block scheme of *MoveIt!* architecture).

7 http://moveit.ros.org/

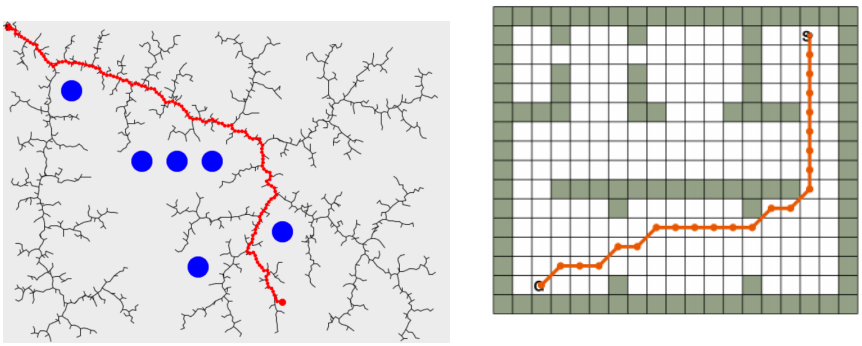


Figure 1.19: *Graphical representation of the ways of proceeding of a sample-based planner (1.19a), and of a grid-based planner (1.19b)*

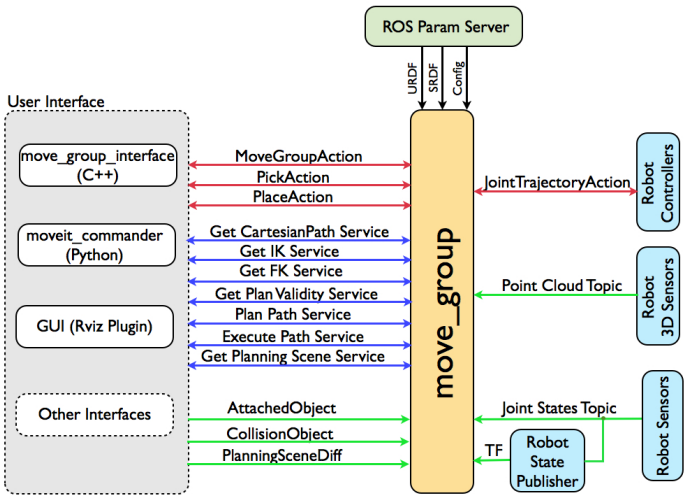


Figure 1.20: *A block scheme representing the high-level MoveIt! architecture.*

2

THE GRAPE PROJECT

In this chapter, we are going to give a description of the project this thesis work is part of, with particular emphasis on the parts that were specifically addressed in the thesis work. For the description of the project, we make reference to its official proposal.

PROJECT DESCRIPTION AND GOALS

GRAPE is an experimental project of European Clearing House for Open Robotics Development Plus Plus (ECHORD++) and, as hinted in the first part of Chapter 1, it focuses on vineyard farming activities and aims at setting up a robotic manipulation platform able to support lead users to develop a variety of farming applications. In effect, the main goal of GRAPE is not the complete development of an industrial platform, but, coherently with the research scenario, the primary objectives are:

1. the development of example applications in the pre-mentioned context, exploiting and improving the so-called *key enabling technologies*[1] in robot navigation, perception and manipulation of the project partners; the resulting capabilities are likely to allow to make it easier for small and medium enterprises working in the fields of agricultural robots and, more in general, plant protection.

2. increase of robot acceptance by farmers and agronomists: since the very last goal of this project is not solely about pure research but about the enterprise world too, particular attention is given into the realization of a robotic platform that could be accepted by potential end users. For this reason, a constant interaction between the project partners and the potential stakeholders (*i.e.* vinegrowers) is mantained also in development phases.

The example applications have been selected in order to challenge perception and action capabilities, to achieve vineyard monitoring, navigation and manipulation tasks. This decision is taken in the scope of turning traditional farming into precision farming; the realization of such a turn would allow for both the decrease of the chemical load

1 Key Enabling Technologies are a group of six technologies: micro and nanoelectronics, nanotechnology, industrial biotechnology, advanced materials, photonics, and advanced manufacturing technologies. They have applications in multiple industries and help tackle societal challenges; they are consedidered crucial in the creation of advanced and sustainable economies.

(a) (b)

Figure 2.1: *A vine before (2.1a) and after (2.1b) having leafed out.*

in food and environment, and an improvements of profits and yield for farmers, that would get a return for their investment (Herring, 2001). The introduction of precision farming tecniques leads indeed to a lot of advantages, for example early detection of plants diseases, or application of pesticides and fungicides with high precision and only when/where needed.

It's clear that some of these high-precision tasks are still too complex to be automated, and current state-of-the-art in research and technology have been proved to be not yet mature to give rise to a commercial product able to compete with a skilled agricultural agent. The goal of GRAPE is creating the enabling technologies to allow agricultural companies to develop vineyard robots to keep filling the gap that exists with respect to traditional methods.

GRAPE project specification identifies these two application examples:

- **Vineyard monitoring and autonomous navigation**: the developed robotic platform must be able to autonomously navigate the vineyard and localize itself into a map of the environment (downstream a mapping phase), to be able to monitor the state of the vineyard (*e.g.* foliage and grapes inspection). The localization and navigation parts, however, are a key point for any kind of tasks for an UGV field robot, because of the probability of high slope ground, rugged terrain morphology and unstructured map. We'll see the characterization and analysis of the problem of navigation in a rugged outdoor environment in chapter 4.

- **Autonomous application of pherormone dispensers**: pherormone dispensers are used for *mating disruption* tecniques, to protect grapevines from grape moths, by disrupting the bugs' reproductive cycle making use of synthesized sex pherormones. Also in this case, the most relevant challenge is brought by the unstructured environment, that makes the sensing and manipulation tasks significantly harder. However, note that pherormones deployment task gets easier if you think that the timing of the reproductive cycle (on which of course we have no control

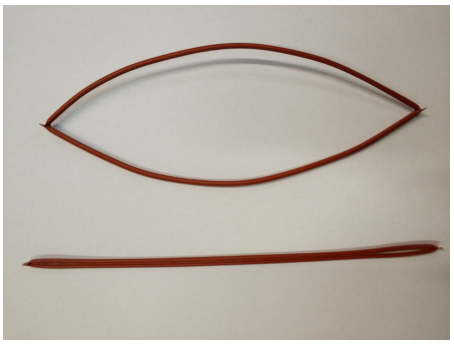


Figure 2.2: *Detail of our dispenser type.*

at all) forces this operation to be performed in a season where grape plants are pruned, and have not leafed out yet (Carde, 1995). Of course, the absence of foliage and grapes makes this task significantly easier (see Figure 2.1).

For what concern the second goal of the project *i.e.* a major commitment about the acceptance of the robotic system in a field that's still strongly led by traditional methods, its realization consists mostly in the creation of an user-friendly interface (possibly for smartphones or tablets) for the GRAPE system, that allows an user to:

- constantly and easily visualize the position and activity of the robot
- perform supervisory control on the platform *e.g.* selection of the plants for the dispensers deployment
- in case of a failure of the UGV in an autonomous task, provide a teleoperation interface to carry out the operation

The whole project was also to be designed with and adequate degree of modularity, in order to naturally support the possible extension to a fleet of robot able to operate in parallel. Note that, as described in Section 1.1, this goal gets very easy by the usage of ROS framework.

The GRAPE projects involved three partners, each with different assignments, and different responsibilities in the project context: **PoliMi**, **Eurecat** (a technology center located in Catalonia, that operates in many industrial and research fields, including robotics), and **Vitirover** (an agricultural robots company, located in France). To make the different teams interact and test the compatibility of the different parts of the system, some communal sessions of integration and tests, that we'll call *integration weeks*, have been carried out within the duration of the whole project. In this thesis we are going to focus mostly on the experiments and results carried out during the last GRAPE integration week, held in March 2018.

(a) (b) (c)

Figure 2.3: *Different shape of pherormone dispensers available on the market. The model used in GRAPE is the one depicted in image 2.3c*

(a) (b)

Figure 2.4: *Other field robots developed in EU projects in the last 6 years: 2.4a Vinerobot, 2.4b Vinbot.*

CONTEXTUALIZATION OF THESIS WORK IN THE PROJECT

Since the timespan of the GRAPE project as a whole covers 18 months, we took advantage of the preceding work described in Serrano et al., 2017, that is a description of the ongoing work around the GRAPE project. As stated in the paper, some results were already accomplished in a few subjects:

- in-depth analysis of the main challenge and obstacles in the project, and requirement extraction
- identification and obtainment of the specific hardware components (mobile platform, robotic arm, LIDARs, IMUs, cameras, network equipment, computational unit)
- segmentation of the macro objectives into smaller subtasks
- results of intermediate experiments carried out during the first integration week in Garriguella, Spain. The experiments mostly concern the mapping, localization and navigation tasks

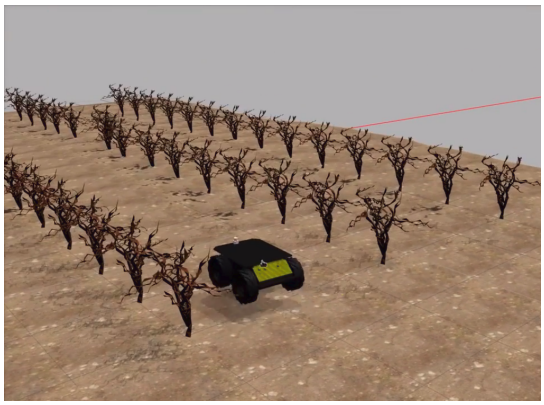


Figure 2.5: *Simulation of the Husky robot in the vineyard environment. Simulation built using Gazebo simulation software.*

Figure 2.6: *The 3D printed vine tree mockup, used in validation phase of the dispenser application task algorithm.*

- creation of a simulated vineyard and Husky environment using Gazebo simulation tool (see Figure 2.5)
- collection of data generated during the same integration week, making use of ROS data logging tool (as described in Chapter 1.1)
- creation of a 3D printed mockup vine tree (see Figure 2.6), to make the laboratory work more efficient and meaningful for the development and validation of the parts that would interact directly with the vine trees (*i.e.* dispenser application task). The mockup was created from the laser scans of a real vine tree in Figueres (ES).

The problems tackled in this thesis are a subset of the ones assigned to PoliMi in the project proposal. The problems we faced are the following, and will be described in depth in next chapters:

- design of the overall software architecture in the ROS framework (see Chapter 3)

(a) (b)

Figure 2.7: *Other field robots developed in EU projects in the last 6 years: 2.7a Rhea, 2.7b CROPS.*

- revision and enhancement of the mapping, localization and navigation systems (see Chapter 4)
- design and part of the implementation of the pherormone dispenser application (see Chapter 5)

RELATED PROJECTS

We are now presenting a short list of EU projects related to robotics for precision farming, with direct or indirect application to vineyards. All these projects have been developed in the last 6 years, and demonstrate the growing interest of the scientic community and potential end users to this application and research field.

VINEROBOT Automated measurement and monitoring of parameters such as grape yield, vegetative growth, water status and grape composition in vineyards by use of an UGV (see Figure 2.4a) endowed with artificial intelligence with sensing technologies. Chlorophyll-based fluorescence, RGB machine vision and thermography technologies are used to monitor vineyard on-the-go (Diago, Tardaguila, et al., 2015).

VINBOT Creation of an all-terrain autonomous mobile robot (see Figure 2.4b) with a set of sensors capable of capturing and analysing vineyard images togethere with 3D data, by means of cloud computing applications, to determine the yield of vineyards and to share information with the winegrowers, in order to mix the grapes of homogeneous quality to efficiently market a range of wines by quality and price. (Lopes et al., 2017).

RHEA Design, development, and testing of a new generation of automatic and robotic systems for both chemical, mechanical and thermal effective weed management focused on both agriculture and forestry, and covering a large variety of European products including agriculture wide row crops and forestry woody perennials. RHEA aims at diminishing the use of agricultural

chemical inputs in a 75%, improving crop quality, health and safety for humans, and reducing production costs by means of sustainable crop management using a fleet (see Figure 2.7a) of heterogeneous robots, both ground and aerial, equipped with advanced sensors, enhanced end effectors and improved decision control algorithms (Santos, Ribeiro, and Fernandez-Quitanilla, 2012).

CROPS Design and development of an highly configurable, modular and clever carrier platform that includes modular parallel manipulators and intelligent tools (sensors, algorithms, sprayers, grippers) for high value crops like greenhouse vegetables, fruits in orchards, and grapes for premium wines. The CROPS robotic platform (see Figure 2.7b) is capable of site-specific spraying (targets spray only towards foliage and selective targets), selective harvesting of fruit (detects the fruit, determines its ripeness, moves towards the fruit, grasps it and softly detaches it), reliable detect and classificate obstacles and other objects to enable successful autonomous navigation and operation in plantations and forests (Oberti et al., 2014). Further development in CROPS projects also includes Sweeper project (Ringdahl et al., 2016).

3

GRAPE HARDWARE AND SOFTWARE ARCHITECURE

In this chapter we are going to describe in a detailed way the architecture of the robotic platform we used to implement the system to fulfill the requirements described in Section 2.1. First, in Section 3.1, we are giving an overview of the hardware components, from the robot base to all the actuators/sensors present on board. In Section 3.2, we'll start to analyze the software architecture of the system, explaining what the main modules are, and the relationships between them.

GRAPE HARDWARE ARCHITECTURE

Robotic Base

The choice of the robotic base is, of course, a crucial point in the development of our system, for two reasons:

- without a well-functioning robotic base, no other objective can be achieved
- the vineyard environment is particularly challenging in nature (this particular aspect will be well-justified in Chapter 4), thus a non-prudent choice could be very dangerous.

