TRABAJO FIN DE GRADO

Grado en Ingeniería Electrónica Industrial

UNIVERSIDAD DE ALMERIA

ESCUELA SUPERIOR DE INGENIERÍA

Design of a control strategy for guiding collaborative robots using wearable EMG and IMU sensors

Curso: 2020/2021

Modalidad TFG: Trabajo Técnico

Alumno/a:

Mariachiara Mariotti

Director/es:
Dr. D. Antonio Visioli
Dr.D. José Carlos Moreno Úbeda

UNIVERSITY
OF BRESCIA

UNIVERSIDAD DE ALMERÍA
Escuela Superior de Ingeniería

Trabajo Fin de Grado
Ingeniería Electrónica Industrial

UNIVERSITÀ DEGLI STUDI DI BRESCIA
Dipartimento di Ingegneria Meccanica Industriale
Corso di Laurea Magistrale in Ingegneria dell'Automazione Industriale

DESIGN OF A CONTROL STRATEGY FOR GUIDING COLLABORATIVE ROBOTS USING WEARABLE EMG AND IMU SENSORS

Diseño de una estrategia de control para guidar robots colaborativos usando sensores EMG y IMU portátiles

Doble Título UNIBS-UAL
Mechatronics for Industrial Automation

Autor: Mariachiara Mariotti

Director: Dr. D. Antonio Visioli
Codirector: Dr. D. José Carlos Moreno Úbeda

Almería (España), Junio 2021
Curso 2020-2021

CONTENTS

LIST OF FIGURES

ABSTRACT

The use of robots is widespread in many industrial applications, but they can be considered as a risk if collaboration with a worker is required. In the last years, researches and studies over human-robot interaction (HRI) have increased and a new generation of robots has been developed: collaborative robots.

Collaborative robots are born to work safely with humans in the same working space, maintaining the efficiency and productivity of an industrial robot. That is possible thanks to a synergy of control strategies and specific mechanical and electronic solutions, as SEA.

The human-robot interaction causes a new problem: the interaction between workers and the collaborative robot.

The present thesis deals with robot control and human-robot interaction: a control strategy for guiding collaborative robots using wearable EMG and IMU sensors is presented in this work. The ultimate goal is to make possible to the human operator the control of robot's behavior by the use of sensors.

After a first descriptive section over the instrumentation, the design and implementation stages of the control system are presented. The control program is implemented inside the ROS environment. The Myo armband is the sensing device. Two collaborative robots were used to test the system, they have six-degree of freedom and they are called FourByThree and UR10e.

The resulted control system can be divided into three different control methods:

- *Cartesian velocity control: the linear velocity of the arm is used to control the cartesian velocity of the robot.*
- *Teaching by manual guidance and path repetition: the robot is manually moved and a gesture command of the worker is used to save the robot configuration. In that way, the robot learns a path executable by itself. Another gesture command allows the repetition of the last memorized path.*
- *Direction control: the robot is moved along the same direction as the arm movement.*

The final section of the present work shows the results of the tests on the robots.

INTRODUCTION

L’uso dei robot è ampiamente diffuso in diversi ambiti dell’industria, ma possono essere considerati come un rischio nel caso in cui sia necessaria una cooperazione con un lavoratore. Negli ultimi anni le ricerche e gli studi sulle iterazioni uomo-robot sono aumentate ed una nuova generazione di robot è stata sviluppata: quella dei robot collaborativi.

I robot collaborativi sono nati con lo scopo di lavorare in sicurezza condividendo con delle persone la medesima area di lavoro, pur mantenendo l’efficienza e la produttività tipica dei robot industriali [1]. Ciò è possibile grazie alla sinergia di strategie di controllo e all’uso di specifiche soluzioni meccaniche ed elettroniche, quali possono essere i SEA.

L’iterazione uomo-robot causa un nuovo problema: la interazione tra l’operatore e il robot collaborativo.

La presente tesi affronta il controllo del robot e l’interazione uomo-robot: una strategia di controllo per la guida di robot collaborativi mediante l’utilizzo di sensori EMG e IMU indossabili viene presentato in questo lavoro. L’obbiettivo finale è rendere possibile all’operatore il controllo del comportamento del robot utilizzando i sensori.

Nella primo capitolo viene illustrata la strumentazione utilizzata. Due differenti robot collaborativi a sei gradi di libertà sono stati utilizzati per testare la strategia di controllo. Il primo è stato realizzato all’interno del progetto europeo FourByThree, da cui prende il nome. Degli attuatori SEA (*Series Elastic Actuator*) sono stati utilizzati per la realizzazione del robot; tale particolarità è stata introdotta nell’ottica di aumentare la sicurezza del robot in caso di collisioni e quindi durante le interazioni con l’uomo. Il secondo è un UR10e prodotto da Universal Robots.

Il bracciale Myo, prodotto da Thalamic Labs, contiene al suo interno 8 elettrodi per la rilevazione dei segnali EMG della muscolatura e un IMU a nove assi, a sua volta formato da un accelerometro, un giroscopio e un magnetometro da tre assi ognuno. Questo bracciale è stato utilizzato come dispositivo di rilevazione.

Sia il robot che il bracciale posseggono un’interfaccia all’interno dell’ambiente ROS. Per tale motivo il sistema di controllo finale è stato implementato come un pacchetto ROS, al cui interno è presente un nodo ROS in grado di mettersi in comunicazione con i nodi delle interfacce. Il nodo in questione riceve dal bracciale i dati dell’IMU, nello specifico quelli dell’accelerometro, e il numero identificativo del gesto attuato dalla mano. Infatti, il software di interfaccia del bracciale consente di riconoscere alcuni gesti prestabiliti della mano e associa ad ognuno di essi un numero identificativo. Tali dati vengono elaborati e utilizzati dal sistema di controllo al fine di movimentare il robot.

Seguono due capitoli riguardanti, corrispettivamente, le fasi di sviluppo e quelle di implementazione del sistema di controllo. Entrambi i capitoli sono suddivisi in quattro sezioni nelle quali vengono spiegati gli algoritmi e il codice implementato in ogni fase.

Il sistema di controllo finale si comporta come una macchina a stati con sei stati, ognuno dei quali corrisponde a una serie di operazioni che vengono ciclicamente svolte fino a quanto non avviene un cambio di stato del sistema. Le transizioni tra gli stati sono controllate mediante dei comandi gestuali. Ciò dà all’operatore il potere di determinare il comportamento del robot usando un gesto della mano.

Gli stati del sistema di controllo sono:

- *None*: quando il sistema di controllo è in questo stato non fa nulla e attende il comando di attivazione di una delle modalità di controllo.

- *Vel_*Control: viene controllata la velocità cartesiana del robot. Questo stato acquisisce l'accelerazione lineare del braccio e ne ricava la velocità da passare al robot.
- *Teach*: questo nodo permette la guida manuale del robot. Esso è trascinabile dall'operatore lungo un percorso di cui vengono memorizzate delle posizioni scelte dall'operatore con un gesto della mano.
- *Execution*: le posizioni del robot memorizzate dallo stato *Teach* vengono usate per far svolgere un percorso al robot. Questo nodo si occupa dell'esecuzione del percorso memorizzato.
- *Direction_Control*: in questo caso si è scelto di far muovere il robot di dieci centimetri nella medesima direzione lungo cui si è mosso il braccio. Dall'accelerazione lineare del braccio viene ricavato il versore della movimentazione.
- *Execute_Direction*: nota la direzione è ricavabile una coordinata cartesiana che il robot deve raggiungere. Questo nodo svolge l'esecuzione della movimentazione del robot dalla sua attuale posizione al punto desiderato.

Noto ciò che i singoli nodi della macchina a stati svolgono, è possibile individuare tre modalità di controllo all'interno del sistema:

- Controllo della velocità cartesiana: tale modalità coincide con lo stato *Vel_control*. Risulta essere la prima idea di controllo sviluppata all'interno del presente lavoro.
- Insegnamento attraverso la guida manuale ed esecuzione del percorso: gli stati *Teach* e *Execution* la compongono. Una volta che il percorso è stato memorizzato vi è un'immediata esecuzione da parte del robot. Successivamente è stato introdotto un comando che permette la ripetizione dell'esecuzione dell'ultimo percorso appreso dal robot.
- Controllo della direzione: è l'ultima modalità implementata all'interno di questa tesi. I nodi coinvolti sono *Direction_Control* e *Execute_Direction*. Dopo l'esecuzione della movimentazione lungo la direzione acquisita, il sistema torna in attesa della rilevazione di una nuova direzione fino a quando la modalità non viene disattivata.

Nella fase finale della presente tesi vengono mostrati i dati legati ai test svolti sul robot.

1 Equipment and Software

Collaborative robots are designed to safely interacts with humans without protective barriers that insulate the working space. For this reason, communication between the operator and the robot is important. The interaction may be based on voice and gesture interpretation [2]. This work is focused on gesture interaction. Wearable IMU and EMG sensors can be used for this aim.

This chapter describes the equipment and the software used for this thesis. Sections 1.1-1.X introduce the robots and the sensors device with their corresponding ROS interfaces, while Sections 1.Y-1.Z deal with the description of the ROS environment.

1.1 Collaborative Robots

To test the control strategy developed for this thesis, two anthropomorphous robotic arms are used. They are both collaborative robots and the same set of ROS packages can control them. This section describes the main characteristics of the two cobots and presents the Configuration Manager.

1.1.1 FourByThree

This collaborative robot is born from a European Project named 'H2020-ICT-FourByThree' to find a safe and specific robotic solution for human-robot interaction (HRI). For this reason, this robot is characterized by being moved by six Serial Elastic Actuators (SEA).

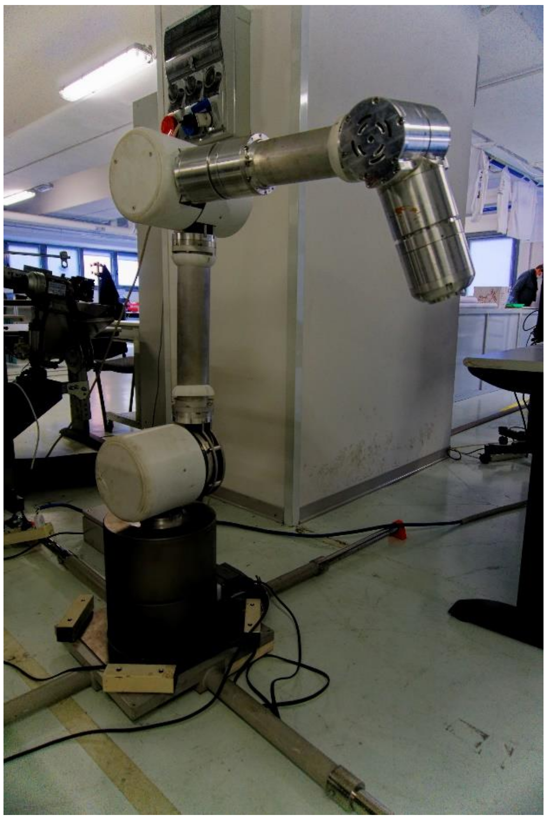


Figure 1-1: FourByThree

The FourByThree Project adopts innovative hardware and software to develop a new generation of modular industrial robots with four different characteristics [2] [3]:

- *Modularity*: each application requires a different solution, which is obtainable with custom robots. They are developed using rotary elastic actuators and software tools for collaborative functions.

- *Safety*: collaborative robots have to work with humans. For this, safe behavior is required during their iterations. Safety strategies are built over elastics actuators, projecting, and vision systems.
- *Usability*: the programming and the x\robot's control are facilitated, providing a set of multimodal interaction mechanisms.
- *Efficiency*: efficient robots have to be reliable, maintainable, and intrinsically safe, particularly if they have to work in the same space with a human.

Modular elements that compose the robot can be classified as follow:

- *Cylindrical links*: they connect joints and their length and diameter vary depending on the applications. The link is hollow, so wires can pass through it.
- *Bracket module*: link modules and actuators are mounted over this module composed of an angled element.
- *Base module*: it is furnished with the LVD-S/Ethernet interface board, which is connected to the control system. Over the base, there is the robot's first actuator.
- *Flange*: it is the interface with the end effector or tools.

1.1.1.1 Serial Elastic Actuators (SEA)

Serial Elastic Actuators (SEA) are used in the joints of the robot. This kind of actuator is operated when high safety is required. This is particularly the case of collaborative robots. Physical interaction between robots and humans imposes limits as reducing contact forces or detect collisions.

A SEA includes an elastic element coaxial with the couple motor-gearbox, reducing the overall rigidity. The presence of elasticity gives different benefits, as an improvement in stock tolerance, low impedance, or the ability to store and then release energy. However, the elastic part creates oscillations, with difficulty in adjusting the load position as a side effect. The spring also introduces a non-linearity in the actuator's model. [4] [5]

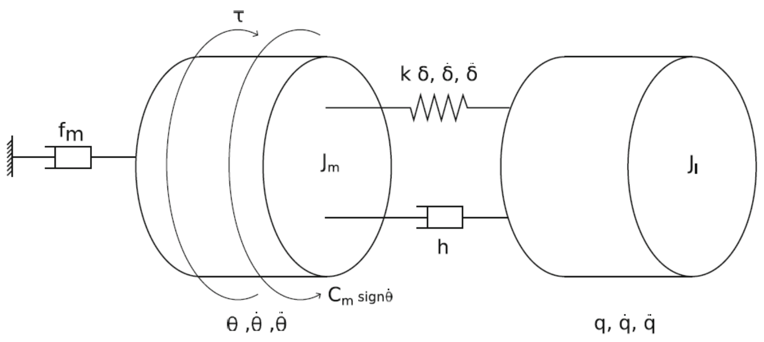

τ
$k\ \delta, \dot{\delta}, \ddot{\delta}$
f_m
J_m
J_l
h
$C_m\ \mathrm{sign}\dot{\theta}$
$\theta, \dot{\theta}, \ddot{\theta}$
$q, \dot{q}, \ddot{q}$

Figure 1-2: Scheme of the SEA model

Three actuator sizes were built for the FourByThree robot with torques of 28Nm, 50 Nm, and 120 Nm respectively.

1.1.2 UR10e - Universal Robots

The UR10e is a cutting-edge robot ideated by Universal Robot for the new e-Series. It is a six DOF robot, and it has a payload of 10 Kg. The UR10e has four principal advantages [6]:

- Fast set-up times.
- Easy programming: the user Teach Pendant interface is intuitive and easy to learn.
- Limitless flexibility: it is a versatile and precise robot, ideated to do any task.

- Safe and collaborative: it can be used on the product line to work safely alongside operators. Safety features are built-in and customizable. It is certificated for ISO 10218-1 and, according to ISO 13849-1, its safety functions are rated Cat3 PL.
- It integrated a force-torque sensor at the end-effector for handling and assembly tasks.

Differently from the FourByThree, this robot integrates rigid actuators in its joints.

ROS drivers are supported for the e-Series. Their features are factory calibration, real-time-enabled communication, integration of the teach-pendant, and the use of speed-scaling. It allows controlling the robot by a ROS Computer at the frequency of 500Hz.

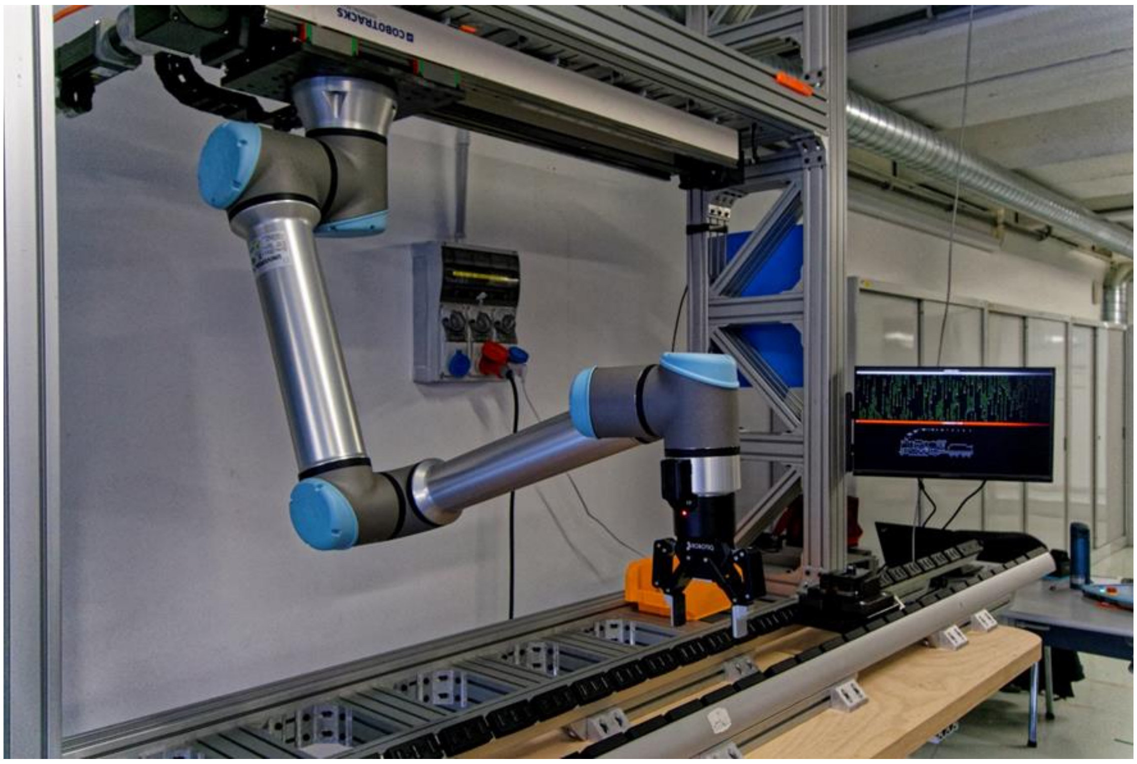


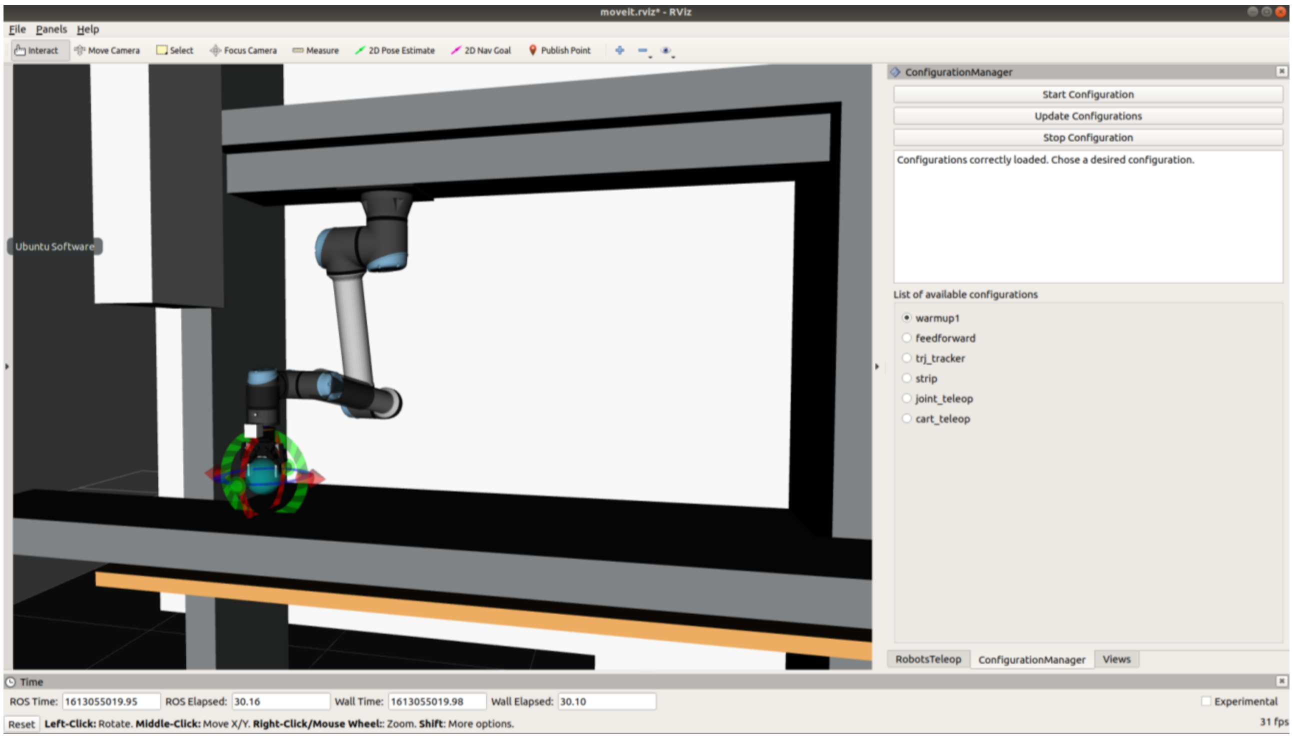
Figure 1-3: UR10e Universal Robots

1.1.3 Configuration Manager

The Configuration Manager is a set of ROS packages design to manage the control system in an agnostic fashion concerning the robot hardware. The robot can be controlled in two main ways: velocity and trajectory (which will be explained later). There are other configurations but, in the end, they all refer to the two specified main types.

The simulator acts as a GUI and allows to choose the specific configuration to use. As shown in Figure 1-4 the possible configurations are:

- *Trj_tracker*: in this modality, the simulator/robot receives the start and the final position of the robot and calculates the trajectory using the Moveit plugin.
- *Joint_teleop*: it allows to control joints velocity. The topic used by this configuration is */planner_hw/joint_teleop/target_joint_teleop*.



Configurations correctly loaded. Chose a desired configuration.

- (●) warmup1
- ( ) feedforward
- ( ) trj_tracker
- ( ) strip
- ( ) joint_teleop
- ( ) cart_teleop

ROS Time: 1613055019.95 ROS Elapsed: 30.16 Wall Time: 1613055019.98 Wall Elapsed: 30.10

Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click/Mouse Wheel:**: Zoom. **Shift**: More options.

Figure 1-4: Sharework_cembre_configuration simulator

- *Cart_teleop*: it sets the cartesian velocity of the gripper and its orientation. The velocity and orientation communication to the simulator pass through the topic */planner_hw/cart_teleop/target_cart_teleop*.
- *Warmup1*: this configuration starts the communication with the robot, with position control disabled.
- *Feedforward*: is like *warmup1* but with position control active.
- *Strip*: the nominal trajectory is deformed according to an algorithm, which detects an obstacle's presence.

1.2 Myo Armband

Thalamic Labs produced a wearable multi-sensor armband called Myo. The armband measures muscle activity and spatial movement, allowing the recognition of the hand's movement and gesture. The used material is a flexible type of elastomer. The Myo kit is fitted with. a Micro USB cable and a Bluetooth adapter [7].

The technical composition is [8]:

- Two lithium battery 3.7 V – 260mAh;
- USB charging port;
- A vibration motor;
- BLE NRF51822 chip for the wireless transmission;
- Freescale Kinetis ARM Cortex M4 120Mhz MK22FN1M MCU processor;
- Eight medical grade stainless steel EMG sensors;
- Eight ST 78589 operational amplifier, one for each electrode;
- Inven-sense MPU-9150 at nine axes IMU;
- Dual indicator LEDs.

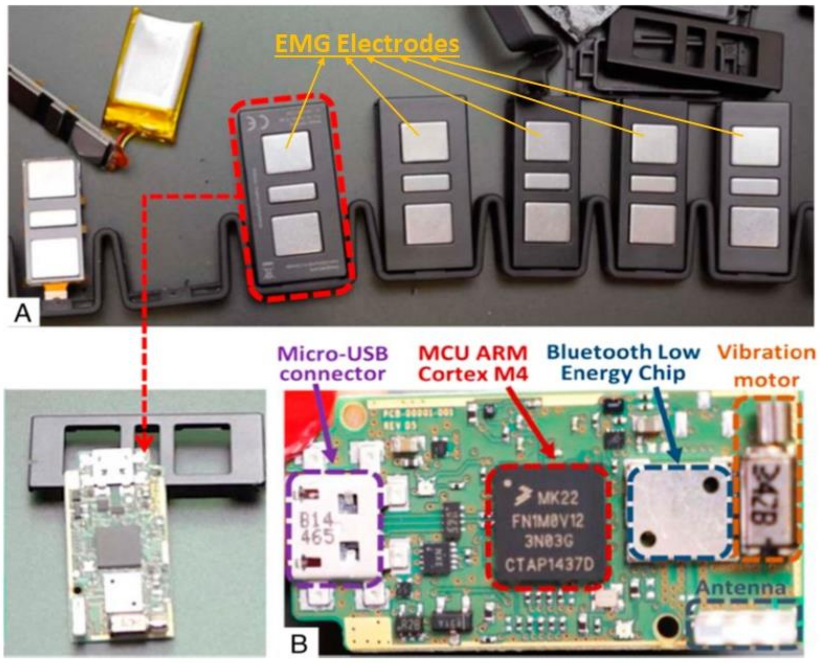


Figure 1-5: Myo Armband composition

Sync state is shown by the logo LED that is static when Myo is synced and pulses when it is not. The other LED shows Myo state by changing color. The LED is blue when the device is connected with Bluetooth, orange when charging and green when fully charged.

Short, medium, and long vibrations give haptic feedback, for example, on sensors or actuators fault, low battery, or high temperature.

The main sensors are the eight electromyograph sensors and the nine-axis inertia measurement unit. The latter is in turn composed of a 3-axes MEMS accelerometer, a 3-axes MEMS gyroscope, and a 3-axes MEMS magnetometer. IMU sampling frequency is 50 Hz.

The EMG electrodes have their surface split into two kinds: the active one and the passive. The formers are constituted of three conductors located along with a muscle: the central one detects muscular activities, and the other act as a noise filter. The passive electrodes have only one conductor and need conductive materials to couple with skin. The sampling frequency of the EMG signal is 200 Hz.

1.2.1 Ros_Myo Package

The interface used is a ROS package based on the homonymous package, available on Github and developed by uts-magic-lab [9]. The modified version is forked at https://github.com/JRL-CARI-CNR-UNIBS. It is written in Python. It creates a ROS node that publishes raw data from the Myo Armband.

The topics generated by the original myo_raw node are:

- */myo_raw/myo_arm*: the message type is *ros_myo/MyoArm,* and it tells on which arm the armband is supposedly worn and the direction of the X-axis.
- */myo_raw/myo_emg*: the EMG readings are published by using *ros_myo/EmgArray* messages.
- */myo_raw/myo_gest*: a *ros_myo/MyoPose* message declare the pose/gesture ID number.
- */myo_raw/myo_gest_str*: in this topic is published a *std_msgs/String* with the pose/gesture name.
- */myo_raw/myo_imu*: a standard IMU message with acceleration expressed in m/s^2, angular velocity in rad/s and orientation as a quaternion. The measure of acceleration includes the constant of gravity.

- */myo_raw/myo_ori*: the message kind is *Vector3* and it contains roll, pitch, yaw in radians.
- */myo_raw/myo_ori_deg*: is the same message of *myo_ori*, but with data converted in degrees.
- */myo_raw/vibrate*: it is a *UInt8* topic used to publish a vibration to make. It accepts as values the numbers from 1 to 3, the higher the value, the longer the vibration.

IMU data are expressed according to the internal reference system of the armband, called *myo_raw*, that is the one showed in Figure 1-6.

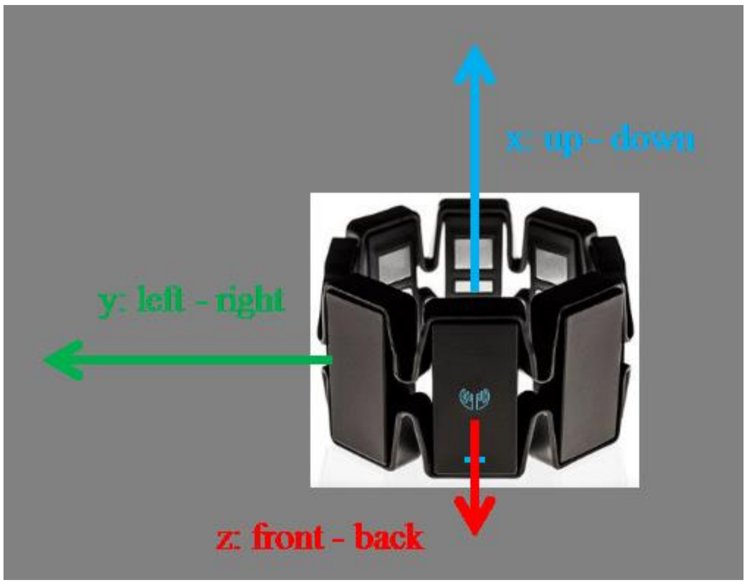


Figure 1-6: Internal reference system of the IMU of Myo Armband

The main change applied to the original code is the linear transform of IMU data according to the global reference system, which is the same used for the robot. Gravitational acceleration has been managed as an offset. The following topics have been added:

- */myo_raw/myo_imu_global*: standard IMU messages are published on the topic. Data are expressed in agreement with the global reference system.
- */myo_raw/acc_twist_global* and */myo_raw/acc_twist_imu*: both the topics are used to publish TwistStamped message with linear acceleration, the first according to the global reference system, the other to the internal one.
- */myo_raw/vel_twist_global* and */myo_raw/vel_twist_imu*: they work as the topics utilized for acceleration, but with the angular velocity.