As anticipated in Section 1.3, the choice fell on the **Husky** UGV from Clearpath Robotics[1], (see Figure 1.10). There are several reasons for the adoption of this specific model:

- it's a widely used platform (*e.g.*: Madhavan et al., 2015; Hinkel et al., 2015; Ostafew, Schoellig, and Barfoot, 2013)
- it has a *skid steering* kinematics that, as already seen in Section 1.3, can be essentially reduced to a *differential drive* kinematics, that is very simple.
- it's specifically designed for outdoor use, so robustness and ability to deal with unstructured terrain are problems addressed by its rugged construction and high-torque drivetrain.
- it offers a large payload capacity, to host the multitude of sensors, actuators and computational units we need

1 https://www.clearpathrobotics.com/husky-unmanned-ground-vehicle-robot/

Figure 3.1: *Husky platform after a navigation session in environment. You can easily note the amount of stones and mud stuck into the wheels.*

- it's fully supported in ROS, with open source drivers, configuration files and examples

This choice turned out to be substantially good. The wheels odometry turned out to be better than expected (TODO grafico della posizione xy dell'odometry delle ruote vs grafico della posizione stimata con robot localization), and the ROS drivers and config files were actually pretty easy to use and modify. The only major weakness identified in the Husky is related to its behavior on muddy ground, and is due to the wheels shape. After a few minutes of navigation in the mud, the UGV tends to collect a significant amount of earth and stones in the wheels groves (see Figure 3.1) and, consequentely, lose grip on the ground. This problem could be attenuated or solved by use of tracks instead of wheels.

Robotic Arm

By observing how pherormone dispensers must be deployed on the vine plant in Figure 2.3 (in Figure 2.3c, you can see the dispenser shape that we actually used for GRAPE project; in Figure 2.2, a close-up of our dispenser type), it's easy to understand that we require a very flexible robotic arm, with advanced movement capacity and capable of precise movement. Additional constraints also come from the limited space available on the Husky base, and from the limited power supply at disposal aboard. The choice fell on *Jaco*^{2} arm from Kinova Robotics[2] (see Figure 3.2a, Table 3.1 for product specifications), for these reasons:

- 6 DOF provide the required flexibility
- 3 fingers can be used for the dispensers deployment

2 http://www.kinovarobotics.com/wp-content/uploads/2017/06/JACO%C2%B2-User-Guide-Asstive-Robotics-April-2017.pdf

PARAMETER	VALUE
Degrees of Freedom	6
Weight	5.2 Kg
Payload	1.3 Kg
Reach	90 cm
Maximum Linear arm speed	20 cm/s
Power supply	$18 \div 29$ VDC
Average Power	25W
Peak power	100W
Communication Protocol	RS485
Material	Carbon fiber

Table 3.1: Product specifications of Jaco^2 from Kinova Robotics.

- while programmatic control is necessary for GRAPE purpose, the joystick control make it easy to perform quick tests and analyze the robot capabilities
- unlimited joints rotations
- provided mechanical support for sensors mounting on top of the arm
- reduced weight and power consumption
- small overall dimensions
- ROS open source drivers already provided by Kinova Robotics

Some of the nice features of this device (weight, dimensions) actually derive from the original purpose of the arm, that is in the context of assistive devices for disabled people, to improve quality of life and independence of people on power wheelchairs (see Figure 3.2b). But this fact also brought some unexpected weaknesses, that introduced a series quite serious problems that slowed down the development process. The identified weaknesses are:

- a very large imprecision in the movements piloted via programmatic control: this is probably due to joystick control seen as a "main control mode" since, as previously stated, the arm is born as an assistive device. This problem was partially solved modifying the open source drivers of the arm, but entire control strategies were introduced to compensate this weakness. The magnitude of the error, measured as distance between the desired cartesian position of the end effector and the actual one, is

(a) (b)

Figure 3.2: *Robotic arm Jaco^2 with 3 fingers, from Kinova Robotics (3.2a) and the same arm mounted on a wheelchair 3.2b*

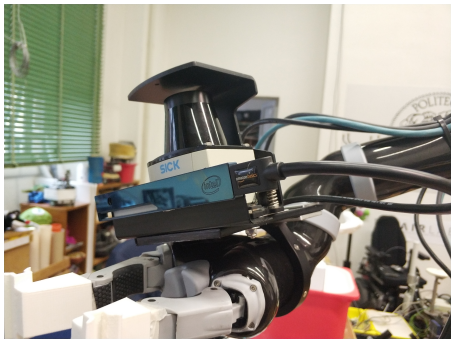


Figure 3.3: *The final configuration of the sensors (LIDAR and RGB-D camera) mounted on top of the end effector.*

$0 \div 5$ cm after the changes to the ROS driver of the arm. TODO sarebbe bello avere dei dati per supportare questa misura

- buggy implementation of joints constraints, probably due to the same cause of the previous point. This created a major problem mostly because of the wiring used for the sensors (described in next Sections) mounted on the arm, that kept rolling up because of the unlimited joints rotation capability. TODO a seconda di come alla fine la risolviamo definitivamente (messi a posto i driver, o procedure di controllo) lo scrivo.

After the analysis of the macro goals of GRAPE project (Section 2.1), it was decided to use the arm as a support for other sensors, as well as manipulation tool, for the dispenser deployment task: a LIDAR and an RGB-D camera (see Image 3.3).

	VLP-16	TIM561	UTM-30LX-EW
Number of Channels	16	1	1
Scan Angle	360°	270°	270°
Rotation rate	5Hz 10Hz 20Hz	15 Hz	40Hz
Angular Resolution	0.1° 0.2° 0.4°	0.33°	025°
Range	100m	10m	30m
Power Consumption	8W	4W	<8W
Weight	590g	250g	210g

Table 3.2: Comparison of onboard LIDARs

LIDARs

The final robotic system mounts three different LIDARs: two in front of the Husky for visualization and navigation purposes, and the other one mounted on a metal support on top of the arm end effector. The purpose of this last LIDAR will be cleared in Chapter 5; for now, all you need to know is that it's used in the procedure of identification of the most suitable point for dispenser deployment. The LIDARs come from different producers, but they are quite similar being designed for outdoor use. The two used for navigation are **Velodyne VLP-16**[3] and **Hokuyo UTM-30LX-EW**[4], the one on top of the arm is **SICK Tim561**[5]. Main differences and similarities between these models are highlighted in Table 3.2

RGB-D camera

The other sensor mounted on top of the arm end effector is an **RealSense** RGB-D camera from Intel. This camera is able to sense the depth of the acquired images (see Figure TODO screenshot di rgb e profondità), by means of an active infrared stereo technology. The main usage of the camera is to provide images used for the feedback loop of the visual servo control of the Jaco[2] arm. Visual servoing is a technique which uses feedback information extracted from a vision sensor to control the motion of a robot (Hutchinson, Hager,

3 http://velodynelidar.com/vlp-16-lite.html
4 https://www.hokuyo-aut.jp/search/single.php?serial=170
5 https://www.sick.com/de/en/detection-and-ranging-solutions/2d-lidar-sensors/tim5xx/tim561-2050101/p/p369446

(a) (b) (c)

Figure 3.4: *Three LIDARs mounted on our UGV: Tim561 (3.4b) from SICK, Puck Lite (3.4a) from Velodyne, UTM-30LX-EW (3.4c) from Hokuyo.*

and Corke, 1996), and it's used in this project to compensate the lack of precision in the arm motion operations (see Section 3.1.2). This camera also applies to operation validation operations (validation of dispenser grasping, validation of dispenser deployment on the plant) and identify exceptional cases (no dispenser left). The positive aspect of this camera model comes from its contained size and weight. We spotted only one significant weak point, that is a quite high minimum depth distance (20 cm), due to an intrinsic limit of the embedded depth technology. This problem created a few complications in dispenser application tasks, but they were easily bypassed.

Tools for dispensers grasping

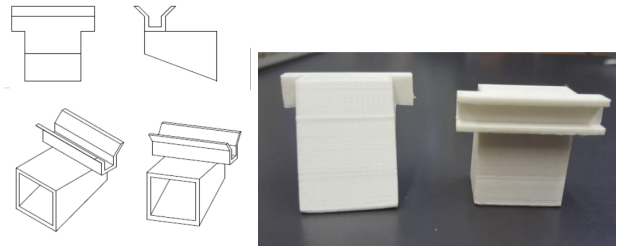
Additional hardware was placed on the UGV specifically for the accomplishment of the grasping of the dispenser, for two separate reasons:

- the shape of the dispenser (see Figure 2.2) was not directly compatible with the grasping capability of the end effector of Jaco^2 arm (see the Jaco^2 fingers in Figure 3.2a)
- the dispensers have to be stored on the Husky platform, in such a way to make them graspable

We required very custom hardware to accomplish these objectives so, given the prototype nature of the project, 3D stamped hardware was used. For what concern the storage of the pheromone dispensers, the final version of the dispenser feeder was built out of two 3D-printed small towers (see Figure 3.5a), with a sequence of very smooth aligned grooves, parallel to the horizontal plane, where the dispensers are slotted in. The smoothness of the grooves is the critical part of the design, indeed the previous versions of the towers, with sharper grooves, highlighted a difficulty of the grasping phase if the dispensers got caught on the feeder's corners.

(a) (b)

Figure 3.5: *The final version of the 3D-printed dispenser feeder (3.5a), and a perspective drawing of one of the earlier versions of it (3.5b), with sharper edges.*



(a) (b)

Figure 3.6: *The final version of the 3D-printed nails (3.6b), and their perspective drawing (3.6a)*

The fingers of the robot are not all the same, because one of them act as a sort of "opposable thumb", that is useless in our application. So, two "nails" were printed to be applied to the two symmetric fingers, with a side dispenser-sized groove each, in order to grasp the dispenser by opening the two symmetric fingers.

Other sensors

The UGV holds onboard a collection of other sensors, useful for the accomplishment of the established tasks. In this section we'll give a short overview of the few remaining ones:

IMU SENSOR Device including a magnetometer, accelerometer, gyroscope. These sensors were used in the odometry estimation system through sensor fusion algorithms (see Chapter 4)

GPS Also this sensor data stream was included in the odometry estimation system

CAMERAS Additional camera mounted in a slightly lifted position with respect to the robot base; they are useful in case of teleop-

Figure 3.7: *The complete robotic platform, with sensors/actuators final arrangement.*

eration of the Husky or the robotic arm, and for visualization and monitoring purposes.

You can see the final sensor/actuators arrangement in Figure 3.7 TODO foto migliori

GRAPE SOFTWARE ARCHITECTURE

Now that the hardware configuration is clear, we'll focus on the *behavior* of the UGV. We'll start by a description of the subproblems that we identified starting from the macro goals we talked about in Section 2.1, and then move to the software architecture.

Evaluation of vineyard navigation

The problem of autonomous navigation of UGVs is a very well-known problem, studied at least from the nineties (Gage, 1995), because of course it's the first problem to deal with in almost all systems involving mobile robots. For this reason, and being our robotic system a prototype, for us the problem of vineyard navigation reduced to the following subproblems:

- identification of a sensor fusion framework in ROS, to provide an odometry estimate, and configuration of such a framework, with particular attention to which sensors fuse together
- identification of a navigation framework in ROS, to provide motion planning and speed control of the mobile base, and configuration of such a framework to fit our problem

Even if these tasks seem trivial, two major problems arose:

- complex framework like these ones have tenths of parameter to tune, and the documentation is not always very precise
- the *off-the-shelf* navigation systems are usually born for indoor environments, that differ a lot from the very unstructured nature of the vineyards.

As hinted in Section 1.5 and 1.4.1, our choices fell on **Move Base** package for the navigation, and **Robot Localization** as odometry system. We'll present more detail of the subproblems mentioned above in Chapter 4. Move Base offers an *action* (see Section 1.1) as interface with the ROS ecosystem.

Evaluation of vineyard monitoring task

For the vineyard monitoring task, as foreseen by the project proposal, we opted for a semi-autonomous monitoring: we provided the UGV with a camera with video stream capability, that allows human users with competences in the domain of plants health to observe in person the live data streamed by the robot for assessment of crop condition. Human operators are also provided teleoperation interface for navigation and manipulation, for supervisioning and emergency handling function.

This choice came from both the absence in the project proposal of precise objectives on this topic, and the complexity of feature extraction and evaluation in this context.

Evaluation of dispenser application task

While navigation task is a very common problem and *off-the-shelf* solutions are easy to find, dispenser application is a completely non-standard task, so we could only rely on libraries and package for low level subtasks but we had to design and implement the whole procedure from scratch. We divided the procedure in 3 subproblems:

1. **Creation of a point cloud of the tree:** the robot is sitting still, with a vine on the same side of the Jaco^2 arm. When the procedure is triggered, the arm moves in order to produce a point cloud of the trees close to it, using the LIDAR mounted onto it.
2. **Deployment points identification:** once the point cloud has been produced, an algorithm based on Point Cloud Library (PCL) trims the point cloud from the points that don't belong to the tree, process it, and outputs a sequence of 3D points (x, y, z) corresponding to points on the tree suitable for the deployment of the dispenser, typically a protuberance of the tree, ordered

from the best one to the worst one according to a suitable metric.

3. **Deployment of the dispenser:** the arm starts the dispenser deployment procedure, taking as input the sequence of points got in the previous step:
   a) it grasps the dispenser (if any) from the dedicated support
   b) it applies the dispenser on the best feasible point from the input sequence
   c) it goes back to a retired position, to achieve better stability during the movement of UGV.