Another amendment to the original code concerns the publication of the *MyoPose* message in *myo_gest* topic. Myo armband has six default recognizable gestures and movements, and each of them has an ID number. The number is published when the movement to the pose is detected, but not if the gesture is maintained. The change consists of publishing the ID number continuously while a gesture switch is recognized.

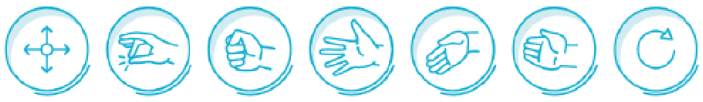


Figure 1-7: The default recognizable gestures and movements of the Myo armband.

1.3 Robot Operating System (ROS)

Figure 1-8: ROS logo

Every robot needs software to operate, ROS is a development framework to write them. It is a meta-operating system, this means that it provides standard operating systems utilities, like the control of low-level devices or hardware abstraction, and high-level features, such as asynchronous and synchronous calls or a robot configuration system. ROS gives tools, libraries, and conventions to simplify the writing of robust, general-purpose robot software. [10] [11] [12]

ROS environment is based on some principles which affect its structure and performance.

- *Peer-to-peer architecture*: the direct dialogue between the components is allowed by a lookup system called '*Master*', which is the nucleus of this kind of architecture.
- *Multi-language*: it accepts different programming languages, like C++, Python, or Java.
- *Tools-based*: ROS is structured like a microkernel. Every component is made up of a huge number of small tools. In this way, the system is more robust and flexible because if a problem occurs in an executable, the others are not affected by it.
- *Thin*: the libraries contain the complexity of the algorithms, some of them are contained in standalone executables, then some drivers allow the reuse of the code. In this way, ROS becomes an easy platform.
- *Free and open-source*: it is based on Unix platforms and it is open-source.

1.3.1 Communication between processes

There is a level for communication between processes in ROS architecture, named "*ROS Computation Graph Level*". It is a peer-to-peer network based upon seven concepts: nodes, master, parameter server, messages, services, topics, and bags.

- *Nodes*: they are processes that perform computation. Specifically, they are instances of executables. A node communicates with others through topics, it has to declare itself to the Master to use them.
    - *Master*: it is a node that controls the ROS communication level. It provides a registration service and allows the nodes to find each other and interact or invoke services.
    - *Parameter Server*: it is part of the Master and it is involved in the storage by a key of data in a central location.

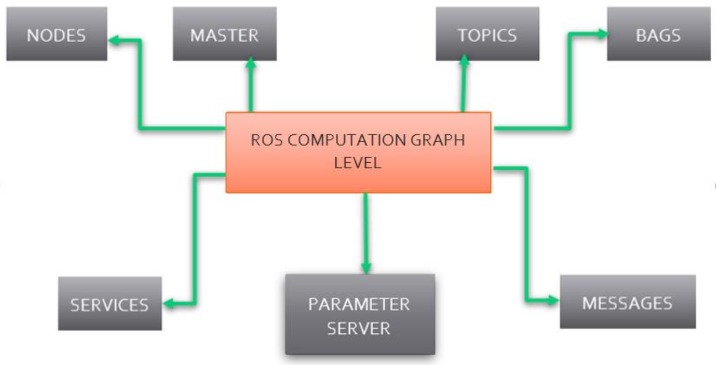


Figure 1-9: ROS Computation Graph Level scheme

- *Messages*: data are sent using predefined combinations of types and messages. Massages are data structures.
- *Topics*: they identify the content of a message. They are based on subscribe/publish logic: a node can register itself to a topic as a "*Subscriber*" if it is interested in receiving a kind of data, or as a "*Publisher*", which can generate new messages in the topic. Communication is unidirectional from the publishers to the subscribers.

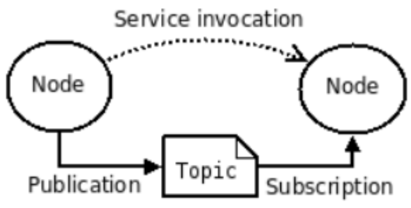


Figure 1-10: Nodes communication through a topic

- *Services*: nodes use them when interactions of the kind request/reply are necessary.
- *Bags*: Bags are a format for saving and playing back ROS message data.

The ROS Master acts as a DNS server, it gives only lookup information to nodes and they connect directly to each other. The Master stores and continuously update nodes, topics, and services registration information. ROS Master can provide data during the conversation with nodes and informs them about changes. In this way, appropriate connections are made by nodes. Names are essential to build up this kind of architecture.

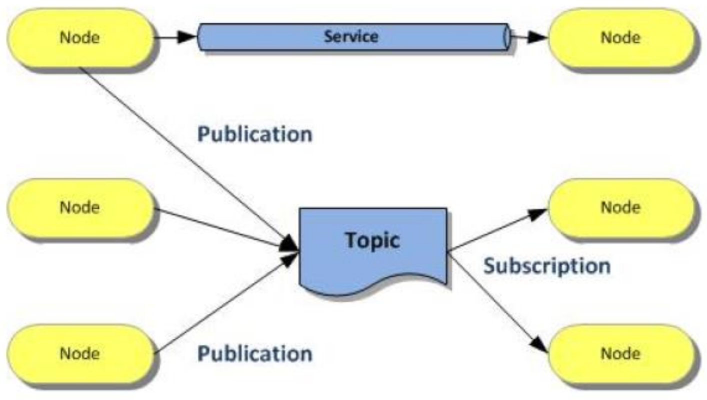


Figure 1-11: Relations between ROS environment elements

1.3.2 ROS on Windows

ROS is a native Linux software. If a Windows computer is available, there are several ways to operate on it with ROS:

- *Dual boot*: disk is partitioned and a Linux operating system is installed on the partition. This can cause overheads, in particular for the computer memory. It is useful if the Linux part does not have to interact with the Windows one.
- *ROS for Windows*: not all the tools available for the Linux version are compatible with this one.
- *Virtualization*: there is an abstracted layer from the hardware in which is possible to run a virtual instance. Virtualization is commonly used to run multiple operating systems at the same time. This option can cause overheads, particularly for the CPU. [13]

A computer with Windows was available for this work. Virtualization is the option that has been deepened, two paths have been followed: the first one was the Windows Subsystem for Linux, the second one was a virtual machine.

1.3.2.1 Windows Subsystem for Linux - WSL2

The Windows Subsystem for Linux is developed to run a GNU/Linux environment directly on Windows. This makes it possible to avoid overhead characteristics of a virtual machine or dual boot. [14]

WSL 2 is the newest version and uses Hyper-V architecture to enable virtualization. Hyper-V is a hypervisor, namely software that generates and runs virtual machines. It is a type 1 hypervisor, which means that it runs directly on the host's hardware.

WSL 2 requires at least Windows Version 1903 with build 18362 for x64 systems. It was developed to run only command-line tools, it does not support graphics applications. This problem is pierced by installing an X-server application and by adding the following lines to the end of the .bashrc file.

```
export DISPLAY="`grep nameserver /etc/resolv.conf | sed 's/nameserver //'`:0";
export LIBGL_ALWAYS_INDIRECT=0;
export XDG_RUNTIME_DIR=/tmp/runtime-userName;
export RUNLEVEL=3;
```

'DISPLAY' is an environment variable that instructs WSL2 where graphics information needs to be sent. The first command-line automatically takes the Windows host IP and utilizes it to set the variable. The second line is useful to avoid issues with OpenGL applications, such as Rviz, while using an NVIDIA video card driver. The other two lines set two variables related to the use of a display and graphics applications.

The Windows Subsystem for Linux has multiple advantages, but it cannot directly see a /dev/ttyACM0 communication device, which is the one used by the Myo Armband. This because WSL2 does not have complete access to hardware. Unfortunately, no solution has been found to this problem, so the WSL option has been discarded.

1.3.2.2 Virtual Machine - Virtual Box

Figure 1-12: VirtualBox Logo

A virtual machine can be considered as an emulation of a computer system. It is a type 2 hypervisor, which means that it runs as any computer program does. The virtual machine operating system runs as a process on the host. [15]

For this work, Virtual Box was used with Ubuntu 18.04 installed. It has better access to USB ports compared with WSL2, this lets communication with Myo Armband possible. This kind of virtual machine is not compatible with Hyper-V, for this reason, it is good to check that it is disabled.

1.3.3 Project Environment

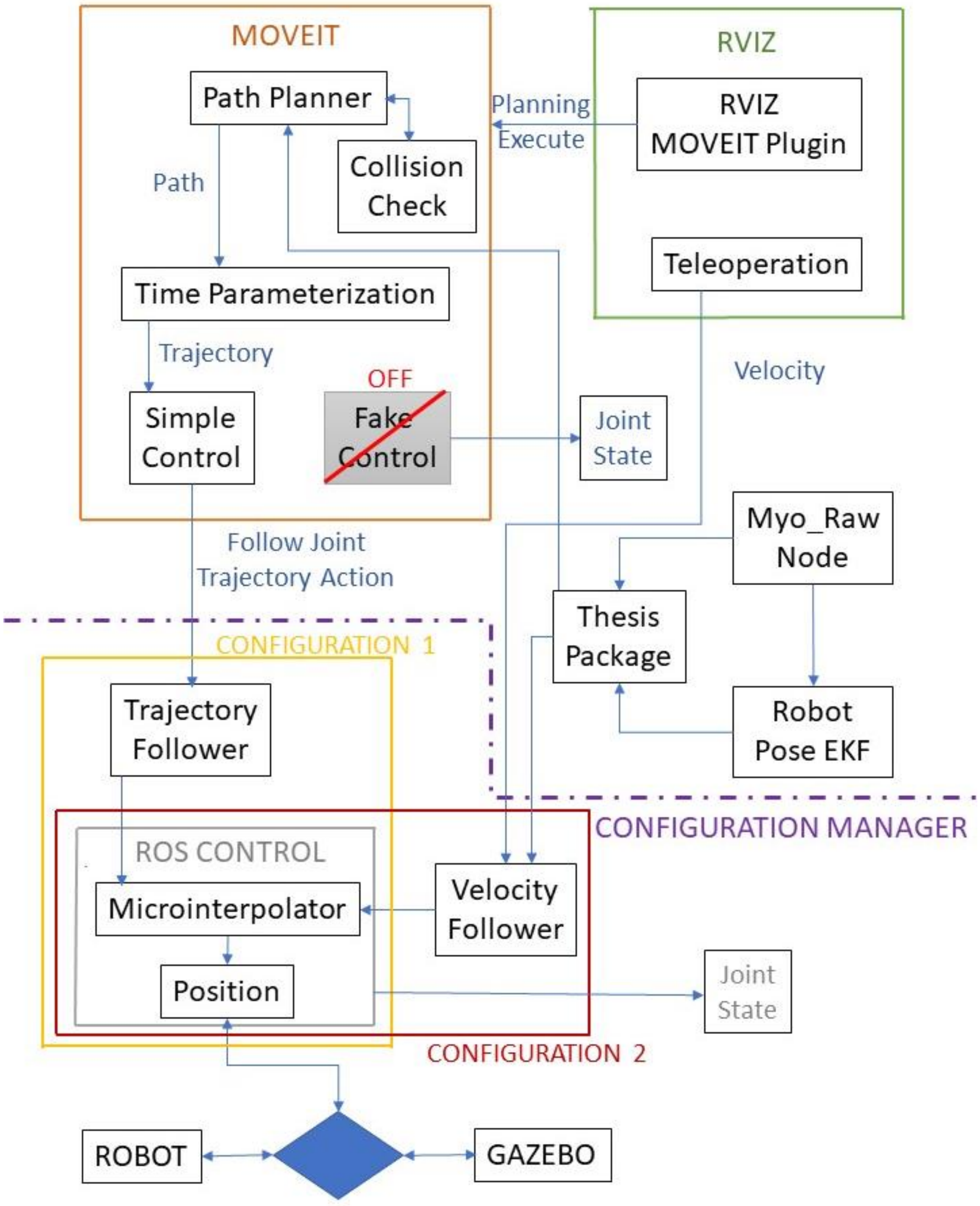


Figure 1-13: Environment scheme

This thesis aims to produce a ROS package that gathers data produced by the Myo_Raw Node and turns them into a signal used to control the robot. The Myo_Raw Node is the ROS node inside the interface package that communicates with the IMU inside the Myo.

The Configuration Manager, as said before, consists of more packages. It is structured to have two main configurations. The input of the first configuration is a trajectory that the robot has to follow; the second one receives a velocity. Usually, the trajectory is generated by Moveit and the velocity by Rviz. The two configurations use the same microinterpolator embedded in the ROS Control package, but they differ for the module that receives the Configuration Manager input data.

The package is build up to fit in this structure and sends information for the trajectory to Moveit or directly to the Configuration Manager. Particularly a cartesian velocity information is obtained integrating data from the accelerometer and it is passed to the robot Configuration Manager.

1.3.4 MATLAB

Figure 1-14: MATLAB logo

MATLAB is software for iterative analysis and design processes. It supports a connection interface to ROS, called '*ROS Toolbox*', through which MATLAB generates a network of ROS nodes and allows topic subscription and message publication. [16]

The first tests on the bracelets were carried out through MATLAB scripts. MATLAB starts a node connected to the master by the following commands.

```
try
    rosinit('10.0.0.215','NodeHost', '192.168.56.1');
end
```

If the ROS master runs on the same machine, it is enough typing `rosinit` with no parameters. In other cases, it is necessary to specify the ROS master IP end the host machine IP.

MATLAB nodes can communicate with other ROS nodes through topics. Publishers and subscribers can be created. The following code is an example of a publisher.

```
joint_setpoint_pub=rospublisher('/manipulator/joint_target');
joint_setpoint_msg=rosmessage('sensor_msgs/JointState');
joint_setpoint_pub.send(joint_setpoint_msg);
```

The first line creates an object representing a ROS publisher for the topic named `/manipulator/joint_target`. The last line sends to the topic a message generated with `rosmessage`. The type of the message is the parameter of this function.

```
js_sub = rossubscriber('/manipulator/joint_states');
js_msg=js_sub.receive;
```

`rossubscriber` inscribe the node to the topic as a subscriber. The function in the second line waits and receives the first message published in the topic after the command has been launched.

The ROS network in MATLAB is stopped when the command `rosshutdown` is launched.

2 Algorithms For Program Development

The development of the final program took place through several stages. To each stage correspond an algorithm. The first algorithm represents an idea for the control strategy, the others are upgrades of the first one. Each stage corresponds to a program as explained in the next chapter.

2.1 Cartesian Velocity Control

The first idea was to take the speed of the arm and use it to control the gripper's cartesian velocity.

As explained in the previous chapter, there is an IMU inside the Myo, and it provides an accelerometer. Arm speed is obtainable by integrating linear acceleration.

Figure 2-1 shows the algorithm developed. The software has to communicate with ROS, in the specific with the *myo_raw* node and with the Configuration Manager. The first part of the program is the setting of all the subscribers, the publishers, and the messages needed for this task.

The second part is the initialization of all the variables used inside the program. These two parts are important for the proper functioning of the software, and they require particular attention in the implementation. They are present in all the following upgrades of the algorithm.

The core of the program is inside the loop. The movements of the arm will control the robot, but maybe not all the actions are desired control commands. For this reason, a command gesture is introduced. In this case, the chosen gesture is the fist, which corresponds to ID number two.

If the operator does a fist, its arm movement is detected and used until the gesture change.

Acceleration data are affected by armband noise, therefore it is good to filter them. The arm is never totally stationary, the IMU always detects some micro-movements. For this, a dead-band around zero is embedded with the filter. Dead-band allows to cut off the micro-movements while the arm is almost still.

After data processing, the acceleration is integrated to obtain the velocity. Robot joints have a maximum velocity. To ensure that these limits are not exceeded, a saturation is placed at this point of the algorithm. Saturation also acts as a safety limit in case of collision.

The saturated signal is passed to the robot by the */planner_hw/cart_teleop/target_cart_teleop* topic. Its data type is TwistedStamped, which contains linear and angular velocity.

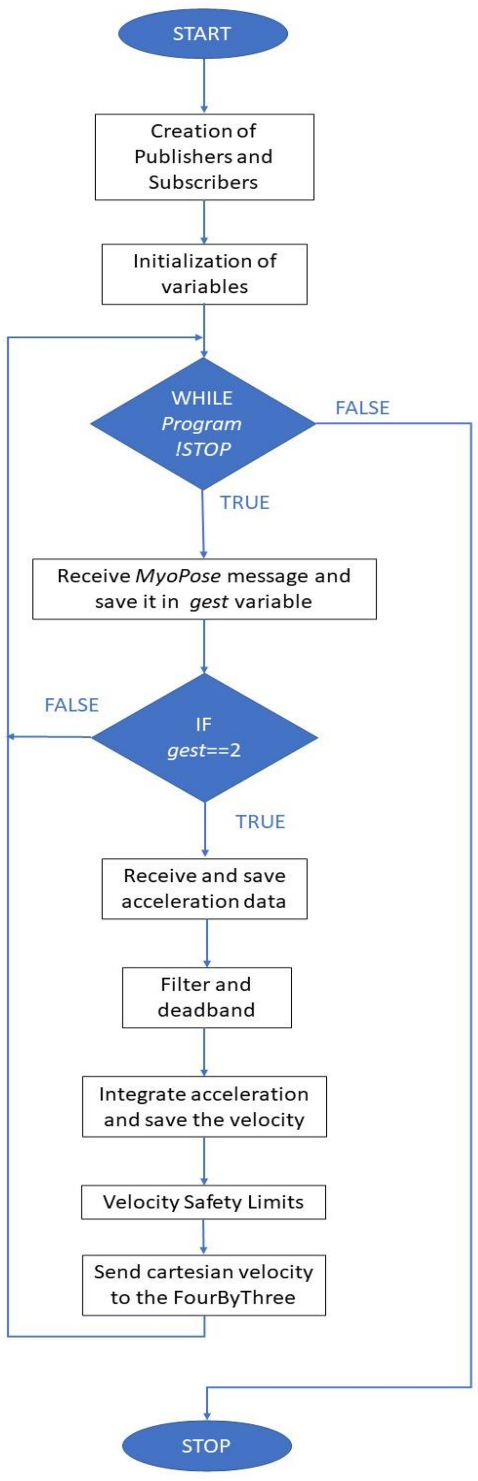

START
Creation of Publishers and Subscribers
Initialization of variables
WHILE *Program !STOP*
FALSE
TRUE
Receive *MyoPose* message and save it in *gest* variable
FALSE
IF *gest*==2
TRUE
Receive and save acceleration data
Filter and deadband
Integrate acceleration and save the velocity
Velocity Safety Limits
Send cartesian velocity to the FourByThree
STOP

Figure 2-1: First Algorithm

2.2 TEACHING BY MANUAL GUIDANCE

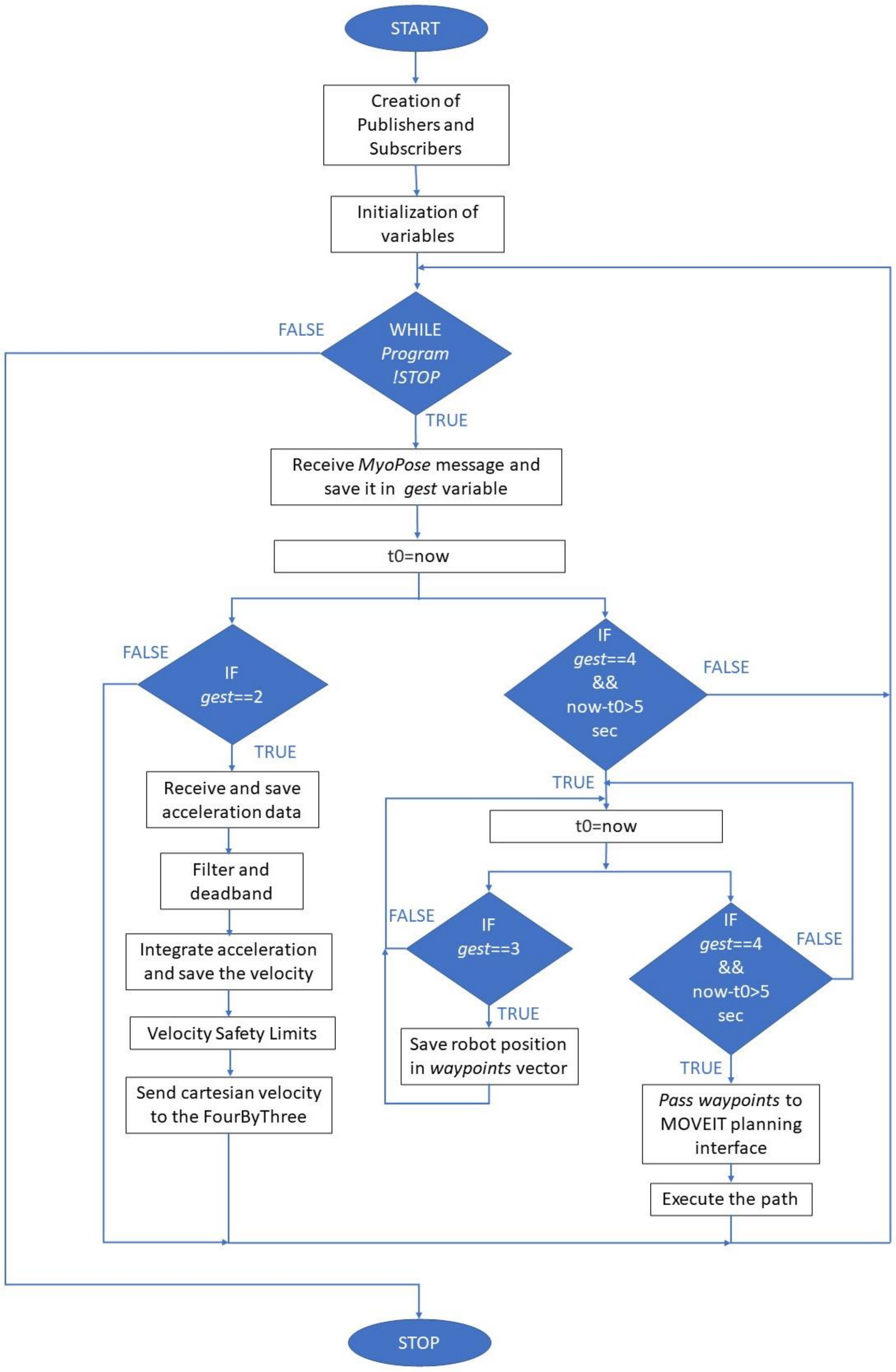

START
Creation of Publishers and Subscribers
Initialization of variables
WHILE Program !STOP
FALSE
TRUE
Receive MyoPose message and save it in gest variable
t0=now
IF gest==2
FALSE
TRUE
Receive and save acceleration data
Filter and deadband
Integrate acceleration and save the velocity
Velocity Safety Limits
Send cartesian velocity to the FourByThree
IF gest==4 && now-t0>5 sec
FALSE
TRUE
t0=now
IF gest==3
FALSE
TRUE
Save robot position in waypoints vector
IF gest==4 && now-t0>5 sec
FALSE
TRUE
Pass waypoints to MOVEIT planning interface
Execute the path
STOP

Figure 2-2: Algorithm with teaching by manual guidance

Collaborative robots can be programmed by teaching [1]. It means that the robot is moved to some poses that are memorized by pressing a button on the teach pendant. This is called direct teaching when the operator manually moves the robot. The robot's gripper is attached a force sensor to push or pull the robot in the desired position.

The second idea was to add to the algorithm a control procedure based on teaching by manual guidance. A command gesture replaces the press of the button on the teach pendant. MoveIt! will calculate the path between stored poses!. The robot teaches pendant can be bulky during the manual guidance so that this replacement can be an advantage.

Figure 2-2 shows the new algorithm, which is an amendment to the previous one. The direct teaching is implemented inside the loop as a parallel algorithm to the control velocity one. As before, a gesture is chosen to activate the new control configuration. The activation condition is to hold the wave out position (ID number: four) for at least five seconds. To do this, it is useful a time variable updated at each cycle to the present time. The same condition is used to exit the direct teaching. In this way, the feature cannot be activated by mistake.

After activation, the robot can be manually moved. When the wave in gesture (ID number: three) is detected, the robot joints configuration is saved in the *waypoints* vector. Before the exit from the configuration, the vector is passed to the Moveit Planning Interface. It calculates the cinematics and the dynamics of the robot and sends them to the Configuration Manager. The path done with the manual guidance is executed by the robot. It is possible to switch from a control method to the other until the program stops.

2.3 Path Repetition – State Machine

The algorithm just presented does not allow the repetition of the memorized path before the execution. So the idea is to add this chance to the algorithm.

As shown in Figure 2-3, this introduces only a condition from which it follows the reiteration of a section of the algorithm, and therefore of the code, already existing.

Adding new features increases the complexity of the algorithm. An equivalent logical structure can be used during the implementation. It has to advance the reusability of the code and simplify its extension or amendment. The algorithm is convertible in a finite-state machine model. It is an abstract device, so a theoretical model of a software system and its behavior.

A state machine is made upon a series of states, machine's input, and transition functions. A state describes one internal situation or configuration of the system. The transition of the system from a state to another depends on the machine's actual state and input values. The transition functions are the conditions for the state change. [17]

The control system in the algorithm can be developed in a finite-state machine with four states related to the control methods. They are:

- *None*: in this case, the control system does nothing. Thus the robot stands still.
- *Vel_Control*: this state corresponds to the cartesian velocity control. While the control system is in this state, acceleration data are saved, processed, and integrated. Then the cartesian velocity is passed to FourByThree.
- *Teach*: the control system saves robot poses achieved during the manual guidance into the *waypoints* vector.
- *Execution*: in this state, the *waypoints* vector is passed to the Moveit Planning Interface, and the resulting path is executed.