   In figure 3.10, you can see an example of successful dispenser application, using the mockup plant. **NOTE:** the integration weeks were not all held in the same place (two of them in Garriguella (ES), and one in Casciana Terme (IT)), so we could observe that the vine trees shape is very different in the different locations (for example, the trees in Garriguella tend to have small almost vertical branches on the top of the pruned tree, while the Italian ones had smaller protuberances with variable orientation). This factor makes our system much harder to validate, because it's even more difficult to identify useful features in the unstructured vineyard environment. Thus, we introduced a further deployment mode, to be used in case our deployment procedure on the plants turned out not versatile enough to be efficient on a new tree type. Since we wanted to bypass the problems due to different tree structures, in this mode we assume to have some appropriately shaped nails, expressly arranged for dispenser applications. As we'll see in Chapter 5, this deployment mode also requires the application of *binary square fiducial markers* (see Figure 3.8), so in this way our application pushes a bit more towards a vineyard that modifies in order to be more suitable to robotic applications.
   The steps of this deployment mode are slightly different from the previous ones:
   a) it grasps the dispenser (if any) from the dedicated support
   b) it applies the dispenser on the nail, regardless of the input sequence (which become useless)
   c) it goes back to a retired position, to achieve better stability during the movement of UGV.

All these procedures are quite long and complex, we wanted the user to have a high control on them, including the monitoring of intermediate results. For these reasons, these procedure are implemented as ROS actions.

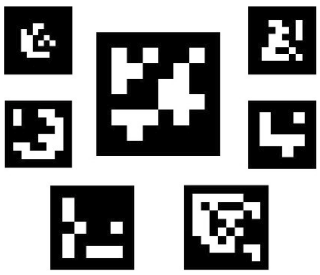


Figure 3.8: *Examples of visual markers (more specifically, ArUco markers) as the ones we used in visual servoing applications.*

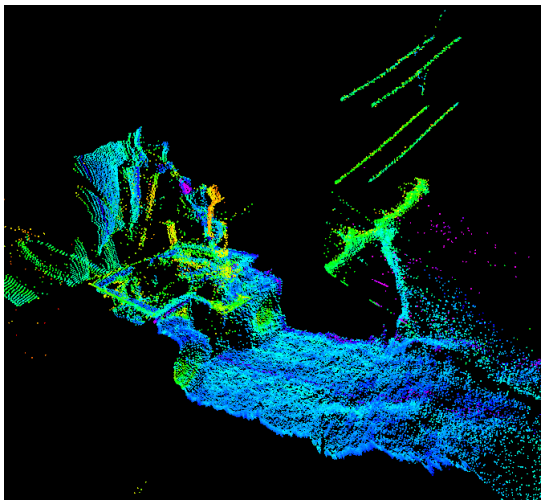


Figure 3.9: *Examples of laser scans collected in Casciana Terme during the integration week in January 2018, with the LIDAR mounted on top of the end effector of the arm. You can recognize the vine plant, and part of the Husky base. The colors are only for visualization purpose.*

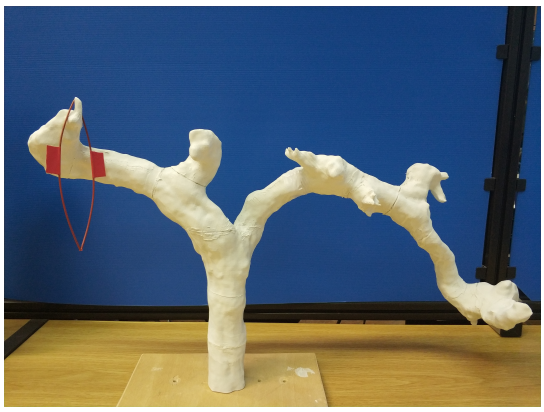


Figure 3.10: *Example of a dispenser application task, correctly executed on a mockup plant.*

Details of the actions interfaces

In this section we are giving a detailed description of the action interfaces, to better understand their scope and goals. Note that none of them includes a flag to check for correctness of execution, because we decided to use the internal status variable embedded in ROS actions.

Move Base action interface

- **Goal Message**:
  - *target_pose*: the pose (position, orientation) we want our robot to reach
- **Result Message**:
  - *base_position*: the final pose (position, orientation) of the robot; in case of success, it should be very close to the specified *target_pose*
- **Feedback Message**: *n.a.*

Scan motion action interface

- **Goal Message**: *n.a.*
- **Result Message**: *n.a.*
- **Feedback Message**: *n.a.*

Note that even an empty ROS action like this makes sense, because of the length (~30 seconds) of the procedure triggered by an action call.

Point cloud processing action interface

- **Goal Message**: *n.a.*
- **Result Message**:
  - *deployment_pose*: contain an array (x, y, z) of possible deployment poses, identified from the plant point cloud; it will be used as input to the dispenser deployment action server (see Table 3.5)
- **Feedback Message**: *n.a.*

Dispenser deployment action interface

- **Goal Message**:
  - *deployment_poses*: sequence of feasible deployment points, ordered from the best one to the worst one. These are the points output by *Point cloud processing action*

Goal Message	`geometry_msgs/PoseStamped target_pose`
Result Message	`geometry_msgs/PoseStamped base_position`
Feedback Message	`---`

Table 3.3: Move Base action specification

Goal Message	`---`
Result Message	`---`
Feedback Message	`---`

Table 3.4: Scan motion action specification

  - *deploy_mode*: an integer, used to select the technique to be used for the dispenser deployment; in fact, three different mode has been developed, in order to fit the different condition that the UGV could face. In short, the modes available for selection are:
    * Closed loop control, deployment target is an expressely placed nail, hammered into a pole
    * Closed loop control, deployment target is a protuberance of the vine tree
    * Open loop control, deployment target is a protuberance of the vine tree

    Details about the deployment mode are given in Chapter 5.

- **Result Message**: *n.a.*
- **Feedback Message**:
  - *ManipulationStatus status*: a message of proprietary type, that contains structures used to communicate the action caller the current execution state (*e.g.* dispenser fallen, arm moving in front of the target point).

Goal Message	`---`
Result Message	DeploymentPose[] deployment_poses
Feedback Message	int8 execution_percent range [0,100]

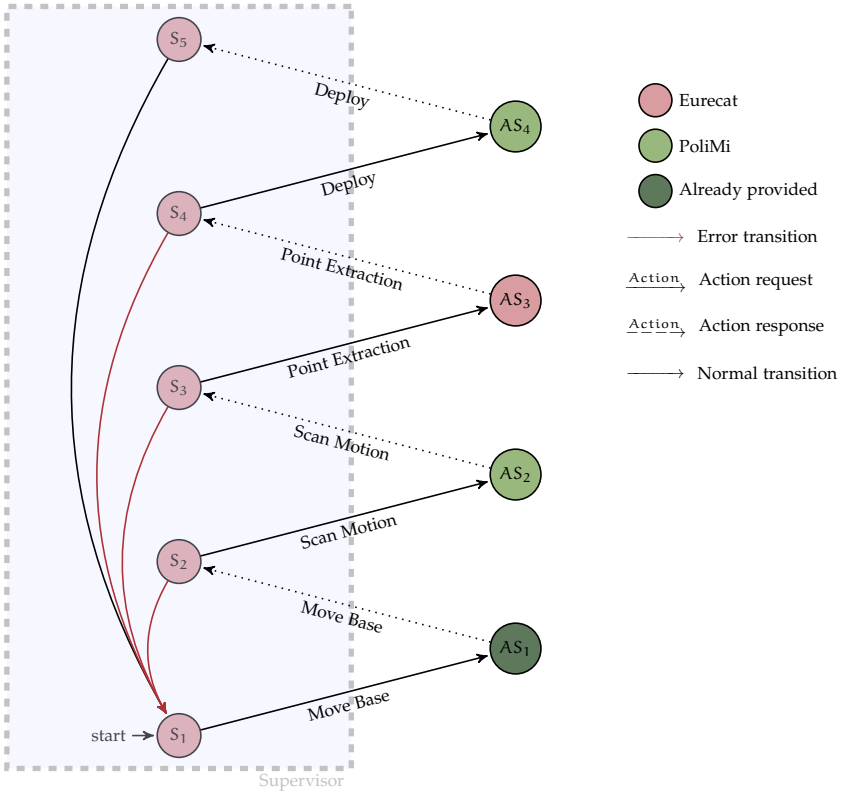
Table 3.5: Process point cloud action specification

Goal Message	DeploymentPose[] deployment_poses int32 deployMode
Result Message	- - -
Feedback Message	ManipulationStatus status

Table 3.6: Dispenser Deployment action specification

The supervisor

Since all main operations in the project structure are implemented through action servers, of course a list of action clients must exists. For our implementation, we opted for a single software module that acts as an action client for all the actions mentioned above. We call this module *Supervisor*, and its implementation uses behavior trees (Colledanchise and Ögren, 2017) as an overall model. The objective of the supervisor is to implement the global Finite State Automata (FSA) of the system, where basically **nodes** are action servers or action clients.



As pointed out by Figure 3.2.5, all action clients belong to the supervisor software module, and major tasks are implemented through action servers. In that graph we also specify the division of labor between the different teams:

- Software modules entrusted to **PoliMi**
  - AS_2: Scan motion action server
  - AS_4: Dispenser deployment action server
- Software modules entrusted to **Eurecat**
  - AS_2: Point cloud processing action server
  - S_i: Supervisor (*i.e.* all action clients and FSA implementation)

Software modules treated in this thesis

After this global overview of the hardware and software components of the GRAPE project as a whole, we can finally specify which of the aforementioned topics has been directly treated in this thesis work. They are:

- UGV navigation system (see Chapter 4)
- development of Scan motion action server (see Chapter 5)
- development of part of Dispenser deployment actions server (see Chapter 5).

4

LOCALIZATION AND NAVIGATION SYSTEMS IN GRAPE

The studies of Singhal, 1997, that analyze main issues in autonomous mobile robot navigation, identify three main problems linked to outdoor autonomous navigation:

- the unstructured environment where the actions take place, in opposition to the clear and definite one that is typical in indoor navigation (flat floor, right angles, smooth surfaces)
- requirement for multiple sensors, to be aware enough of the surrounding environment, in order to take decisions
- moving obstacles, like for example pedestrians or cars in a city road

The system developed in the context of GRAPE project suffers critically mostly of the first two point listed above. Indeed, the vineyard environment, even if characterized by a certain degree of regularity due to the presence of parallel rows of trees, presents strongly difficulties regarding lack of structure in the terrain configuration (steep, bumpy, clay-rich, muddy, according to the weather conditions and the intrinsic composition and morphology of the terrain) and in the obstacles (non-straight surfaces, different reflectivity and lighting conditions). Moreover, the problem of the fusion of the data coming from the disparates sensors mounted on board of the robot was already addressed from a theoretically point of view in Section 1.4, and it's the major component in the localization system of the robot. Note that, being our robot an agricultural UGV, the problem of moving obstacles is not particularly relevant, since they are most likely to be humans, aware of the robot presence and functionalities, or at most small wild animals (*e.g.* foxes, hares, cats) in remote cases, that however have a very low probability of approaching the Husky base.

In the next section we are going to analyze with detail the configuration of both the odometry and navigation systems running on the UGV. It will be clear that both the problems, even if eventually solved using *off-the-shelf* solutions, were all but simple problems and required a real understanding of the numerous problems that arose.

ODOMETRY SYSTEM

The configuration of a robust odometry system for vineyard navigation has been a workpoint since the beginning of the project, because

of the strong instability of the wheel odometry. To the reasons mentioned at the beginning of the Chapter, we recall also the intrinsic lack of precision due to skid steering kinematics of the Husky base. We can describe the search for a robust odometry system for the UGV in three main phases:

1. As hinted in Section 1.4, in the early phases of GRAPE project, the designated sensor fusion framework was **ROAMFREE**, that had given proof of good functioning in the past (see Section 1.4 for precise references). ROAMFREE platform provides multi-sensors pose tracking and it is designed to be flexible and to adapt to every kind of mobile robotic platform. The tracking module is based on Gauss-Newton minimization of the error functions associated to sensor readings. In order to improve generality, physical sensors are abstracted with *logical sensors*, which are characterized only by the type of the proprioceptive measurement they produce. An other ROAMFREE module provides instead self-calibration modules, using error models for each sensor category to provide on-line correction of the common sources of distortion, bias and noise (*e.g.* hard and soft magnetic distortion, sensor displacement or misalignment). The usage of ROAMFREE framework was specified in the project proposal, thus it was of course the default choice for the sensor fusion framework in the first *integration week* in Garriguella (ES), before this thesis work beginning. The ROS implementation of ROAMFREE is still experimental, but a configuration was identified to fit the requirements. Unfortunately, the same configuration turned out to perform poorly during the second integration week in Casciana Terme (IT). This was caused by concurrence of:
   - presence of a significant slope in the vineyard, in opposition to the flat (even if bumpy) terrain in Garriguella
   - difficulty to handle inaccurate configuration of magnetometer and IMU
   - overall trickiness in the configuration procedure, being ROAMFREE integration with ROS still experimental

   The low performances of this framework in this context were pretty clear, so we opted for a more tested and consolidated solution, **Robot Localization**.

2. The configuration of Robot Localization requires, for each fused sensor, a configuration vector, in which specify which components of the input estimate should be fused into the final pose estimate. Note that the the configuration vector is given in the *frame_id* of the input message: for example, consider a velocity sensor that produces a *geometry_msgs/TwistWithCovarianceStamped*

message with a *frame_id* of *velocity_sensor_frame*. If we assume that the transform would convert X velocity in the *velocity_sensor_frame* to Z velocity in the *base_link_frame*. To include the $\dot{X}$ data from the sensor into the filter, the configuration vector should set the $\dot{X}$ velocity value to true, and not the $\dot{Z}$ velocity value. In table 4.1 you can see the configuration vectors for the sensors taken as input from Robot Localization. Note that, for example, GPS only gives a contribute about x and y, since the altitude estimation of GPS is often not precise, and it provides no information about the robot orientation. Our configuration of Robot Localization relied on **two** different Extended Kalman Filter (EKF) nodes in cascade:

- a *local* localization node, that fuses continuous data, and estimates the position and orientation of the *base_link* frame of the robot inside the *odom* frame *i.e.* the frame fixed in the initial position of the robot. In other words, the task of the node is to publish *odom* $\rightarrow$ *base_link tf* transform (see Section 1.2 for recall).
- a *global* localization node, that also fuses low frequency sensors *i.e.* GPS, and estimate the estimates the position and orientation of *odom* frame inside *map* frame. The *map* $\rightarrow$ *odom* trasfomation "adjusts" the position of frame *odom* in order to contextualize the position of the UGV inside the map. Otherwise, the robot pose would be tracked only with respect to its initial position (origin of *odom* frame), but no positioning inside a map would be provided.