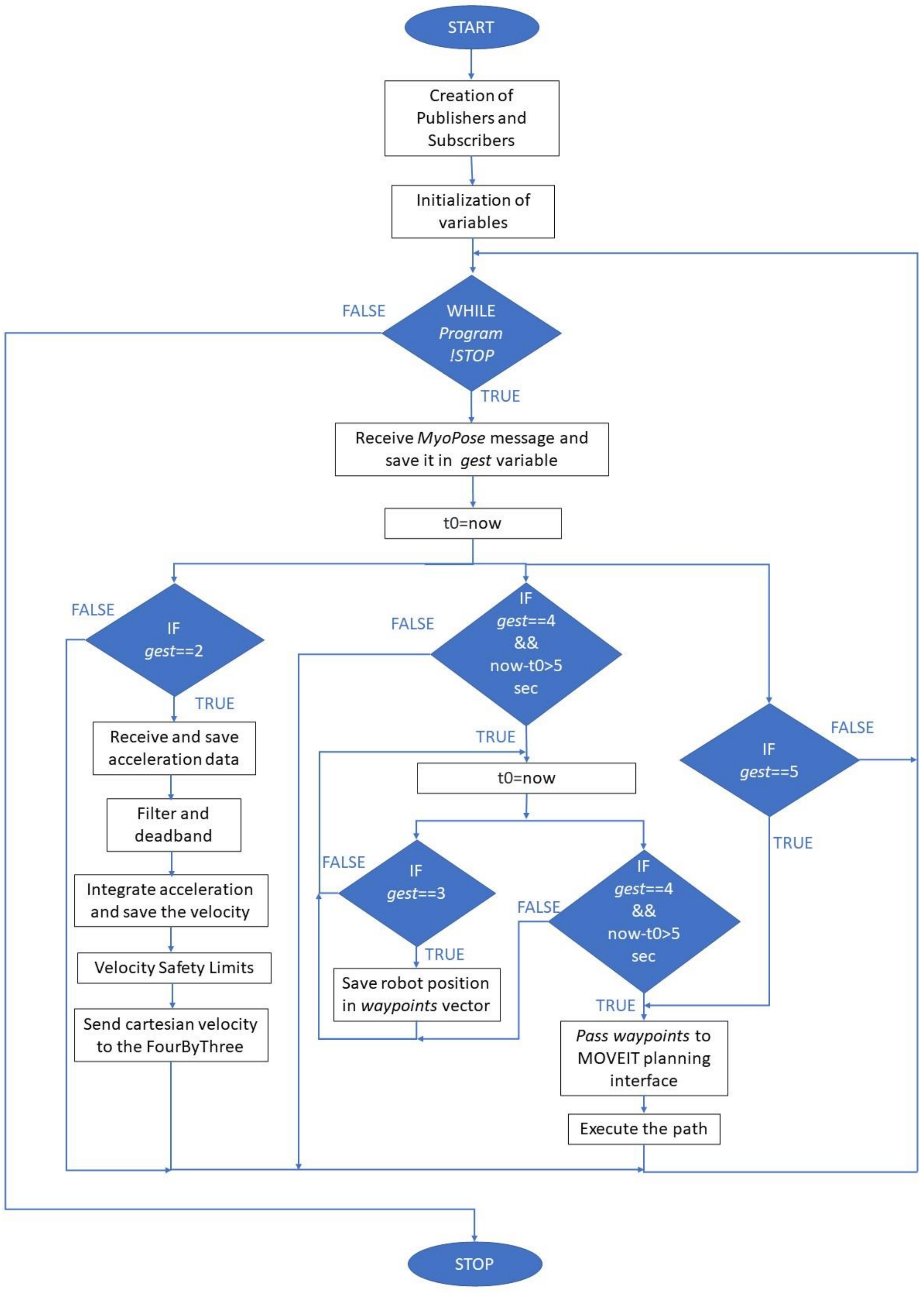

START
Creation of Publishers and Subscribers
Initialization of variables
WHILE Program !STOP
FALSE
TRUE
Receive MyoPose message and save it in gest variable
t0=now
IF gest==2
FALSE
TRUE
Receive and save acceleration data
Filter and deadband
Integrate acceleration and save the velocity
Velocity Safety Limits
Send cartesian velocity to the FourByThree
IF gest==4 && now-t0>5 sec
FALSE
TRUE
t0=now
IF gest==3
FALSE
TRUE
Save robot position in waypoints vector
IF gest==4 && now-t0>5 sec
FALSE
TRUE
Pass waypoints to MOVEIT planning interface
Execute the path
IF gest==5
FALSE
TRUE
STOP

Figure 2-3: Algorithm with path repetition

Figure 2-4 shows the correspondence between the states and parts of the algorithm in Figure 2-3.

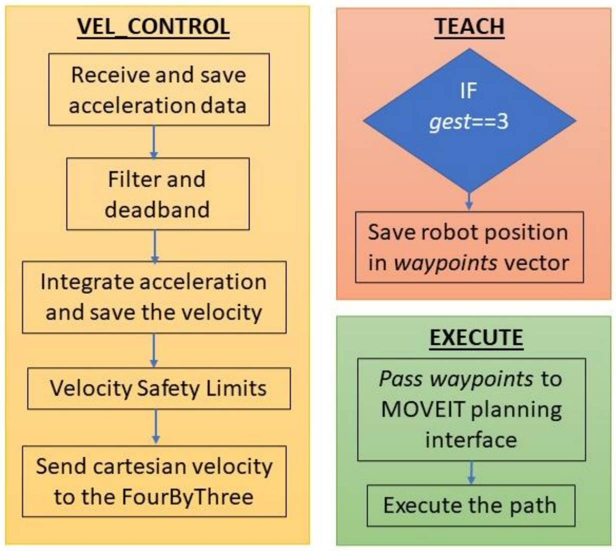

VEL_CONTROL
Receive and save acceleration data
Filter and deadband
Integrate acceleration and save the velocity
Velocity Safety Limits
Send cartesian velocity to the FourByThree
TEACH
IF gest==3
Save robot position in waypoints vector
EXECUTE
Pass waypoints to MOVEIT planning interface
Execute the path

Figure 2-4: States algorithms

The main inputs of the machine are the hand gesture ID (saved in the variable *gest*), the actual time, and the time variable *t0*. This last input saves actual time when the gesture ID number is different from four. Inputs values are constantly updated and used to verify the transition functions.

The transitions functions compare *gest* and the difference between the actual time and *t0* with some constant values.

Figure 2-5 shows the machine state behavior. When the system starts, its state is None. From this state, there are three possible transitions:

- *To Vel_Control*: the condition is gesture ID number equal to two.
- *To Execute*: the condition is gesture ID number equal to five
- *To Teach*: there are two conditions. The ID number has to be equal to four for at least five seconds.

From *Vel_Control* the system can only return at None, same from Execute. For this transition, it is necessary to introduce a machine input that indicates the end of robot handling. From *Teach*, it is possible only a transition to *Execute*.

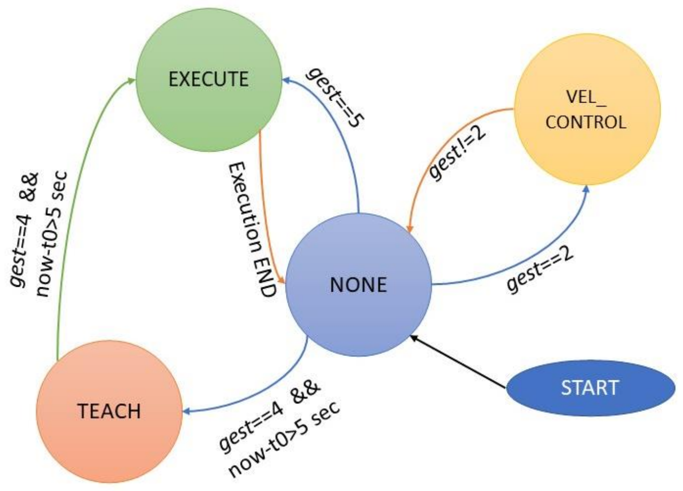


Figure 2-5: State machine diagram

2.4 Direction Control State

The final design of the control system introduces another functionality. The robot gripper moves ten centimeters in the same direction as the arm movements.

Two states are added to the state machine:

- *Direction_Control*: this state operates similarly to *Control_Vel*, but adds the velocity to obtain the position and determines the versor of position variation. The direction is taken only when the fist is detected as in the cartesian velocity control. Otherwise, the control system waits.
- *Execute_Direction*: it is like the *Execute* state, but, in this case, it passes the Moveit Planning Interface the and-effector position. When the path is planned, the robot actuates it.

The two states act as a single control modality. When a direction is identified, it is executed by the robot. After the execution, the control system waits for the detection of a new direction. There is a cyclic exchange between these states until the control modality is deactivated.

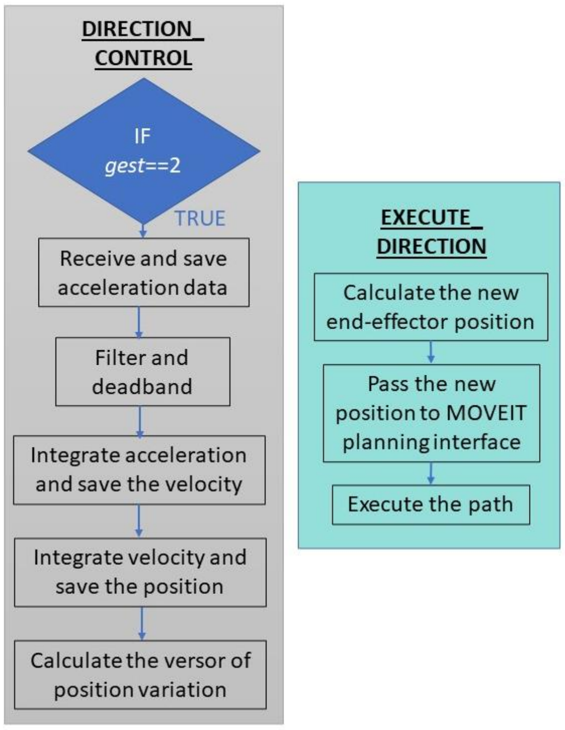

DIRECTION_ CONTROL
IF gest==2
TRUE
Receive and save acceleration data
Filter and deadband
Integrate acceleration and save the velocity
Integrate velocity and save the position
Calculate the versor of position variation
EXECUTE_ DIRECTION
Calculate the new end-effector position
Pass the new position to MOVEIT planning interface
Execute the path

Figure 2-6: New states algorithms

The inclusion of these states in the machine changes the behavior of the control system as shown in Figure 2-7.

The activation and the disable of this feature use the same logic of the transition from the state *None* to *Teach*, but with the wave-in gesture (ID number: three). At the feature start, there is a transition from *None* to *Direction_Control*. The transition from *Direction_Control* to *Execute_Direction* and the vice-versa requires introducing two new inputs: a variable that indicates when a direction is taken and another one that indicated the execution end.

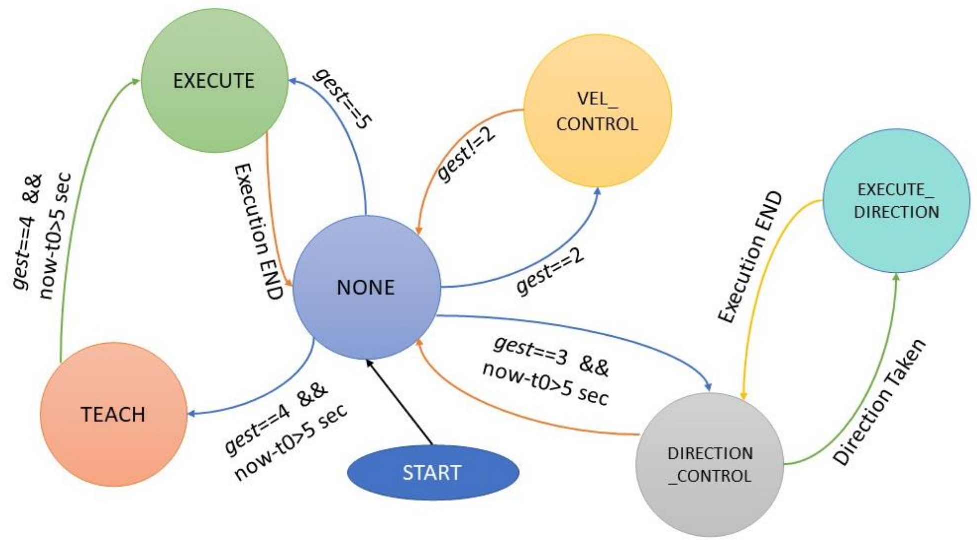


Figure 2-7: State machine diagram

3 DEVELOPED PROGRAMS

This chapter explains the implementation as a C++ code of the algorithms shown in the previous chapter. The control system has been developed as a ROS package, and the explained codes are in the executable file of the package.

The code below is the general structure of the main function in the executable. In all the implementations, the main begins with the initialization and the start of the ROS node of the control system (lines 3 and 4). In the main, there is a *while* loop, of which the exit condition is the node shutdown (line 11). Loops need a cycle frequency to runs at. The command at line 7 fixes the loop frequency. The algorithms are sets of cyclic operations, therefore the implemented code is inside the loop. The chosen cycle time is a hundredth of a second, so the frequency is 100 Hz.

```
 1  int main(int argc, char **argv)
 2  {
 3    ros::init(argc, argv, "imu_teleop_node");
 4    ros::NodeHandle nh;
 5
 6    double st=0.01;
 7    ros::WallRate lp(1.0/st);
 8
 9    ...
10
11    while (ros::ok())
12    {
13      ...
14    }
15
16    return 0;
17  }
```

Before the loop, services, subscribers, publishers, and variables are declared and initialized. Those are operations necessaries for the correct behavior of the program. The following code is a series of examples of those operations written in C++.

```
 1  ros_helper::SubscriptionNotifier<sensor_msgs::JointState>
        js_sub(nh,"/manipulator/joint_states",1);
 2  if (!js_sub.waitForANewData(ros::Duration(10)))
 3  {
 4    ROS_ERROR("No topic received");
 5    return 0;
 6  }
 7
 8  ros::Publisher jteleop_pub=nh.advertise<sensor_msgs::JointState>
        ("/planner_hw/joint_teleop/target_joint_teleop",1);
 9
10  ros::ServiceClient configuration_client=nh.serviceClient<configuration_msgs:
        :StartConfiguration>("/configuration_manager/start_configuration");
11
12  if (!nh.getParam("scale_coef",coeff))
13  {
14    ROS_WARN("scale_coef not set, using 0.01");
15    coeff=0.01;
16  }
```

The first line is the creation of a subscriber. For the proper program functioning, the subscriber must receive data from the topic. If this does not happen, an error warning is printed (line 4), and the node dies (line 5). Line 8 is the creation in C++ of a publisher, and line 10 of service. This one is important to communicate the desired robot configuration to the Configuration Manager.

A variable can be a parameter. After the declaration, it is possible to set the variable value equal to a parameter or fix it to a predefined value if it is not found. That is showed in the code from lines 12 to 16.

3.1 Cartesian Velocity Control

The names of the topics to which the node have to subscribe are: */myo_raw/acc_twist_global*, and */myo_raw/myo_gest*. It publishes on */planner_hw/joint_teleop/target/cart_teleop*.

The program core is inside the loop. It begins memorizing the latest gesture ID number sent to the topic in the variable *gest*. Two other variables manage the operations done during the cycle: *do_disable* and *disable*. They are both booleans initialized to false.

```
 1  while (ros::ok())
 2  {
 3    ros::spinOnce();
 4    ros_myo::MyoPose gest=myo_pose_sub.getData();
 5
 6    if (gest.pose==ros_myo::MyoPose::FIST)
 7    {
 8      do_disable=false;
 9      disabled=false;
10      ROS_INFO_THROTTLE(1,"actived");
11
12      ... data processing code ...
13    }
14    else
15    {
16      ROS_INFO_THROTTLE(1,"disable");
17      acc_in_g.setZero();
18      acc_in_g_filt.setZero();
19      acc_in_g_satured.setZero();
20      acc_in_b_satured.setZero();
21      acc_in_b_satured_old.setZero();
22      vel_in_b.setZero();
23      do_disable=true;
24    }
25
26    if (!disabled)
27    {
28      geometry_msgs::TwistStamped twist;
29      twist.header.frame_id="BASE";
30      twist.header.stamp=ros::Time::now();
31      twist.twist.linear.x=vel_in_b(0);
32      twist.twist.linear.y=vel_in_b(1);
33      twist.twist.linear.z=vel_in_b(2);
34      cteleop_pub.publish(twist);
35    }
36    if (do_disable)
37      disabled=true;
38
39    lp.sleep();
40  }
```

Acceleration and velocity data are stored inside Eigen::Vector3d variables. When the chosen command gesture is not detected, all the vectors are reset to zero (lines from 14 to 24) and the two booleans variables to true. In the other case, *do_disable* and *disable* turn to true and the communication of the cartesian velocity to the robot is enabled (lines from 26 to 35). Data processing is done while the gesture is identified as a fist. The following code shows this procedure which starts with the storage into an Eigen::Vector3d of the acceleration compared to the global reference system.

```
1       geometry_msgs::TwistStamped imu=imu_sub.getData();
2       acc_in_g(0)=imu.twist.linear.x;
3       acc_in_g(1)=imu.twist.linear.y;
4       acc_in_g(2)=imu.twist.linear.z;
5       acc_in_g_filt=a*acc_in_g_filt+(1-a)*acc_in_g;
6       for (int idx=0;idx<3;idx++)
7       {
8         if ((std::abs(acc_in_g_filt(idx))<noise) && (std::abs(acc_in_g(idx))<noise))
9           acc_in_g_satured(idx)=0;
10        else
11          acc_in_g_satured(idx)=acc_in_g_filt(idx);
12      }
13
14      acc_in_b_satured=T_b_g*acc_in_g_satured;
15      vel_in_b=vel_in_b+coeff*st*0.5*(acc_in_b_satured+acc_in_b_satured_old);
16      for (int idx=0;idx<3;idx++)
17      {
18        if (vel_in_b(idx)>vel_max)
19          vel_in_b(idx)=vel_max;
20        else if (vel_in_b(idx)<-vel_max)
21          vel_in_b(idx)=-vel_max;
22      }
23      acc_in_b_satured_old=acc_in_b_satured;
```

3.1.1 Discrete First Order Low-Pass Filter

After the storage, the first step is the filtering of the received signal. Sensor output is generally affected by noise, which can be considered a high-frequency signal overlaid to the accelerometer signal. A low-pass filter reduces signals with frequencies higher than a certain level. It is indicated to filter the noise.

A first-order filter was chosen for this situation. The continuous-time transfer function of the filter is:

$$F(s) = \frac{\omega_c}{s + \omega_c}$$

Where ω_c is the cut-off frequency of the filter, expressed in rad/s. In this case, there is a discrete signal, so the discrete filter was used. The discretize transfer function is:

$$F(z) = \frac{Y(z)}{X(z)} = \frac{1 - e^{-\omega_c T}}{1 - e^{-\omega_c T} z^{-1}}$$

where T is the sampling time of the system, so 0.01 seconds. The transfer function can be written in the following way:

$$Y(z)(z^{-1} - e^{-\omega_c T}) = X(z)(1 - e^{-\omega_c T}) \quad \rightarrow \quad y[k] - y[k-1]e^{-\omega_c T} = x[k](1 - e^{-\omega_c T})$$
$$\rightarrow \quad y[k] = y[k-1]e^{-\omega_c T} + x[k](1 - e^{-\omega_c T})$$

where Y(z) is the discrete output of the filter and X(z) the input. This equation is the same function implemented at line 5, with $a = e^{-\omega_c T}$.

The chosen value of ω_c is 10 rad/s. It allows obtaining good filtering of the acceleration data, as visible in Figure 3-1. It displays acceleration of an oscillatory movement of the arm in the XY-plan. The initial data are represented in green, and the noise effect is evident on them. Filtered data are in red, and, as shown, they follow the line of the raw data.

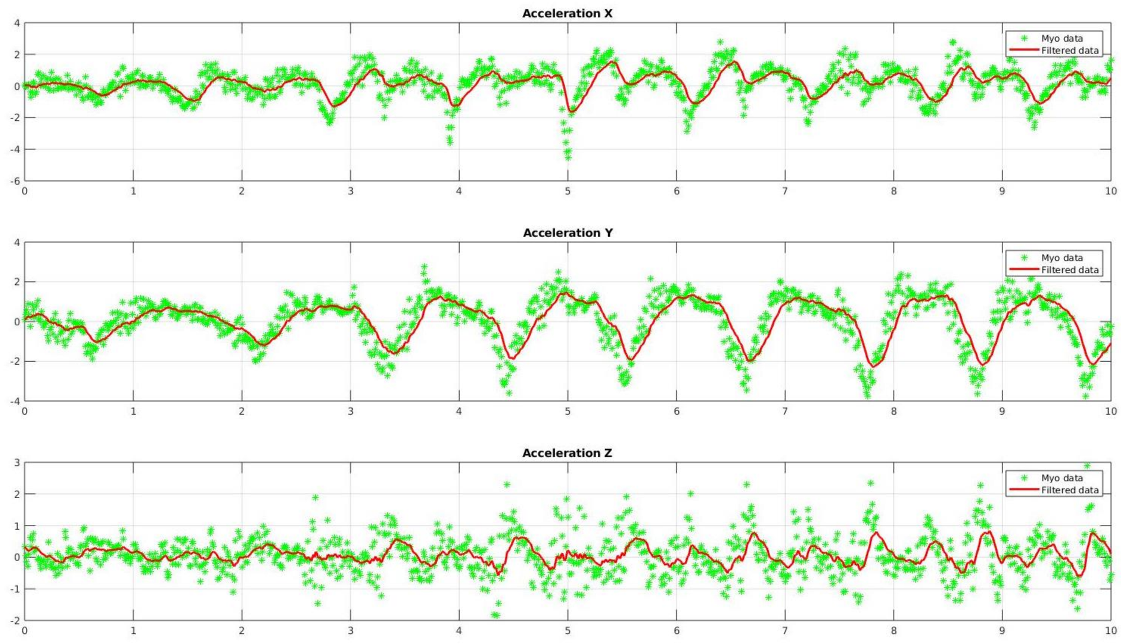


Figure 3-1: Comparison between detected data (in green) and filtered data (in red)

3.1.2 Dead-band

The dead-band corresponds to the code section from lines 6 to 12. There are two relevant issues: the choice of the *noise* value and the expression of the condition of the *if* statement.

To choose the value of *noise*, some tests over the Myo armband have been done. Figure 3-2 displays the acceleration detected by the sensor while resting steadily on a flat surface. Adopting a value of $0.05\ m/s^2$ most of the noise is eliminable.

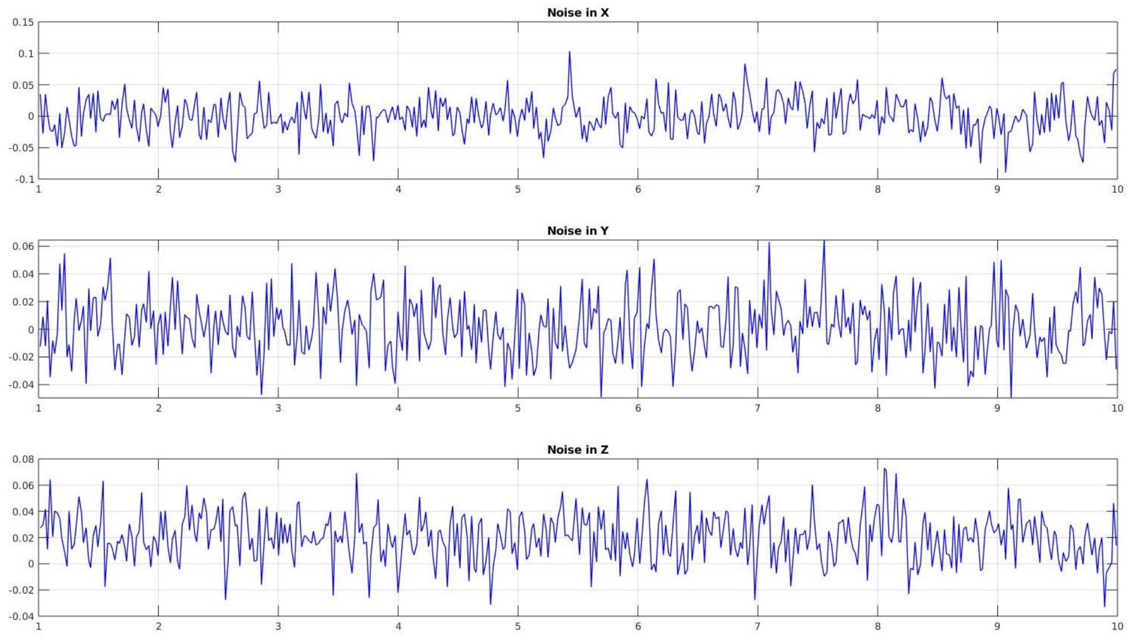


Figure 3-2: Myo armband accelerometer noise

However, the armband is a wearable device, and the arm always has some micro-movement while standing still. So the previous value is not enough for the dead-band purpose, which is to cut-off the micromovements effect.

Other tests were done with the arm still as much as possible. Figure 3-3 shows the acceleration detected in one of these tests, and the other ones gave similar results. Consequently, the chosen value of *noise* is $0.2\ m/s^2$.

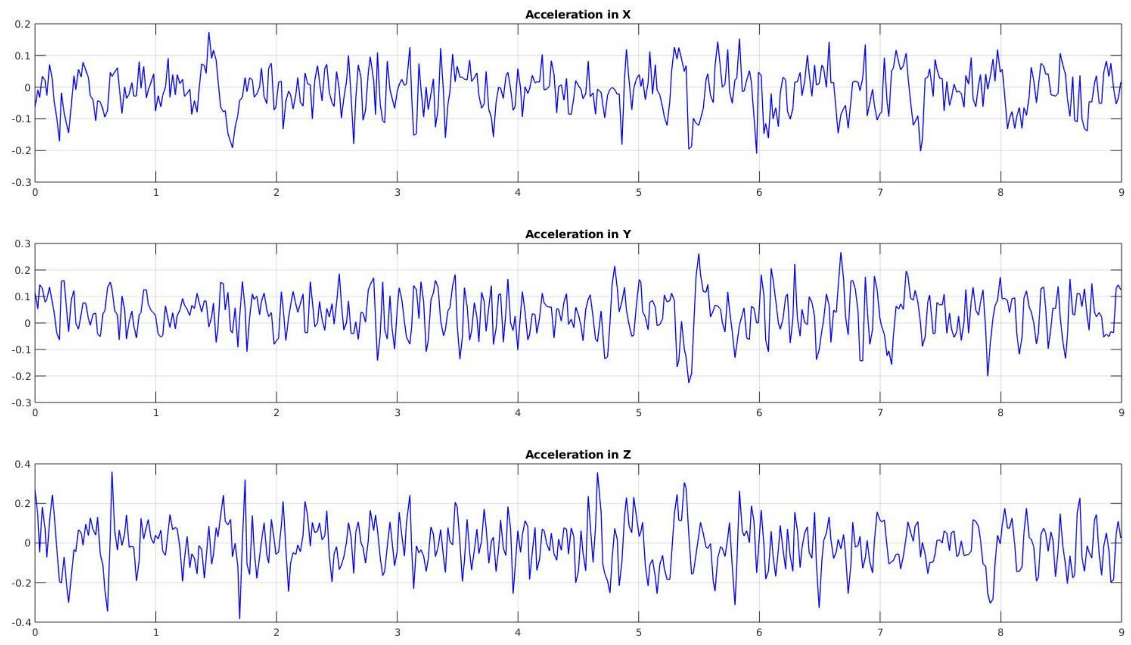


Figure 3-3: Acceleration while the arm stands still

The dead-band is designed to be applied to the raw acceleration data, but this causes an abnormal signal behavior when it crosses the zero value and approaches the dead-band. Indeed, the signal continuously goes in and out from the dead-band as a consequence of the noise. For that reason, the dead-band is also applied to the filtered acceleration. Both raw and filtered data have to be in the dead-band for applying it (line 8). The following graph shows the explained phenomenon and compares the two ways to apply the dead-band. The red line is the signal with the condition on raw data, the blue one with the condition on both signals.

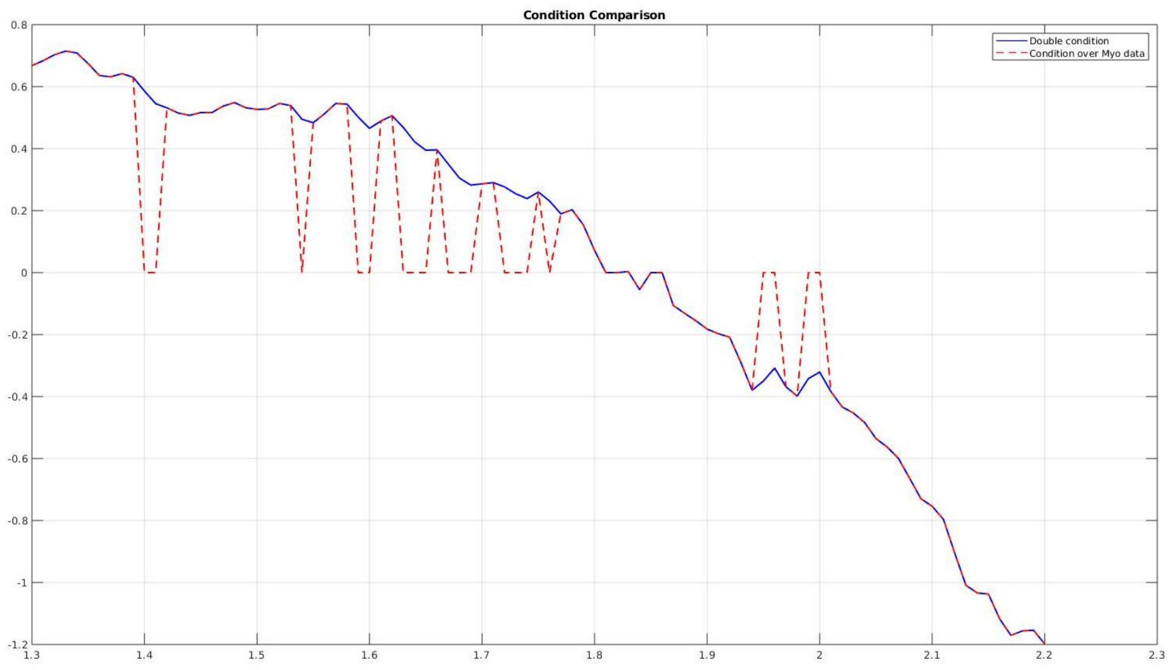


Figure 3-4: Comparison between the two conditions expressions of the if *statement*

3.1.3 From acceleration to velocity: integration and safety limits

Once the acceleration signal has been processed, it can be integrated to obtain the cartesian velocity.