In Figure 4.1 you can see the computation graph of the aforementioned odometry system: you can observe that:

- both nodes write on *tf* topic the respective *tf* transform
- the local node takes IMU (*/imu/data* topic) and wheels (*/husky_velocity_controller/odom* topic) estimate for the fusion task
- the global node takes as input the pose estimate from GPS (*/odometry_gps* topic), and the pose estimate from the local node (*/odometry/ekf_local_filtered* topic), to implement the cascade.

3. Even if the cascade produced a pretty precise odometry estimate, the odometry computation was shrunk in a single EKF node for two main reasons:
   - as we'll describe in Section 4.2, the navigation system switched to a mapless configuration, so the *tf* transform *map* $\rightarrow$ *odom* became useless
   - the EKFs cascade configuration led to a short delay in the odometry computation (~200-300 ms). This caused some

	x	y	z	ψ	θ	ϕ	$\dot{x}$	$\dot{y}$	$\dot{z}$	$\dot{\psi}$	$\dot{\theta}$	$\dot{\phi}$	$\dot{\theta}$	$\ddot{x}$	$\ddot{y}$	$\ddot{z}$
Wheels	✓	✓	✓			✓	✓	✓	✓							
IMU				✓	✓	✓					✓	✓	✓	✓		
GPS points	✓	✓														

Table 4.1: Configuration vectors for the sensors fused for odometry estimation with Robot Localization.

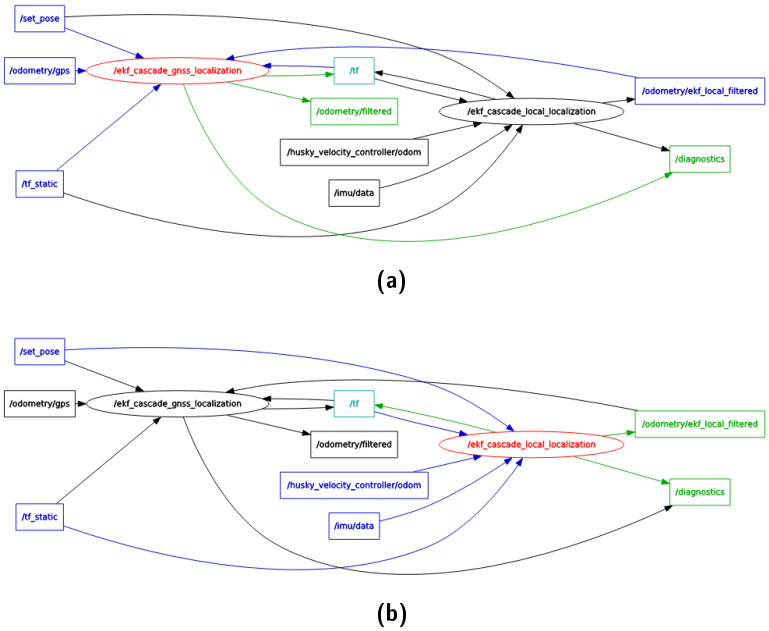


Figure 4.1: *Computation graph of the cascade of Robot Localization nodes; in blue, the input topics, and in green the output topic, of the nodes highlighted in red.*

problems with the obstacle detection implemented through LIDAR data, that are processed with negligible delay, and especially when the robot moved from the end of a row of vines to the beginning of another one.

NAVIGATION SYSTEM

We already anticipated in Section 1.5 that the reference navigation framework is **Move Base**, that is an *off-the-shelf* ROS software stack for autonomous navigation (see Figure 1.16 for a graphical representation of the main software modules of Move Base). Since, as already stated, this framework usually provides very stable autonomous navigation in indoor mapped environment, the first attempt was to shift the same configuration in the vineyard environment. In this Section

we'll explain the steps from this initial naive solution to the final accepted one.

Navigation in mapped environment with AMCL

Before the robot could be able to navigate the vineyard, this configuration required a preparatory *mapping* phase; in particular, this is a Simultaneous Localization And Mapping (SLAM) problem, that is the computational problem of constructing or updating a map of an unknown environment, while simultaneously keeping track of an agent's (the UGV) location within it. Note that *localization* is a different task with respect to *odometry estimate*, because the first aims to estimate the position of the robot inside a map, while the latter aims to estimate the position of the robot with respect to a reference frame. Even if it is a *chicken-and-egg* problem (localization is easy in a known map, under some assumptions, and mapping is easy if the robot position is assumed known), several algorithms exist that solve, even if approximately, the SLAM problem. In the early phases of GRAPE, previous to this thesis work and described in Serrano et al., 2017, three different SLAM algorithms were tested in our specific environment: *Gmapping*, *Google's Cartographer* and *KartoSLAM*, and **Gmapping** algorithm was selected because of its performance and reactivity. In Figure 4.3 you can see the map created in Mas Llunes during the last integration week, using Gmapping. The accomplished accuracy of the maps was satisfying, but actually a lot of problems occurred in the main phase, that is the autonomous navigation of the robot in the mapped environment. The main problems that emerged were the following ones.

AMCL IN THE VINEYARD MAP

As you can see in Figure 1.16, an optional node called **amcl** exists, that provides a ROS implementation of AMCL (Thrun et al., 2001). AMCL is a localization algorithm, based on a particle filter to provide an estimate of the *belief i.e.* the robot's estimate of its current state through a probability density function distributed over the state space. A single particle represents an hypothesis about the real pose of the robot in the environment (see Figure 4.4); thus, regions in the state space with many particles correspond to a greater probability that the robot will be there, and regions with few particles are unlikely to be where the robot is.

In AMCL each particle is assigned a weight that correspond to the probability that, had the robot been at the state of the particle, it would perceive what its sensors have actually sensed (in particular, *amcl* ROS package is only capable to deal with laser scans as sensor input). For this reason, the algorithm requires both a sensor model of the used sensors, and a motion model

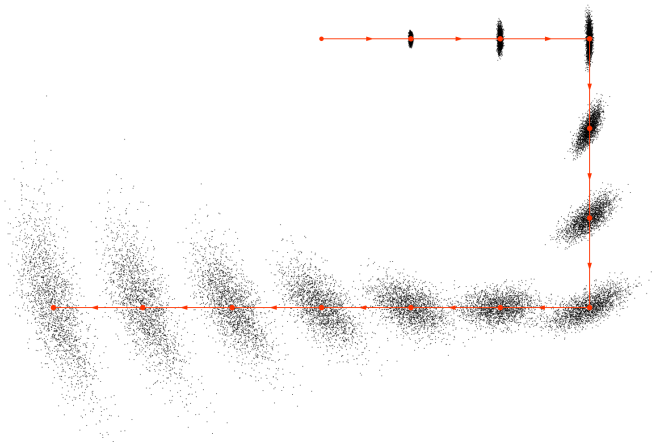


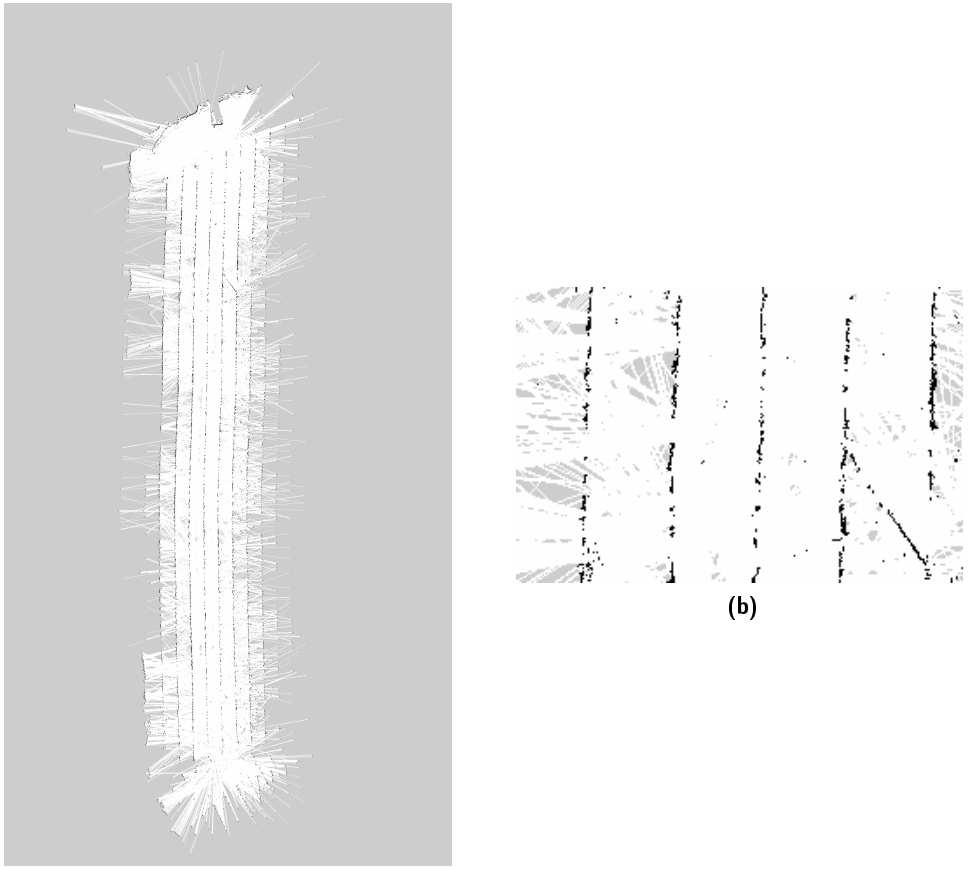
Figure 4.2: *Evolution of the belief estimated through AMCL, only taking into account the motion model.*

of the involved robot, in order to update the belief. In Figure 4.2 you can see several steps of *belief* update, only based on the motion model of the robot. Pseudo code of Monte Carlo Localization is provided in Algorithm 2.

Algorithm 2 Monte Carlo Localization(X_{t-1}, u_t, z_t)

1: $X_t = \emptyset$ ▷ X_t is belief about the robot state
2: $\overline{X}_t = \emptyset$
3: for $m = 1$ to M:
4: $\quad x_t^{[m]} = \text{motion\_update}(u_t, x_{t-1}^{[m]})$ ▷ u_z is actuation command
5: $\quad w_t^{[m]} = \text{sensor\_update}(z_t, x_t^{[m]})$ ▷ z_t are sensed data
6: $\quad \overline{X}_t = X_t + \langle x_t^{[m]}, w_t^{[m]} \rangle$
7: endfor
8: for $m = 1$ to M: ▷ M is number of particles
9: $\quad$ draw $x_t^{[m]}$ from $\overline{X}_t$ with probability $\propto w_t^{[m]}$
10: $\quad X_t = X_t + x_t^{[m]}$
11: endfor
12: return X_t

AMCL with laser scanners provides a very effective localization in feature-rich environments, like in typical indoor environments with very sharp corners and well defined obstacles, as showed in Figure 4.4. However, as hinted at the beginning of this Section, a few tests were sufficient to show the weakness of this localization algorithm in our context, due to the high repetitiveness of the vineyard map. In Figure 4.3b you can see a magnified detail of the map of Mas Llunes vineyard, computed using Robot Localization as a odometry system, and Gmapping for SLAM; most of the obstacles detected by the frontal LIDAR are the trunks of the trees, together with poles and high weeds. It's



(a) (b)

Figure 4.3: *Map of the Mas Llunes vineyard in Garriguella (ES), computed during the last integration week using Robot Localization and Gmapping. In 4.3b, a magnified detail.*

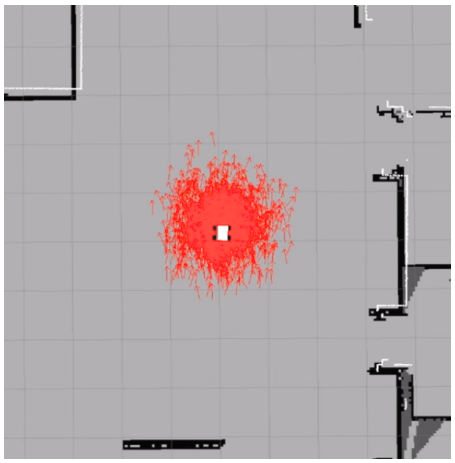


Figure 4.4: *Belief about the robot estimate using AMCL, through a particles cloud where each particle represents a hypothesis about the real pose.*

intuitive that if the map in which the robot moves presents a lot of repeated patterns, the *survival of the fittest* paradigm that the *sensor update* step of AMCL should implement, is not very effective. Moreover, note that the bumpiness of the terrain leads the plan scanned by the LIDAR to change significantly its orientation in time, so for example given a position in the map, the sensed obstacles in time in this location could be different portions of the same tree over time so the shape is likely not to be constant.