Geometrically, the integral of a signal corresponds to the area subtended to it. The numerical integration of a discrete signal can be done using the trapeze formula. [18]

At each sample, it is calculated as the trapeze area that has as a base the sample time and as heights the actual and the previous value of the signal. The integral is the sum of the sample area to the areas of the preceding samples.

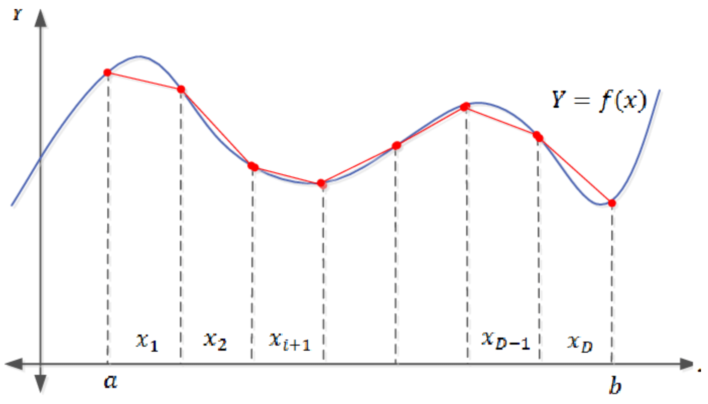


Figure 3-5: Trapezoidal rule for integration

$$Integral = Area(1) + Area(2) + \cdots + Area(n)$$
$$Area(k) = \frac{(signal(k) + signal(k-1))}{2} \cdot sample\ time$$

The code implementation is at line 15. In the code, there is a variable called *coeff*. It is used as a scale factor for the resulting velocity. To avoid losing control of the robot during testing, the velocity is reduced with this variable. Its value has changed depending on the tests done, for this was set as a parameter.

The last section of the loop (lines from 16 to 22) corresponds to the implementation of the velocity safety limits. The Cartesian velocity cannot exceed a specific value, so the corresponding signal saturates when it reaches that limit.

An example of the result of this procedure is in Figure 3-6.

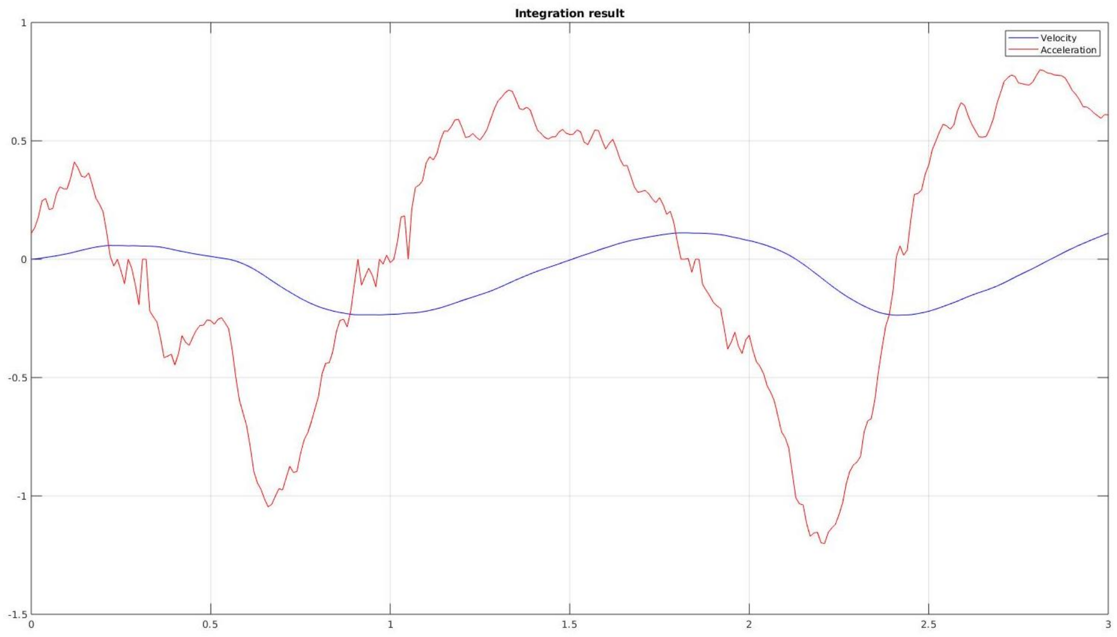


Figure 3-6: Processed acceleration and correspondent velocity

3.2 Teaching by Manual Guidance

To implement the second algorithm, a new executable file with a second node was created. In the second node, there is the implemented code for the manual guidance. The two nodes have to communicate with each other. While the manual guidance is active, the cartesian velocity control cannot be call up, even if a fist is detected.

A topic was created for the communication between the two nodes: */moveit_planning/active*. The cartesian velocity node is subscripted to this topic, and the other node publishes data on it. A boolean message is sent through the topic and tells if the manual guidance is on.

The two control strategies need two different configurations turned on to control the robot. For this reason, on both nodes, the *StartConfiguration* service is prepared to send the activation request of a specific configuration to the Configuration Manager.

3.2.1 Cartesian Velocity Control Node

The node remains mostly the same as the previous version. The communication with the new node and the configuration change request is added.

From lines 4 to 6, it is shown how to set the configuration message used to send the activation request. The message contains a String with the name of the desired configuration. This node communicates with the *cart_teleop* configuration.

```
 1  int main(int argc, char **argv)
 2  {
 3    ...
 4    configuration_msgs::StartConfiguration srv_start;
 5    srv_start.request.start_configuration="cart_teleop";
 6    srv_start.request.strictness=1;
 7    ...
 8    bool first_cicle=true;
 9    while (ros::ok())
10    {
11      ...
12      std_msgs::Bool msgs_bool=bool_sub.getData();
13      bool activation=msgs_bool.data;
14
15      if (gest.pose==ros_myo::MyoPose::FIST && !activation)
16      {
17       if (first_cicle)
18       {
19         configuration_client.call(srv_start);
20         ros::Duration(2).sleep();
21       }
22       first_cicle=false;
23       ...
24      }
25      else {...}
26      if (!disabled) {...}
27       ...
28    }
29    return 0;
30  }
```

At each loop cycle, the node controls if the new node has turned on the manual guidance (lines 12 and 13) and starts to store acceleration data only if it is not enabled (line 15).

A new boolean variable is introduced: *first_cicle*. It indicates if it is the first instant in which a fist is detected. If this variable is true, the request to enable the configuration is sent (line 19). The activation requires a few moments, for this a pause is introduced. Then the variable is set to false. At each cycle in which the cartesian velocity is not enabled, *first_cicle* is fixed to true.

3.2.2 Manual Guidance Node

Even the manual guidance node is a subscriber of the */myo_raw/myo_gest* topic. It has to communicate with MoveIt; therefore it is a publisher in the */moveit_planning/active* topic. In the code below, it is possible to notice some important initialization related to MoveIt (lines from 4 to 20). That passage is essential to pass MoveIt the correct information over robot joint state. The Move Group Interface provides many functionalities to work with robots, for example, set joints positions and targets or save the robot configuration. A move-group related to the used robot needs to be set to work with that interface.

```
 1  int main(int argc, char **argv)
 2  {
 3    ...
 4    std::string group_name;
 5    if (!nh.getParam("group_name",group_name))
 6    {
 7      ROS_INFO("Group name is not defined, using 'manipulator' as default value");
 8      group_name="manipulator";
 9    }
10
11    moveit::planning_interface::MoveGroupInterface group(group_name);
12    group.setStartState(*group.getCurrentState());
13    if (!group.startStateMonitor(10))
14    {
15      ROS_ERROR("Unable to get the current state of the move group
            %s",group_name.c_str());
16      return 0;
17    }
18    moveit::core::RobotState trj_state = *group.getCurrentState();
19    std::vector<double> current_joint_configuration;
20    trj_state.copyJointGroupPositions(group_name,current_joint_configuration);
21
22    ...
23     while (ros::ok())
24     {
25      ros::spinOnce();
26      ros_myo::MyoPose gest=myo_pose_sub.getData();
27
28      if (active) {...}
29      else {...}
30
31      msgs.data=active;
32      activation_pub.publish(msgs);
33      gest_prec=gest.pose;
34     }
35    return 0;
36  }
```

If the simulator is used, the present node communicates with the *trj_tracker* robot configuration. When the ROS node interacts directly with the robot, the required configuration is *manual_guidance*.

There is a boolean variable that indicates when the manual guidance is enabled. Its value is published in the */moveit_planning/active* topic. This variable becomes true when the wave-out hand position is maintained continuously for five seconds. When the hand pose is different, the time variable *t0* is updated to the actual ROS time (line 18). When the gesture ID four is identified *t0* stops to be updated. When the difference between the actual ROS time and *t0* is greater than five seconds, manual guidance is enabled. If the gesture change before five seconds, *t0* is again set to the actual ROS time.

During the activation, the request to change the robot configuration is sent, and *t0* is newly updated.

```
if (active) {...}
else
{
    if (gest.pose==4)
    {
        if((ros::Time::now()-t0).toSec()>5)
         {
            active=true;
            t0=ros::Time::now();
            srv.request.start_configuration="trj_tracker";
            configuration_client.call(srv);
            ros::Duration(2).sleep();
         }
    }
    else
    {
        t0=ros::Time::now();
    }
}
```

Once *active* is equal to true, there are two possibilities: deactivate the modality or save the robot pose.

The code below shows the second option. If the correlate command gesture is done, the current state of the robot move-group is stored into a vector called *waypoints*.

The sample time is very little, and also, a quick hand movement takes multiples cycles to be done. To avoid that the same robot position is stored multiples times while the gesture ID is equal to three, it is also considered the value of the gesture ID at the previous cycle. The pose is memorized if the actual gesture is a wave-in and the previous hand pose is different from it (line 3).

```
 1    if (active)
 2    {
 3        if (gest.pose==3 && gest_prec!=3)
 4        {
 5          moveit::core::RobotState trj_state = *group.getCurrentState();
 6          std::vector<double> current_joint_configuration;
 7          trj_state.copyJointGroupPositions(group_name,current_joint_configuration);
 8           waypoints.push_back(current_joint_configuration);
 9           lenght=waypoints.size();
10        }
11        ...
12    }
13    else {...}
```

The manual guidance deactivation condition is the same used to turn it on. As shown in the following code, the same logic and code structure are used to exit from this modality.

The execution of the memorized path before the deactivation is what changes. Each configuration saved in the vector is set as a joint target of the robot move-group Then, MoveIt plans the trajectory from the robot's actual position to the target one.

If MoveIt does not find a trajectory to reach the target, an error message is printed, and the node is terminated. In the other case, the path is executed. The process is repeated for each position saved in the *waypoints* vector.

The following code shows the implementation of what has just been explained.

```
    if (active)
    {
        ...
        if (gest.pose==4)
        {
            if((ros::Time::now()-t0).toSec()>5)
             {
                srv.request.start_configuration="trj_tracker";
                configuration_client.call(srv);
                ros::Duration(2).sleep();
                active=false;
                t0=ros::Time::now();
                for (unsigned int iw=0;iw<waypoints.size();iw++)
                {
                  ROS_INFO("go to waypoint %u",iw);
                  group.setStartState(*group.getCurrentState());
                  group.setJointValueTarget(waypoints.at(iw));
                  moveit::planning_interface::MoveGroupInterface::Plan plan;
                  bool success = (group.plan(plan) ==
                      moveit::planning_interface::MoveItErrorCode::SUCCESS);
                  if (!success)
                  {
                    ROS_ERROR("Planning failed computing waypoint %u", iw);
                    return 0;
                  }
                  group.execute(plan);
                }
             }
        }
        else
        {
            t0=ros::Time::now();
        }
    }
    else {...}
```

3.3 Path Repetition – State Machine

In the previous chapter, there is an explanation of how the idea for the control strategy evolved. The third algorithm has been developed and implemented as a state machine.

In the present case, the program is inside a single node, so there is only one executable file. The */moveit_planning/active* topic is no longer needed. The new node has the same subscribers and publishers of both the nodes of the previous version of the program, adding a publisher for the */myo_raw/vibrate* topic. In the code, haptic feedback is introduced by using the vibration motor of the Myo armband. The node sends the length of the vibration through the topic.

Inside the program, the robot configuration has to switch from *cart_teleop* to *trj_tracker* or *manual_guidance* at different times. At every exchange, the same code is repeated, so it is redundant. A function that changes the robot configuration has been implemented before the main. It receives a string with the configuration name as an input.

```
void change_config(std::string new_config, ros::ServiceClient
    configuration_client_channel)
{
    configuration_msgs::StartConfiguration srv;
    srv.request.start_configuration=new_config;
    srv.request.strictness=1;
    configuration_client_channel.call(srv);
    ros::Duration(2).sleep();
}
```

With the same cases, two switch statements are used to realizes the state machine: the first is for the implementation of the states, the second one of the transitions. States and transitions have been implemented in two separate *switch* statements to make the code clearer and more easily editable.

The enumeration type *State* is defined and a variable of the same type is used as the condition for both the *switch* statements. The starting value of the variable is None, then its value change in the transitions *switch*.

```
enum State {NONE, TEACH, EXECUTION, VEL_CONTROL};
```

The following code is an example of the structure assumed by the code inside the loop.

```
while (ros::ok())
{
    ros::spinOnce();
    lp.sleep();
    ros_myo::MyoPose gest=myo_pose_sub.getData();

    switch (state) {
    case imu_teleop::State::TEACH:{ ...
    } break;

    case imu_teleop::State::EXECUTION:{ ...
    } break;

    case imu_teleop::State::VEL_CONTROL:{ ...
    } break;

    case imu_teleop::State::NONE:{ ...
    } break;

    default: {
    } break;
    }

    switch (state) {...}

    gest_prec=gest.pose;
}
```

3.3.1 States

As told in the previous chapter, the states of the machine are parts of the old algorithm. For the code, it is a similar thing: sections of the old code are taken and reused to implements the states and the transitions.