ROW CROSSING PROBLEM

The vineyard environment also presented another criticality, due to the average distance between the vines belonging to the same row; indeed, because of the height of the LIDAR used for obstacle detection, the trunks of the plants are detected while the horizontal branches of the vines often are not. If this situation occurs, the space between the trees is sensed as free of obstacles, even if the horizontal branches of the vines are an actual obstacle to for the UGV. The first attempt to solve this problem was to simply provide an higher inflation radius (see Section 1.6) to the obstacles, in order to fill the gap between a plant and the adjacent ones. However, this naive solution couldn't deal with the following considerations:

- a too small inflation radius is not able to provide a solution to the problems *i.e.* fill the gap between adjacent trees
- an inflation radius large enough to fill the aforementioned gap, make the navigation almost impossible, because even a small obstacle among two rows completely occludes the path.

Insertion of prohibition layer

Given the impossibility to solve the problem of row crossing with the standard inflation radius of the obstacle layer, we found a very effective solution that relies on the very simple global structure of the vineyard. Indeed, looking at the map in Figure 4.3a, it's easy to notice that the obstacles that cause the row crossing problem are of course aligned, and parallel to each other, so the idea was to implement a series of "virtual walls" corresponding to the vine rows. The implementation was possible thanks to a feature of *costmap_2d*, the package of ROS responsible of the creation of the occupancy grid maps. Indeed, this package implements a **layered** costmap: this means that, for example, the global and local costmap (see Section 1.6) live in two different layers, that are computed independently:

- the global costmap is computed from the obstacles described in the occupancy grid map, possibly inflated of a chosen radius

- the local costmap is computed around the robot position, from the sensed laser scans. The obstacles possibly are inflated of a chosen radius

Since an arbitrary number of custom layers can be added by implementing C++ base class *costmap_2d::Layer* of Move Base package, two additional layers were introduced, in order to include the virtual obstacles of the rows both in global and local costmap:

- **Global prohibition layer:** this includes the virtual walls in the global costmap, and it's useful for the global planner to take them into account for the creation of the global navigation plan
- **Local prohibition layer:** this includes the virtual walls in the local costmap, and it's useful for the local planner to take them into account for the creation of the local navigation plan. This is required because otherwise the local planner might over-optimize the global plan and, if the walls were not included also in the local plan, the local planner could however take advantages of the gap between the plant. This situation is comparable to an indoor planning problem in which a local planner detect an open door that was shut at mapping time: even if the global path doesn't take advantage of the door gap, the local plan might choose to traverse it if it reduces the motion execution.

Switch to mapless navigation

After several effort in Move Base parameter tuning, the role of AMCL in the poor performance in navigation tasks was clear. Since the map in navigation task is used in terms of:

- **global planning**, because it provides a set of obstacles that helps the robot to optimize the global path in planning phase

Switch to teb

amcl in robot localization

inizialmente, map: molto ripetitiva, grossi problemi con amcl; inoltre problema perché sto scemo prova a passare in mezzo ai filari. Quindi virtual fences, inflated in global map, non inflated in local. Allora visti problemi di amcl e mappa, proviamo mapless: mappa bianca in cui mettere solo fences, e poi local costmap dice la sua. In questo modo le virtual fences sono usate per global planning. Dopo lo switch, provato a reintegrare la mappa usando amcl come input al nodo gnss di rob loc. Tuttavia, per motivi da capire non funzia molto bene. Problemi con il local planner dwa (tende a incastrarsi negli angoli), passaggio a elastic band.

Figure 4.5: *Belief about the robot estimate using AMCL, through a particles cloud where each particle represents a hypothesis about the real pose.*

Localization Chapter SKETCH:

- problemi avuti: mappa molto ripetitiva, il robot tenta di passare attraverso i filari, odometria delle ruote poco affidabile dato il terreno, problemi con amcl
- virtual obstacles: cenni (perché non l'ho fatto io) e immagini
- configurazione di robot localization, con i sensori usati (ruote+imu+gps)
- non sappiamo ancora quale sarà la soluzione utilizzata alla fine tra la versione mapless e amcl integrato con robot_localization (o eventuali altre soluzioni che salteranno fuori). Una volta che è stata presa una decisione, presentare quella come soluzione definitiva e spiegare perché l'altra è stata giudicata meno efficace

5 MANIPULATION

In this chapter, we'll explain our implementation of the action servers described in Chapter 3 related to the scan motion for point cloud registration, and to the pheromone dispenser application. But before it, we'll see a few deail about the Kinova API, and the different techniques we used for the motion planning and execution for the Jaco^2 arm.

KINOVA ROS API

ROS drivers of Kinova devices are very complex, and provide via ROS topics a large amount of API useful for both monitoring the arm state, and controlling it. Besides it, *jaco_arm_driver* node also publishes the *tf* subtree that describes the position of all the arm joints. In this section we do not want to provide a deep analysis of the drivers software architecture, but we limit ourselves to list the interfaces offered as ROS topics, since they are the ones that we used in GRAPE project. The published topics are, used to output information about the arm state, are:

- */j2n6s300_driver/out/cartesian_command*
- */j2n6s300_driver/out/finger_position*
- */j2n6s300_driver/out/joint_angles*
- */j2n6s300_driver/out/joint_command*
- */j2n6s300_driver/out/joint_state*
- */j2n6s300_driver/out/joint_torques*
- */j2n6s300_driver/out/tool_pose*
- */j2n6s300_driver/out/tool_wrench*

They allow to observe the position of the end effector both in Cartesian and *C-space*, together with position of the arm in *C-space*, the position of the fingers, the angular velocity of the joints, the torque and force applied to the joints.

The subscribed topics, used to control the arm movement, are:

- */j2n6s300_driver/in/cartesian_force*
- */j2n6s300_driver/in/cartesian_velocity*

- */j2n6s300_driver/in/joint_torque*
- */j2n6s300_driver/in/joint_velocity*

They allow to control the end effector both in force and velocity, and the joints in both torque and velocity.

MOTION EXECUTION METHODS FOR JACO^2 ARM

MoveIt! framework

The first, and easier, motion execution method analyzed is simply to use an *off-the-shelf* motion planning and execution framework, **MoveIt!**. However, in Section 3.1.2 we revealed in advance that the movement of the chosen arm, even after a bug fixing session in ROS drivers of Jaco^2 arm, presented a series of major weakness using the arm API through *MoveIt!*. In detail, the problems we identified were:

- non-repeatability of the movements; this was a problem because in situations with plenty of obstacles like the dispenser application, we could not rely on a few test movement to be sure that the movements did not interfere with obstacles
- lack of precision in the movements; if a target position of the arm was given, the actual position of the end effector at the end of the execution was likely to be several centimeters away from the expected one, about $0 \div 5$cm. This led to huge problems for precision tasks (dispenser grasping, dispenser deployment) where the maximum tolerable error is of the order of 1mm because of the dispenser narrowness
- inability in following a trajectory, maintaining the end effector orientation; this capability is mostly required during the scan motion, because it guarantees a uniform density of points collected on the plant, and this is useful with a view to the post-processing operations for the identification of the deployment point. If we impose such a director trajectory, only a small percentage of it is actually executed
- normalization of the joint angular positions in $[0; 2\pi]$ planning phase, with resulting loss of information about the absolute angular position of the joints of the arm. Even if in the vast majority of situations this is not a problem (since all joints of the Jaco^2 are capable of unlimited rotations), we needed to keep this factor into account because of the wiring for power supply and data interface of the sensors (LIDAR and RGB-D camera) mounted on top of the arm end effector. Indeed, even if the end effector joint is capable of infinite rotation, there is only a subset of this interval, that we approximate to $[0; 2\pi]$ (TODO verifica il

range reale nella configurazione finale), that is compatible with the wiring setup of the system, to avoid the wires being pulled. But since the planner has no access to the absolute angular position of the joints, it doesn't exists a smooth solution[1] to the problem of imposing one or more complete revolutions of the end effector joint.

Beyond all the flaws of our specific arm, an intrinsic weakness of this control type is that it can be very precise in reaching a specific target pose, but they cannot handle situations where there is a degree of uncertainty about the target pose, and we are in this situation in two different moments:

- when we need to grasp a dispenser, because the position of the dispenser in the dispenser feeder depends on how many dispenser have already been removed and however we do not want to have very strict constraints on the positioning of the dispensers in the feeder
- when we need to deploy the dispenser on the plant, because the deployment points output from the point cloud processing are likely not to be perfectly precise.

However, one big advantage of *MoveIt!* framework is that it supports, in planning phase, the specification in URDF[2] of user-specified static obstacles, to be introduced as constraints in *C-space*. For example, in Figure 5.1 you can see the volumes of sensors and other physical obstacles on top of the real Husky base, modeled as parallelepipeds; motion planning with *MoveIt!* will never produce a motion plan containing configurations that intersect these obstacles.

For all the aforementioned reasons, *MoveIt!* framework is suitable for the motions where:

- extreme precision is not a requirement
- constant orientation of the end effector is not a requirement
- no constraints abount the absolute angular position of the arm joints
- the goal position is well known

Direct publication of joint velocities

We introduced this method to bypass the problem of the normalization of the joints angular position. This method consists in a very

1 a non-smooth solution using *MoveIt!* is to consider the revolutions as concatenation of a sequence of rotations of an angle $\theta \in [0; 2\pi)$ of the end effector joint.

2 Unified Robot Description Format; it's an XML format used for representing physical components of a robot model

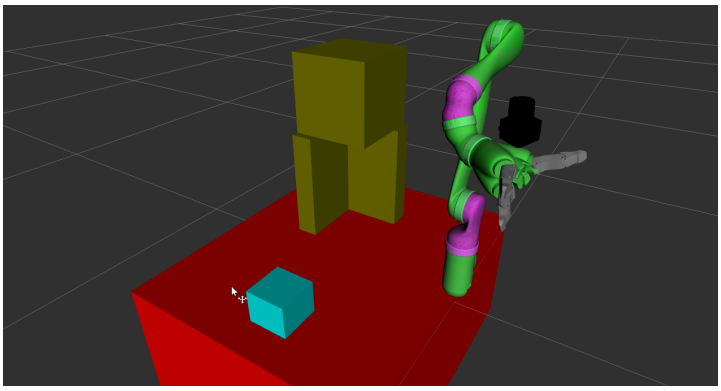


Figure 5.1: *Visual representation with RViZ of the obstacles on top of the Husky (red shape) used by MoveIt! planner: the Velodyne sensor (light blue shape), and the physical support for camera, GPS and IMU (green shapes).*

simple closed-loop control that compares the target position of a joint with the the actual one (obtained through Jaco^2 ROS API), and applies a velocity to the concerned joint if the error is still too high. This method is feasible because Jaco^2 ROS API provides (among other) a topic for controlling speeds in joint space (*i.e.* the configuration space, see Section 1.6) of the arm, thus we could control the velocity of each of the joints by publishing at a given frequency (100Hz) a message with the desired velocity for each joint. The stop condition in movements executed with this method is given by the distance of the joint position from the target angular position for the controlled joint being less than a given threshold. The actual joint position is monitored on the specific topic, published by Jaco^2 ROS API.

This method applies well for:

- simple movements, with no sophisticated control on velocity or position
- execution through *MoveIt!* is uncomfortable for one of the aforementioned reasons

In particular, we were interested in applying a constant velocity to the end effector for problem of the wires entanglement, to solve the problem of the normalization of the angular position of the joints in planning phase in *MoveIt!*.

Inversion of differential kinematics

We introduced the method of *inversion of differential kinematics* to bypass the problem of moving the arm maintaining the end effector pose constant. This method, like the previous one, relies on ROS API for the publication of velocities in joint speed, but is suitable for more complex movement than the previous one. Our goal is to express the

velocity of the end effector frame in terms of joint velocities *i.e.* given the desired velocity of the end effector, calculate the 6 joints velocities to be published on the suitable topic in order to achieve the desired end effector velocity. This is a *inversion of differential kinematics* problem, formally: being q a coordinate vector that represent the position of the robot in joint space, v the vector of linear and angular velocity of the end effector frame with respect to the fixed base frame, and $J(q)$ the geometrical Jacobian matrix of the manipulator, composed by the stacking of $J_P(q)$, related to the end effector position, and $J_o(q)$, related to the end effector orientation:

$$v = \begin{bmatrix} \dot{p} \\ \omega \end{bmatrix} \qquad q = \begin{bmatrix} q_1 \\ \vdots \\ q_n \end{bmatrix} \qquad J(q) = \begin{bmatrix} J_P(q) \\ J_o(q) \end{bmatrix}$$

The following relationships hold:

$$\dot{p} = J_P(q) \cdot \dot{q}$$
$$\omega = J_o(q) \cdot \dot{q}$$

In compact form:

$$v = \begin{bmatrix} \dot{p} \\ \omega \end{bmatrix} = J(q) \cdot \dot{q}$$

Thus, we can solve the initial problem with:

$$\dot{q} = J^{-1}(q) \cdot v \tag{5.1}$$

Systematic algorithm exists to compute the geometric Jacobian matrix (**jacobian**); in our case, it's simply provided by *MoveIt!* ROS API. In our implementation, the computation (with expression 5.1) and publication of the joint speed is a periodic task executed with frequency 100Hz. Of course, this is required because the Jacobian varies in time, according to the positions of the joints. Since this motion technique allows for very smooth and uniform motion of the end effector, maintaining its orientation, we used it to execute the arm scan motion that requires exactly the aforementioned characteristics.

Note that, even if this method is very effective, it has some limitations for what concerns the *exceptional behavior*, since it does not embed dangerous situations detection (that is instead provided by more complex frameworks like *MoveIt!*). After a few tests, it was clear that we had to introduce some checks during movements executed with this method, since the poses required to perform the scan motion were dangerously close to arm singularities. The check we introduced acts like a stop condition for the movement execution, and it's based on the assumption that, in a non-pathological context, the Cartesian

distance between the end effector pose and the target pose is monotonically decreasing in time. The value $d(t)$ computed as:

$$
\begin{aligned}
&p_{endEffector}(t) = (x_c(t), y_c(t), z_c(t)) \\
&p_{target} = (x_t, y_t, z_t) \\
&d(t) = \sqrt{(x_c(t) - x_t)^2 + (y_c(t) - y_t)^2 + (z_c(t) - z_t)^2}
\end{aligned}
$$

is monitored in time using the values of $p_{endEffector}$ provided by Jaco^2 drivers, and the arm is stopped with consequent abortion of the corresponding action if $d(t)$ is observed to have grown in time instead of shrinking. In order to avoid false positives due to sensor noise when the arm still has to actually start to move in the very first moments of the action execution (the arm is not moving, but of course the sensed end effector pose is not 100% stable over time), we apply this error detection measure only after the arm has moved a few centimeters (*e.g.* 2-3 cm) from the initial pose. Of course, the other stop condition for the motion execution is that the value of $d(t)$ shrinks under a certain threshold (*e.g. 2-3cm*). Keep in mind that the focus in this motion technique is on smoothness of the movement and constancy in the end effector orientation, rather than the precision in reaching the target pose.

Image Based Visual Servoing

This technique allows for much more precise positioning of all the other described in previous sections; it is a closed-loop control and makes use of the RGB-D camera mounted on top of the end effector. Now, we'll give a formal description of image-based visual servoing algorithm (**imageBasedVisualServo**).

Define a feature point as the vector:

$$
f = \begin{bmatrix} u \\ v \end{bmatrix}
$$

where (u, v) are the coordinates of the point on the image plane representing the image of the considered point in 3D space. In case more than one 3D point, and thus more than one feature point, a feature vector can be defined as:

$$
\mathbf{f} = \begin{bmatrix} f_1 \\ \vdots \\ f_n \end{bmatrix}
$$

In IBVS the goal configuration is defined by a desired configuration $\mathbf{f_d}$ of the image features. Therefore, the image error function is given by

$$\mathbf{f_d} = \begin{bmatrix} f_u \\ f_v \end{bmatrix}$$
$$\boldsymbol{e}(t) = \mathbf{f}(t) - \mathbf{f_d}$$

The image-based control problem aims at finding a mapping from this error function to a commanded camera motion, under the following assumptions:

- the manipulator is a kinematic positioning device, *i.e.*, manipulator dynamics are neglected
- the trajectories generated by the IBVS controller can be tracked by a lower level manipulator controller
- the target is fixed and the desired configuration $\mathbf{f_d}$ is constant.

The most common approach to solve the aforementioned problem is to compute a desired camera velocity and use this as the control input, assuming that the robot can be commanded by a set of linear and rotational Cartesian velocities. The camera velocity can be computed using the relation, represented by the *interaction matrix*, between the velocity of the camera frame and the velocity of the features on the image plane:

$$\dot{\mathbf{f}}(t) = L(\mathbf{f}, q)v_{cam}$$

Where the interaction matrix for a feature point is given by:

$$L(f) = \begin{bmatrix} -\frac{f_u}{Z} & 0 & \frac{u}{Z} & \frac{uv}{f_u} & -\frac{f_u^2+u^2}{f_u} & v \\ 0 & -\frac{f_v}{Z} & \frac{v}{Z} & \frac{f_v^2+v^2}{f_v} & -\frac{uv}{f_v} & -u \end{bmatrix}$$

and v_{cam} is the vector of linear and rotational Cartesian velocities of the camera. In the case of a vector of feature points (*i.e.* $n > 1$), the interaction matrix can be obtained stacking the interaction matrices associated to each point. Assuming that the number of selected features is such that the whole interaction matrix as more rows than columns ($n \geqslant 4$), the previous relation can be inverted using a least squares solution as follows:

$$v_{cam} = (L^T L)^{-1} L^T \dot{\mathbf{f}}(t) \tag{5.2}$$

Since the derivative of the error function is given by

$$\dot{e}(t) = \frac{d}{dt}(f(t) - f_d) = \dot{f}(t) = L(f)v_{cam}$$

the following control rule is obtained:

$$v_{cam} = (L^T L)^{-1} L^T \dot{e}(t)$$

Since the Jaco^2 arm drivers provide a ROS topic to control the end effector in Cartesian velocities, the assumption holds. In this motion technique, the stop condition is given by the error between the target features position and the actual feature position $e(t)$, computed starting from the desired and actual position for each feature f:

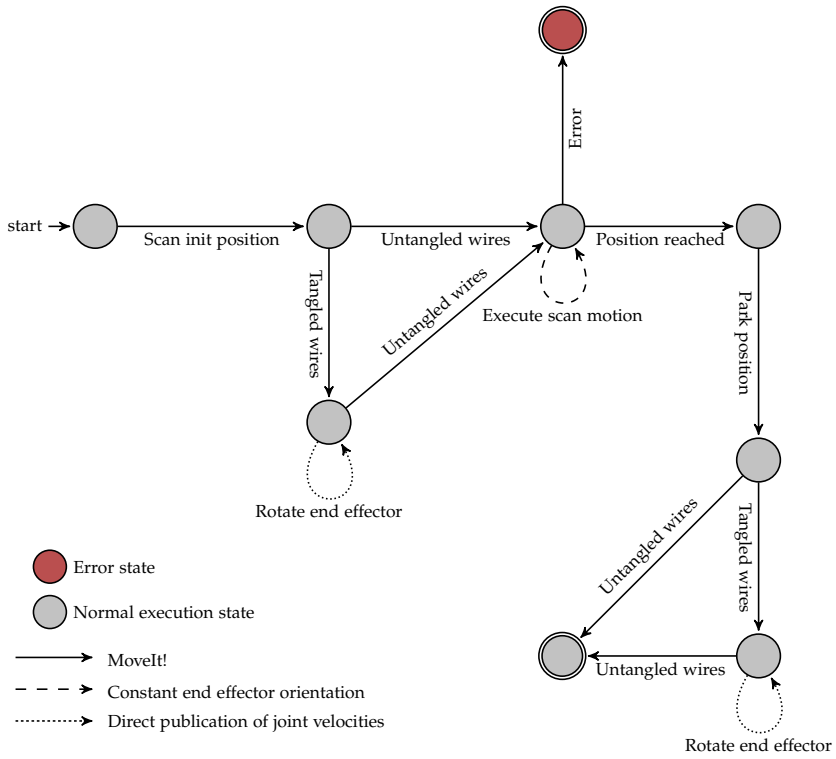
$$e_{fx}(t) = (x_{actual} - x_{target})$$
$$e_{fy}(t) = (y_{actual} - y_{target})$$
$$e(t) = \sqrt{\sum_f e_{fx}^2(t) + \sum_f e_{fy}^2(t)}$$

being below a predetermined threshold. The usage of visual servoing techniques in our implementation will be clearer in Section 5.4.

SCAN MOTION ACTION SERVER

The scan motion is a quite simple procedure; in Figure 5.3 you can see the FSA triggered by the action call, with highlight of the different motion methods used for each of the motions. As you can see, most of the movements are implemented through *MoveIt!* framework, except for:

- the disentanglement of the wires after critical movements. The wires problem was mostly accentuated in the scan movement because some of the joints were more stretched, so a check is made before and after the scan motion. The motion is implemented with direct publication of end effector joint velocity
- the scan motion itself, that requires constant orientation of the LIDAR and so of the end effector, too. The motion is implemented through inversion of differential kinematic to maintain the end effector orientation.



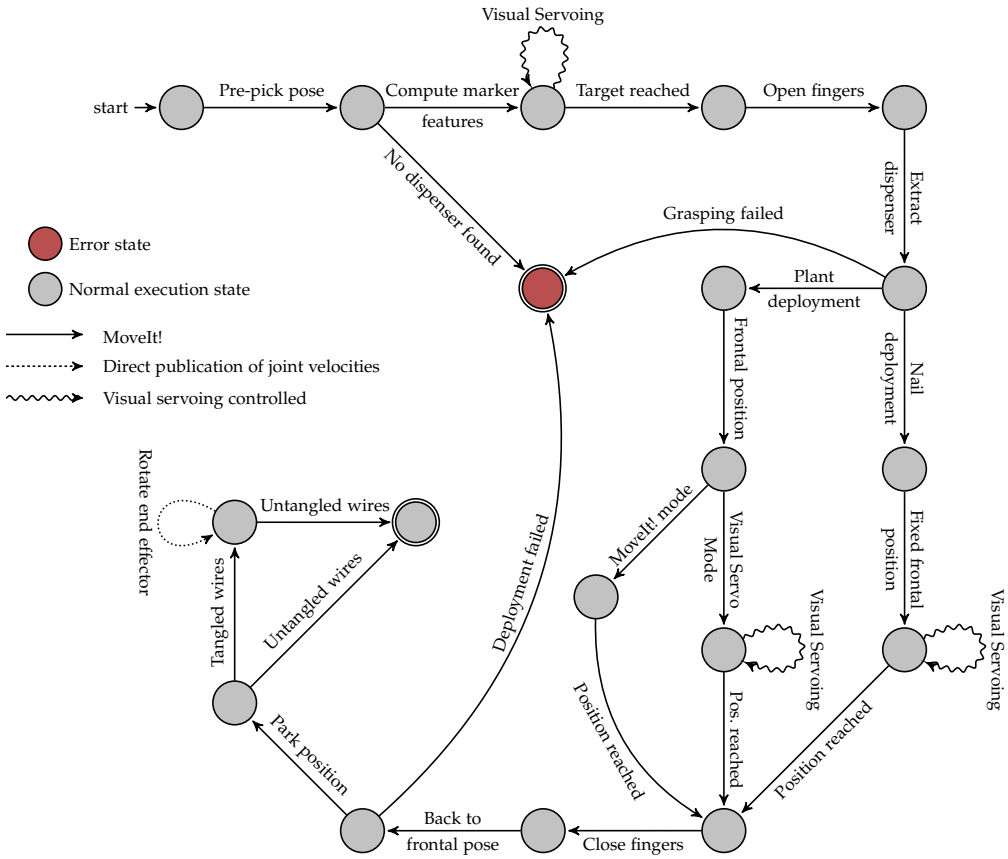
DISPENSER APPLICATION ACTION SERVER

In Figure 5.4 you can see the FSA triggered by the action call, with highlight of the different motion methods used for each of the motions. As in the previous action, most of the movements are executed with *MoveIt!*, except for:

- the disentanglement of the wires, as in scan motion action server
- the grasping of the dispenser from the feeder, executed with visual servoing control
- the deployment of the dispenser in nail mode, executed with visual servoing control
- one of the dispenser deployment mode on the plant, executed with visual servoing

This action is composed of two main parts:

1. Grasping of the dispenser, and image-based grasping validation
2. Deployment of the dispenser, and image-based deployment validation



Our image processing tasks are based on C++ API of **OpenCV library**[3]. Implementation detail are provided in Sections 5.4.1 and 5.4.2.

Image based validation

In order to check for the presence of dispensers in the feeder, and validate the grasp and deploy operations, we make use of the Intel RealSense RGB-D camera mounted on top of the arm end effector. The thinness of dispensers with respect to the camera resolution make them hard to detect, especially if, after a deployment operation, a rotation of the dispenser occurs and it's not possible framing it from a frontal position. For this reason, all our dispensers are equipped with two small red wings built out of tape, one on each side, as showed in Figure 3.10. Note that the red color of the wings was chosen because it stands out with respect to the typical colors of the vineyard during winter or spring months, and it's an important factor in the image processing algorithms that follow.

CHECK DISPENSER AVAILABILITY In this phase, the camera is placed on the vertical of the feeder and, to check for dispenser availability, it looks for two red wings in a central rectangular area (*valid area*) that should contain the feeder. In Figure 5.5, you can see some intermediate results of the algorithm execution:

3 https://opencv.org/

1. Color segmentation of the image, to find pixel areas with color similar to the wings (see 5.5a)
2. Morphological operations (*dilatation* and *erosion*) on the selected pixel regions, for noise reduction and discard of too small regions (see Figure 5.5b)
3. Discard of the regions outside a predefined rectangular area in the center of the image (see Figure 5.5c)

In Image 5.5d you can see the outcome of the aforementioned analysis; the wings identification is successful and their detected contour highlighted. The whole procedure is declared *failed* if no marker is detected in the image plane for a given amount of time (*e.g.* 5 seconds).

GRASPING VALIDATION As you can see in Figure 5.4, this check is performed after the fingers has opened and the end effector has already moved from the grasping pose but it still is on the vertical of the feeder, because we want to detect a possible failure as soon as possible. In case of successful grasping, we expect to see the red wing of the dispenser in the bottom right corner of the image (*valid area*). The copious amount of tests showed that that the grasping procedure couldn't lead to a deterministic position of the wing in the image plane, thus we had to take additional measure for the validation procedure to be effective. In particular, in the laboratory environment where our procedure were developed, problems came from:

- the color of the test bench similar shade of the dispenser wings color, especially under direct enlightenment
- the *specular component* (see Figure 5.2) of the reflection from the desk surface causes noise in the depth sensing when using stereoscopic infrared sensors (**reflectionDepthSensing**), in particular the depth of pixels belonging to highly reflective surfaces is always very low. For this reason, discarding pixel regions by thresholding over the distance of the regions from the camera is useful, but not sufficient to discard all the noise regions.

Because of the non-deterministic position of the dispenser in the image, we could not simply discard these noise by cropping a narrower valid area or setting a stricter threshold value for the region area. We noted that noise regions that survive the first steps of the filtering procedure often have a jagged contour, while we know that the dispenser wings have very sharp edges in the image. Basing on this observation, we introduced an additional filtering step, where for each of the pixel regions survived to previous filtering steps:

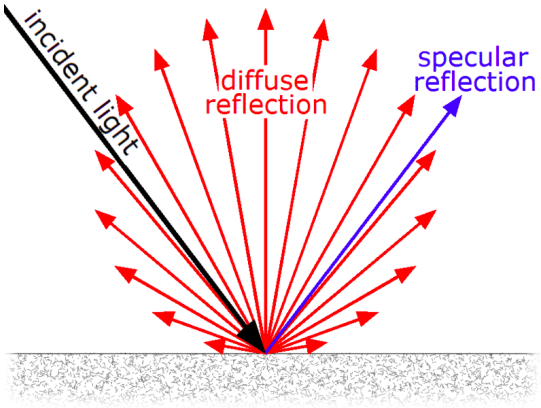


Figure 5.2: *Reprentation of the reflection of light on a glossy surface: a ray incident on the surface is both scattered at many angles (**diffuse reflection** component), and reflected at one single angle (**specular reflection** component).*

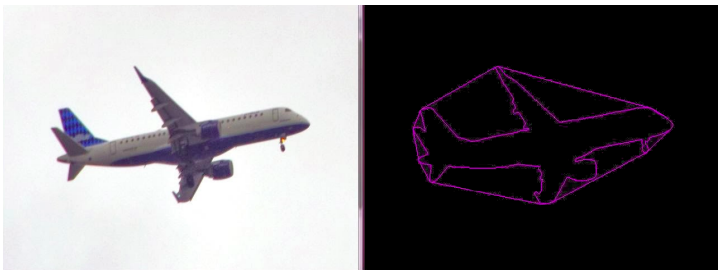


Figure 5.3: *Convex hull computation of the edges of a sample image.*

- compute the convex hull of that region *i.e.* the smallest convex set of pixels that contains it (see Figure 5.3)
- compute the matching between the region itself, and its convex hull; the OpenCV method for calculation of the match between different images are based on the Hu invariants (**huInvariants**)
- discard of the regions with matching value below a fixed threshold

The idea behind this filtering procedure is that the trapezoid region of the dispenser is very regular and convex so it will be very similar to its convex hull, while the jagged contour of a noise region will be quite different from its convex hull.

Even if these problems were due to elements in the test environment and not in the actual vineyard, we decided to introduce the additional measure also in the final version of the procedure to make it more robust to possible noise we were not aware of in development phase.

The algorithm steps for the validation of the dispenser grasping are:

1. Color segmentation of the image, to find pixel areas with color similar to the wings (see Figure 5.6a)
2. Morphological operations (*dilatation* and *erosion*) on the selected pixel regions, for noise reduction and discard of too small regions (see Figure 5.6b)
3. Discard regions with too high depth value *e.g.* wings of dispensers still in the feeder, that are not took into account (see Figure 5.8b)
4. Discard regions outside the valid area (bottom right corner)
5. Discard region with a too low degree of similarity with their convex hull

If at least one region survives the whole filtering procedure, this region is likely to be the one identified by the dispenser and the grasping is validated. **NOTE:** we check for *at least* one region instead of *exactly* one, because the procedure is much more likely to select noise regions as correct than discard the dispenser region, so false negatives are very unlikely and if we checked for exactly one region we would be more likely to produce false positives. At each grasp validation request, the whole filtering procedure is executed on a fixed number of consecutive frames (*e.g. 50* frames), and the final output is given by the majority vote of the cumulated executions. This procedure is made to reduce the effect of random noise in the measurement.

DEPLOYMENT VALIDATION This operation is similar to the previous one, because we assume that, if the deployment task has been successful, by moving the camera back and a few centimeters downward maintaining the deployment orientation, we expect to detect the dispenser in the captured image. In our implementation we do not deal with cases where the dispenser rotates and, from the side, the wings are not visible. There is a slightly difference between the case of deployment on the nail and on the plant, because:

- in nail mode, the camera position from which search for the dispenser wings is fixed
- in plant mode, the camera position from which search for the dispensers wings is computed as fixed offset from the deployment point took as an input by the action server, while in this mode it's a fixed position.

Apart from that, the validation procedure is the same in both plant and nail mode, and it's composed by the following steps:

1. Color segmentation of the image, to find pixel areas with color similar to the wings

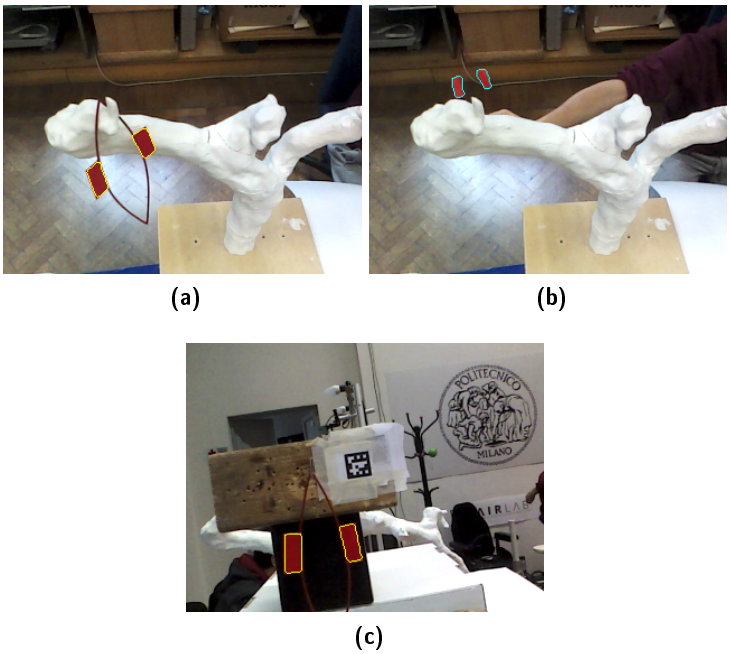


Figure 5.4: *Different outcomes of the deployment validation algorithm: success in plant mode (Figure 5.4a), failure in plant mode (FIgure 5.4b; the dispenser in the background does not induce a false positive because it does not satisfy depth requirements), success in nail mode (Figure 5.4c).*

2. Morphological operations (*dilatation* and *erosion*) on the selected pixel regions, for noise reduction and discard of too small regions
3. Discard regions with too high depth value *e.g.* wings of dispenser deployed on other plants, in adjacent rows
4. Discard region with a too low degree of similarity with their convex hull

Like grasping validation, also the deployment validation procedure is executed on a fixed number of consecutive frames, that produce a cumulated result given by their majority vote. In figure 5.4 you can see possible outcomes of the deployment validation.

Visual servoing tasks

In this section we'll describe with more detail each of the operations executed with visual servoing control. However, this is done mostly for the sake of completeness, because the design and implementation of the automatic control was not directly part of this thesis work, so technical details will be omitted.

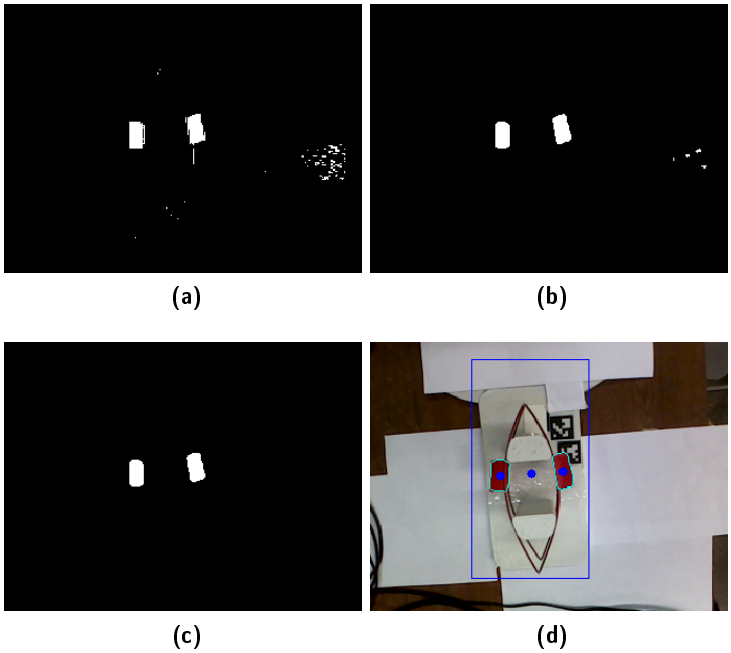


Figure 5.5: *Visual representation of the dispenser availability check algorithm.*

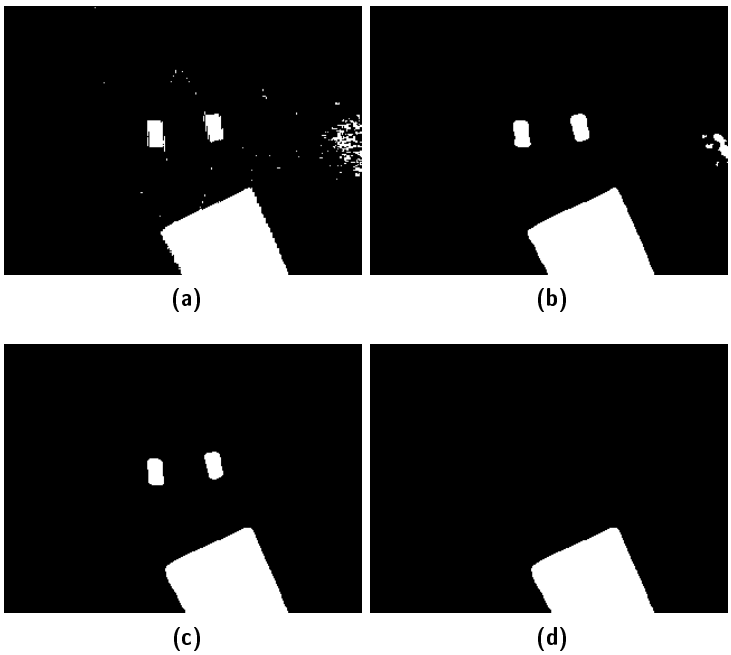


Figure 5.6: *Visual representation of the dispenser availability check algorithm.*

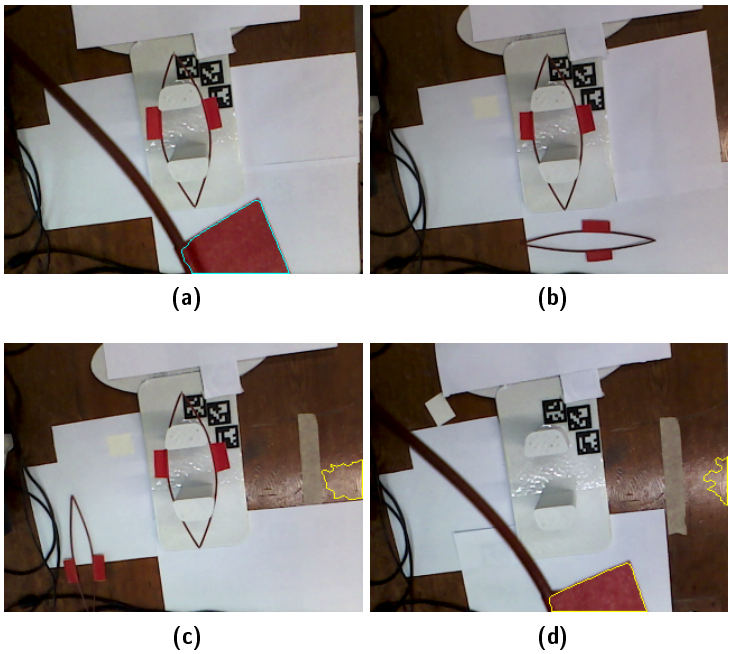


Figure 5.7: *Different outcomes of the grasp validation algorithm: true positive (5.8a), true negative (5.8b), false positive (5.7c), error situation that however brings to a true positive (5.7d)*

As mentioned in Section 5.2.4, for executing image based visual servoing control, some features must be tracked in order to calculate the Cartesian velocities for the end effector. For this reason, all the arm motions controlled with visual servoing are preceded by a motion, executed through *MoveIt!*, that brings the end effector in a position that is frontal to the features that are tracked for the visual servoing control, such that they are present in image plane. For each case, we'll see which features are used and how. We'll see that some of the operations rely on ArUco markers (**aruco**); they are a type of binary square fiducial markers (see Figure 3.8), with some characteristics designed to make them particularly suitable for obtainment of the camera pose for computer vision applications:

- a black wide black border that facilitates its fast detection in the image
- an inner binary matrix which determines its identifier and allows for the application of error detection and correction techniques
- the marker size determines the size of the internal matrix *e.g.* a marker size of 4x4 is composed by 16 bits.

Make reference to the FSA of figure 5.4 for better understanding of the operations in the whole context.

DISPENSER GRASPING For this task, the tracked features are the 4 corners of an ArUco marker placed on the side of the dispenser feeder, in a fixed position with respect to the feeder center. For this task, the desired target positions of the marker corners (the tracked features) are computed starting from the detected position (in image frame) of the midpoint of the two dispenser wings, and the sensed depth of the wings pixel regions. This last information is required because the desired target of the end effector varies according to the position of the highest dispenser in the feeder.

In the first implementation of this operation problems occurred when, during the descent of the end effector toward the feeder, the *specular component* of the reflection from the marker surface pointed directly the camera eye. In that case the noise in the image was too high to detect the marker features. For this reason, a second marker with different ID (see Image 5.8) was placed next to the previous one, on the same side of the feeder, in order to be always visible from the camera. Before visual servoing control kicks in, the target positions of the features of both markers are computed independently; the procedure has a "main" reference marker that, if detected, provides the features for the visual control; if the primary marker is not visible (see Figure 5.8a), the control procedure smoothly switches to use the features of the second marker for the visual control.

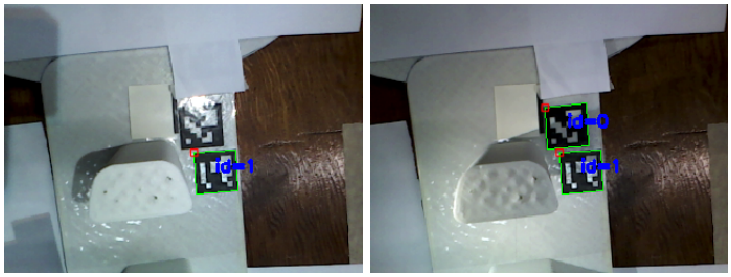
As explained in Section 5.2.4, if the number of features is enough, we can compute a full set of linear and angular Cartesian velocities

$$< \dot{x}, \dot{y}, \dot{z}, \dot{\psi}, \dot{\theta}, \dot{\phi} >$$

for the end effector. Since we have at disposal 4 features in this task, the end effector speed can be fully determined by visual servoing control.

DEPLOYMENT IN NAIL MODE This task is a simpler version of the previous one, because:

- also in this case, the features are given by the corners of an ArUco marker, placed this time at a small fixed offset from the target nail (see Figure 5.4c)
- a single marker is used, because the specular reflection effect is much less relevant on a vertical surface
- the target positions of the features are a fixed value, because there is no such a thing like different dispenser height in grasping phase.
- the initial frontal position is not function of the deployment point passed in the action server goal, but is a fixed value.



(a) (b)

Figure 5.8: *Two markers used for visual servo task in grasping phase. In Figure 5.8b both the markers are detected, while in Figure 5.8a the dispenser with ID= 0 is not visible due to specular reflection on marker surface, but the other marker is still detected so the visual servo task can proceed.*

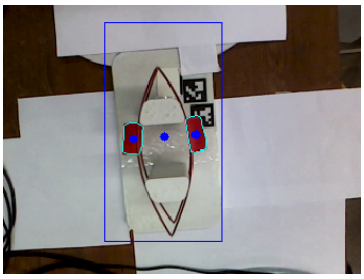


Figure 5.9: *Example of the computed wings midpoint, used to compute the target position of the features (marker corners)*

As highlighted in the last point, this deployment procedure is 100% independent from the input deployment point. Thus, if this deployment mode is selected, scan phase is not required. Note that, also in this phase, we can rely on 4 different features (the marker corners) to be tracked in the control phase, so all 6 linear and angular velocities can be computed.

DEPLOYMENT IN PLANT MODE This task is quite different from the previous ones, because of course we cannot take advantage of markers since our deployment point has been selected on the vine plant through the scan procedure described in Section 5.3, so we must use a different type of features. The visual feature used for this control task is a single point in the image, corresponding to the center of a circle, which contains points close to the deployment point in image plane, and with similar color. The idea behind it is to select a internal point of the same small branch that the input deployment point belongs to. In figure 5.10 you can see some of the image processing steps that leads to the identification of the feature point. The target position of the feature in image frame, instead, is hardcoded to a specific pixel, corresponding to the position of the dispenser center. Note that, since we only have one feature at disposal, matrix $L(f)$ in this case is not large enough to provide a solution for all linear and angular velocities (see Equation 5.2). Indeed, in this deployment mode:

- $\dot{\psi} = \dot{\theta} = \dot{\phi} = 0$, *i.e.* the end effector orientation is constant
- $\dot{z}$ is proportional to the distance between the end effector pose and the deployment point
- $\dot{x}$ and $\dot{y}$ are controlled by the visual servoing controller.

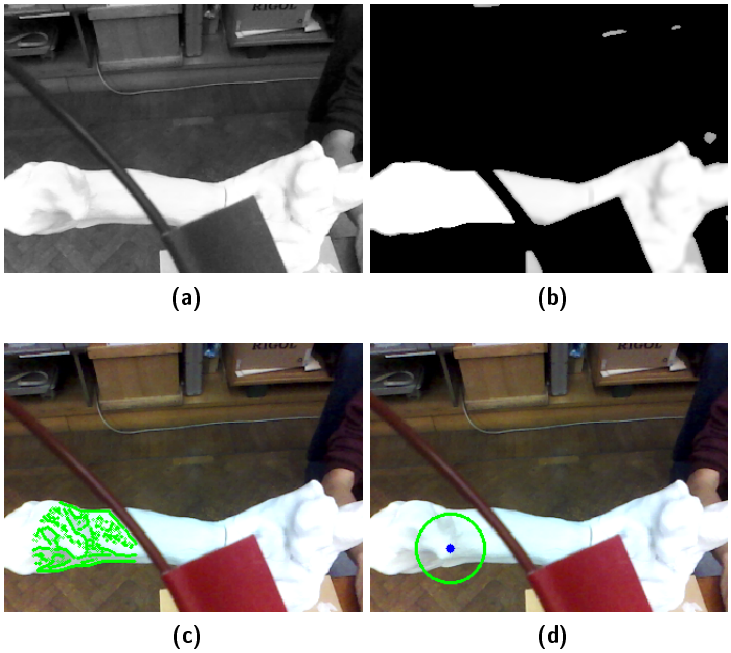


Figure 5.10: *Some of the steps computed to identify the feature point tracked for dispenser application on the vine tree: switch from RGB to grayscale (Figure 5.10a), selection of pixels similar to the pixel corresponding to deployment point (Figure 5.10b), identification of branch area close to the deployment point (Figure 5.10c), identification of the circle including the selected points (Figure 5.10d).*

BIBLIOGRAPHY

Calabrese, Luca (2014). "Robust odometry, localization and autonomous navigation on a robotic wheelchair." In: (cit. on p. 15).

Carde Ring T Minks, Albert K (1995). "Control of moth pests by mating disruption: successes and constraints." In: *Annual review of entomology* 40.1, pp. 559–585 (cit. on p. 25).

Colledanchise, Michele and Petter Ögren (2017). "Behavior Trees in Robotics and AI, an Introduction." In: *arXiv preprint arXiv:1709.00084* (cit. on p. 44).

Cucci, Davide A and Matteo Matteucci (2014). "Position tracking and sensors self-calibration in autonomous mobile robots by gauss-newton optimization." In: *Robotics and Automation (ICRA), 2014 IEEE International Conference on*. IEEE, pp. 1269–1275 (cit. on p. 15).

Cucci, Davide Antonio and Matteo Matteucci (2013). "A Flexible Framework for Mobile Robot Pose Estimation and Multi-Sensor Self-Calibration." In: *ICINCO (2)*, pp. 361–368 (cit. on p. 15).

Diago, Maria P, Javier Tardaguila, et al. (2015). "A new robot for vineyard monitoring." In: *Wine & Viticulture Journal* 30.3, p. 38 (cit. on p. 28).

Dudek, Wojciech, Wojciech Szynkiewicz, and Tomasz Winiarski (2016). "Nao robot navigation system structure development in an agent-based architecture of the RAPP platform." In: *Challenges in Automation, Robotics and Measurement Techniques*. Springer, pp. 623–633 (cit. on p. 15).

Elmenreich, Wilfried (2002). "Sensor fusion in time-triggered systems." In: (cit. on p. 13).

Foote, Tully (2013). "tf: The transform library." In: *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on*. Open-Source Software workshop, pp. 1–6 (cit. on p. 5).

Fox, Dieter, Wolfram Burgard, Frank Dellaert, and Sebastian Thrun (1999). "Monte carlo localization: Efficient position estimation for mobile robots." In: *AAAI/IAAI* 1999.343-349, pp. 2–2 (cit. on p. 17).

Fraser, Alex S (1957). "Simulation of genetic systems by automatic digital computers I. Introduction." In: *Australian Journal of Biological Sciences* 10.4, pp. 484–491.

Gage, Douglas W (1995). *UGV history 101: A brief history of Unmanned Ground Vehicle (UGV) development efforts*. Tech. rep. NAVAL COMMAND CONTROL, OCEAN SURVEILLANCE CENTER RDT, and E DIV SAN DIEGO CA (cit. on p. 38).

Hart, Peter E, Nils J Nilsson, and Bertram Raphael (1968). "A formal basis for the heuristic determination of minimum cost paths." In:

IEEE transactions on Systems Science and Cybernetics 4.2, pp. 100–107 (cit. on p. 20).

Herring, David (2001). "Precision farming: Feature articles." In: (cit. on p. 24).

Hinkel, Georg, Henning Groenda, Lorenzo Vannucci, Oliver Denninger, Nino Cauli, and Stefan Ulbrich (2015). "A domain-specific language (DSL) for integrating neuronal networks in robot control." In: *Proceedings of the 2015 Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering*. ACM, pp. 9–15 (cit. on p. 31).

Hutchinson, Seth, Gregory D Hager, and Peter I Corke (1996). "A tutorial on visual servo control." In: *IEEE transactions on robotics and automation* 12.5, pp. 651–670 (cit. on p. 35).

Latombe, Jean-Claude (2012). *Robot motion planning*. Vol. 124. Springer Science & Business Media (cit. on p. 18).

LaValle, Steven M (1998). "Rapidly-exploring random trees: A new tool for path planning." In: (cit. on p. 21).

Lopes, Carlos M, Albert Torres, Robert Guzman, João Graça, Miguel Reyes, Gonçalo Vitorino, Ricardo Braga, Ana Monteiro, and André Barriguinha (2017). "Using an unmanned ground vehicle to scout vineyards for non-intrusive estimation of canopy features and grape yield." In: *GiESCO International Meeting, 20th, Sustainable viticulture and wine making in climate chenge scenarios, 5-10 November 2017*. GiESCO (cit. on p. 28).

Lozano-Perez, Tomas (1983). "Spatial planning: A configuration space approach." In: *IEEE transactions on computers* 2, pp. 108–120 (cit. on p. 19).

Madhavan, Raj, Lino Marques, Edson Prestes, Renan Maffei, Vitor Jorge, Baptiste Gil, Sedat Dogru, Gonçalo Cabrita, Renata Neuland, and Prithviraj Dasgupta (2015). *2015 humanitarian robotics and automation technology challenge* (cit. on p. 31).

Marder-Eppstein, Eitan, Eric Berger, Tully Foote, Brian Gerkey, and Kurt Konolige (2010). "The Office Marathon: Robust Navigation in an Indoor Office Environment." In: *International Conference on Robotics and Automation* (cit. on p. 16).

Mohanty, Vikram, Shubh Agrawal, Shaswat Datta, Arna Ghosh, Vishnu Dutt Sharma, and Debashish Chakravarty (2016). "DeepVO: a deep learning approach for monocular visual odometry." In: *arXiv preprint arXiv:1611.06069* (cit. on p. 15).

Moore, T. and D. Stouch (2014). "A Generalized Extended Kalman Filter Implementation for the Robot Operating System." In: *Proceedings of the 13th International Conference on Intelligent Autonomous Systems (IAS-13)*. Springer (cit. on p. 15).

Oberti, Roberto, Massimo Marchi, Paolo Tirelli, Aldo Calcante, Marcello Iriti, M Hocevar, and H Ulbrich (2014). "Crops agricultural robot: application to selective spraying of grapevine's diseases."

In: *Proceedings of the Second RHEA International Conference on Robotics and associated High-technologies and Equipment for Agriculture* (cit. on p. 29).

Ostafew, Chris J, Angela P Schoellig, and Timothy D Barfoot (2013). "Visual teach and repeat, repeat, repeat: Iterative learning control to improve mobile robot path tracking in challenging outdoor environments." In: *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*. IEEE, pp. 176–181 (cit. on p. 31).

Remagnino, Paolo, Dorothy N Monekosso, and Lakhmi C Jain (2011). *Innovations in Defence Support Systems-3: Intelligent Paradigms in Security*. Vol. 336. Springer Science & Business Media (cit. on p. 13).

Ringdahl, Ola, Polina Kurtser, Ruud Barth, and Yael Edan (2016). "Operational flow of an autonomous sweetpepper harvesting robot." In: *The 5th Israeli Conference on Robotics 2016, 13-14 April 2016. Air Force Conference Center Hertzilya, Israel* (cit. on p. 29).

Santos, Pablo Gonzalez-de, Angela Ribeiro, and C Fernandez-Quitanilla (2012). "The RHEA Project: using a robot fleet for a highly effective crop protection." In: *Proceedings of the International Conference of Agricultural Engineering (CIGR-Ageng'12), Valencia, Spain* (cit. on p. 29).

Serrano, Daniel, Pietro Astolfi, Gianluca Bardaro, Alessandro Gabrielli, Luca Bascetta, and Matteo Matteucci (2017). "GRAPE: Ground Robot for vineyArd Monitoring and ProtEction." In: *ROBOT 2017: Third Iberian Robotics Conference*. Vol. 1. Springer, p. 249 (cit. on pp. 26, 51).

Shamah, Benjamin (1999). "Experimental Comparison of Skid Steering vs. Explicit Steering for Wheeled Mobile Robot," M. Sc." In: (cit. on p. 12).

Singhal, Amit (1997). "Issues in autonomous mobile robot navigation." In: *Computer Science Dept, U. of Rochester*, p. 74 (cit. on p. 47).

Thrun, Sebastian (2002). "Probabilistic robotics." In: *Communications of the ACM* 45.3, pp. 52–57 (cit. on p. 14).

Thrun, Sebastian, Dieter Fox, Wolfram Burgard, and Frank Dellaert (2001). "Robust Monte Carlo localization for mobile robots." In: *Artificial intelligence* 128.1-2, pp. 99–141 (cit. on p. 51).

Wang, Tianmiao, Yao Wu, Jianhong Liang, Chenhao Han, Jiao Chen, and Qiteng Zhao (2015). "Analysis and experimental kinematics of a skid-steering wheeled robot based on a laser scanner sensor." In: *Sensors* 15.5, pp. 9681–9702 (cit. on pp. 12, 13).

Yi, Jingang, Junjie Zhang, Dezhen Song, and Suhada Jayasuriya (2007). "IMU-based localization and slip estimation for skid-steered mobile robots." In: *Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on*. IEEE, pp. 2845–2850 (cit. on p. 12).