If the state None has no commands, the system waits for a transition to another state. The other states are:

- *Teach*: if the gesture ID is equal to three and the previous gesture is different from it, the robot configuration is saved in the *waypoints* vector. At least the state does nothing. A short vibration is added at the end of the *if* statement that stores the robot position. It confirms that the storage takes place. The code lines below are the publication of the length of a short vibration.

```
msgs_vibr.data=1;
vibration_pub.publish(msgs_vibr);
```

- *Execution*: this state corresponds to the *for* loop before the manual guidance deactivation. It executes the memorized path. When the loop ends, a variable called *active* is set to false. It is used to indicate the conclusion of the path execution.
- *Vel_Control*: it is equal to what the cartesian velocity control does when a fist is detected. However, another strategy has been chosen. Inside the package library, a class called *ImuConfig* has been created. It has the acceleration and velocity vectors as attributes, and its methods are the filter, the dead-band, the integration, and the saturation for safety limits. There is also a method

that set to zero all the vectors and one that does all the procedure. All the non-vector variables used for the calculations are considered method parameters. This allows the reuse of the code. In the main, there is *imuData*, which is an *ImuConfig* object.

```
imuData.velocityConfiguration(a,noise,T_b_g,vel_max,coeff,st);
```

In this state, there is only the storage of the acceleration in the corresponding attribute, the call of the function with all the procedures, and the publishing of the cartesian velocity.

3.3.2 Transitions

Transitions are code sections in which the value of the variable *state* changes. Also in this case, parts of the codes seen in the previous sections are reused. The program controls the actual state and the possible transitions in the loop. If the condition for a transition occurs, the *state* value is changed.

- *From None*: from this state, all the other states are reachable. The passage to the state *Vel_Control* occurs if the gesture ID is equal to two. When this happens, there is a vector reset, a vibration is activated, and then the robot configuration request is sent. In this way, there is a pause between the first storing and the vibration. This because the haptic feedback can affect the acceleration measurement.
  Figure 3-7 shows how the vibration affects the accelerometer's measurement. When the arm is still, the signal oscillates between $5\ m/s^2$ and $-5\ m/s^2$. In this case, acceleration is an order of magnitude larger than the dead-band value.
  If the control system takes the acceleration while the armband's vibration, velocity saturates instantly. Then its value switches from the maximum to the minimum values and vice-versa at each loop cycle.
  Each time the haptic feedback is introduced before or after the acceleration request of the control system, a pause is introduced. That prevents the described phenomenon. This can be done by calling the *change_config* function, which includes a pause.

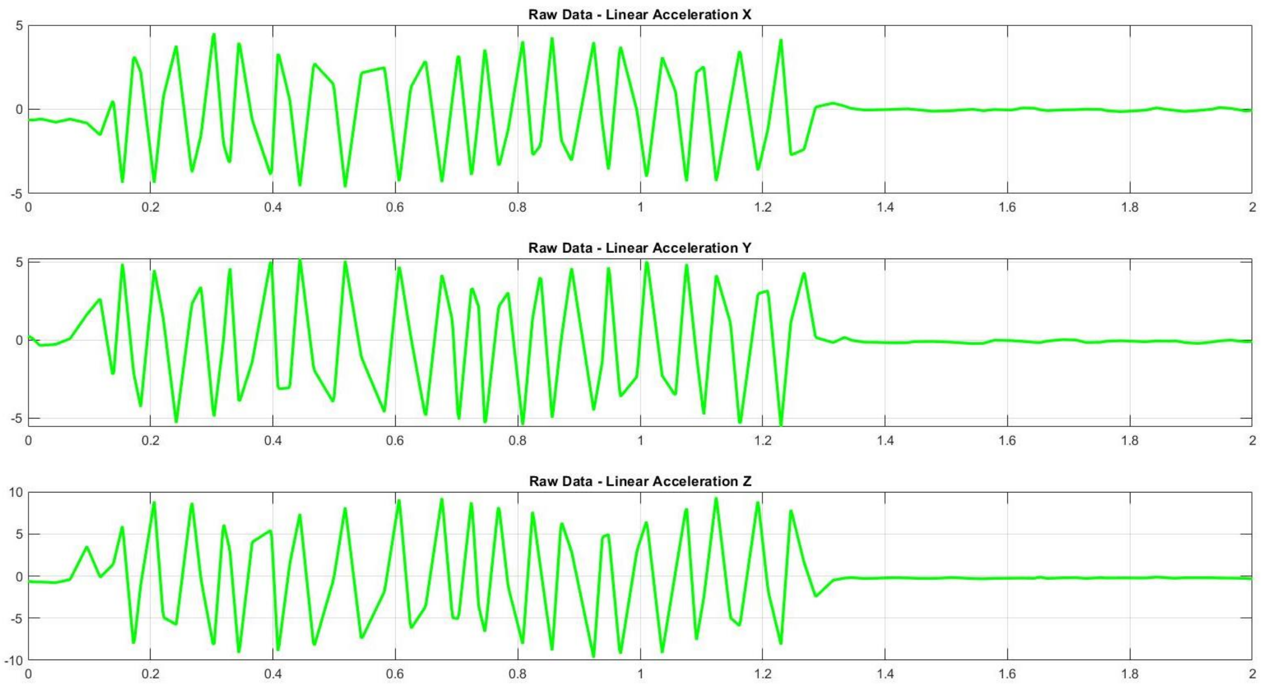


Figure 3-7: Vibration effect over the acceleration measurement

The passage to *Teach* uses the same logic of the manual guidance activation. A vibration advises of the manual guidance activation. Only a transition to *Execute* is possible from this state. The boolean variable *active* assumes a true value during the transition.
The direct passage to *Execute* is the novelty in the code. The transition takes place if the finger spread gesture (ID number: five) is performed. *active* is set to true, and the haptic feedback warns of the path execution beginning.

```
case imu_teleop::State::NONE: {
    ...

    if (gest.pose==5)
    {
        state=imu_teleop::State::EXECUTION;
        active=true;

        ROS_INFO_THROTTLE(1,"active EXECUTION");

        msgs_vibr.data=3;
        vibration_pub.publish(msgs_vibr);
    }
    ...
} break;
```

- *From Teach*: there is only a transition to *Execution*. The same condition for the transition from *None* to this state is used.
- *From Execution*: from this state is possible only to return at *None*. It is possible to notice why the variable *active* has been introduced in the code.

```
case imu_teleop::State::EXECUTION: {
    if (!active)
    {
      msgs_vibr.data=3;
      vibration_pub.publish(msgs_vibr);

      state=imu_teleop::State::NONE;
      ROS_INFO_THROTTLE(1,"deactive EXECUTION");
    }
} break
```

The variable is true from before the state activation, and it is set to false only when the execution ends. A long armband vibration informs about this event.

3.4 DIRECTION CONTROL STATE

The last amendments to the code are presented in this part. The last implementation includes the arm movement direction and uses it to control the robot. Subscribers and pushers are the same as the preceding implementations. Every state requires its robot configuration and sends the change request during the transitions. The amendment consists of the adding of two cases inside the *switch* statements.

3.4.1 States

Two new states are added to the program: *Direction_Control* and *Execute_Direction*.

The following code shows the first state, which calculates and memorizes the versor of a movement. It starts with storing raw acceleration data into *imuData* attributes (lines from 6 to 8). While testing, a variable offset on the z-axis acceleration has been found. A function that calculates the offset has been created. It is deepened in the last section of this chapter.

After that, there is the call of the function with all the procedures done for *Vel_Control*. The velocity integration and the versor calculation are added in this function (line 10). The last step is set to true a boolean a variable called *direction_taken*. It signals if a direction is taken (line 12).

```
 1        case imu_teleop::State::DIRECTION_CONTROL:{
 2          if(gest.pose==2)
 3          {
 4            geometry_msgs::TwistStamped imu=imu_sub.getData();
 5            double offsetZ=findMyoError(listener);
 6            imuData.acc_in_g_(0)=imu.twist.linear.x;
 7            imuData.acc_in_g_(1)=imu.twist.linear.y;
 8            imuData.acc_in_g_(2)=imu.twist.linear.z-offsetZ;
 9
10            imuData.velocityConfiguration(a,noise,T_b_g,10000.0,coeff,st);
11
12            direction_taken=true;
13          }
14        } break;
```

Two new methods (shown in the code below) are implemented in the *ImuConfig* class: the velocity integration (lines from 1 to 4) and the versor calculation (lines from 6 to 9). The trapeze formula is used again for the integration. However, there is no scale factor.

```
1  inline void ImuConfig::integralPosition(double st)
2  {
3     position_=position_+(st*0.5*(vel_in_b_+vel_in_b_old_));
4  }
5
6  inline void ImuConfig::generateVersor()
7  {
8    versor_=position_.normalized();
9  }
```

The *Execute_Direction* state is more complex, as observable in the following code. It communicates with MoveIt and uses the robot move-group as *Execution*. The actual robot joints configuration is taken, and the gripper Cartesian position is obtained (lines from 4 to 10). The gripper target position is determined as a translation of ten centimeters in the versor direction of its current pose (lines from 12 to 14).

The joints' target position is obtained from the gripper target position by the inverse kinematics calculation (lines from 17 to 25). Then the target robot configuration is sent to MoveIt. It planners the robot's movement to that position. In the end, the execution command is sent to MoveIt, and the variable called *execute_end* is set to false. It is used to indicate the path execution conclusion, as *active* does in the Execution state.

```
case imu_teleop::State::EXECUTE_DIRECTION:{
  versor=imuData.getVersor();

  moveit::core::RobotStatePtr kinematic_state=group.getCurrentState();
  std::vector<double> joint_values;

  kinematic_state->copyJointGroupPositions(joint_model_group, joint_values);

  geometry_msgs::PoseStamped T_w_t=group.getCurrentPose();
  geometry_msgs::Pose gripperPose=T_w_t.pose;

  gripperPose.position.x+=(versor(0)*movement_lenght);
  gripperPose.position.y+=(versor(1)*movement_lenght);
  gripperPose.position.z+=(versor(2)*movement_lenght);

  double timeout = 0.1;
  bool found_ik = kinematic_state->setFromIK(joint_model_group, gripperPose,
      timeout);
  if (found_ik)
  {
    kinematic_state->copyJointGroupPositions(joint_model_group, joint_values);
  }
  else
  {
    ROS_INFO("Did not find IK solution");
  }

  group.setStartState(*group.getCurrentState());
  group.setJointValueTarget(*kinematic_state);

  moveit::planning_interface::MoveGroupInterface::Plan plan;

  bool success = (group.plan(plan) ==
      moveit::planning_interface::MoveItErrorCode::SUCCESS);
  ros::Duration(2).sleep();
  if (!success)
  {
    ROS_ERROR("Planning failed computing waypoint");
    return 0;
  }
  group.execute(plan);
  ros::Duration(2).sleep();

  execute_end=true;
} break;
```

3.4.2 Transitions

Only the transition to *Direction_Control* has been added from *None*. The transition condition structure is the same used to activate the manual guidance, but with the wave-in gesture (ID number: three). The vice-versa has the same condition; it is observable in the code below (lines from 2 to 18). Haptic feedback is introduced, as in the other transitions. In the passage to *Direction_Control*, the robot configuration change request is after the vibration. In this way, the acceleration detected by the state cannot be affected by the armband oscillations because a pause is introduced.

From *Execute_Direction*, only the passage to *Direction_Control* is possible. The same logic used for the transition from Execution to *None* has been implemented for the passage. However, *execute_end* is used instead of *active*. There is the reset of all vectors during this transition.

The transition from *Direction_Control* to *Execute_Direction* occurs when, at the same time, the gesture ID is different from two, and *direction_taken* is set to true (lines from 20 to 28). While the passage between the two states, the variable return to false.

```
    case imu_teleop::State::DIRECTION_CONTROL: {
        if(gest.pose==3)
        {
         if ((ros::Time::now()-t0_direction).toSec()>5)
         {
           msgs_vibr.data=2;
           vibration_pub.publish(msgs_vibr);
          state=imu_teleop::State::NONE;
          ROS_INFO_THROTTLE(1,"deactive DIRECTION CONTROL");

          imuData.vectorReset();
          t0_direction=ros::Time::now();
         }
        }
        else
        {
          t0_direction=ros::Time::now();
        }

        if(gest.pose!=2 && direction_taken)
        {
          state=imu_teleop::State::EXECUTE_DIRECTION;
          msgs_vibr.data=3;
          vibration_pub.publish(msgs_vibr);
          change_config("trj_tracker",configuration_client);
          ROS_INFO_THROTTLE(1,"active EXECUTE DIRECTION");
          direction_taken=false;
        }
    } break;
```

3.4.3 FindMyoError Function

As mentioned above, the Myo armband accelerometer has an offset in the z-axis value. The offset changes with the armband position. This was noticed after some tests.

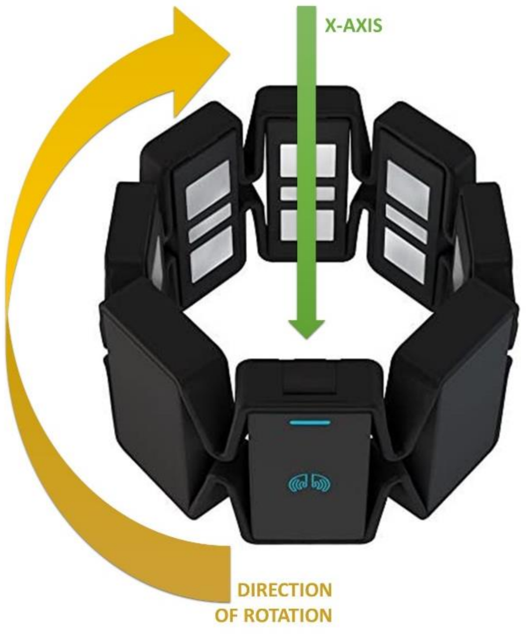


Figure 3-8: Myo rotation used for tests

The offset depends on the angle taken among the Myo internal z-axis and the global z-axis.

Figure 3-8 shows the rotation used to measure the offset. During the tests, the internal x-axis was perpendicular as much as possible to the global z-axis. A rotation between $-3/4\pi$ and $+\pi$ has been considered. The references for the rotation angle were the armband sensors. The following table shows the value of the offset depending on the angle between the internal z-axis and the global z-axis.

Angle	**Test 1**	**Test 2**
$-{}^{3}/_{4}\,\pi$	0.546595	0.414886
$-{}^{1}/_{2}\,\pi$	0.13169	0.14118
$-{}^{1}/_{4}\,\pi$	-0.162079	-0.143188
-0	-0.357639	-0.363893
$+{}^{1}/_{4}\,\pi$	-0.357469	-314031
$+{}^{1}/_{2}\,\pi$	-0.0762921	-0.0612203
$+{}^{3}/_{4}\,\pi$	0.219007	0.288148
$+\pi$	0.502188	0.512429

Table 1: Offset data from tests

The graphical representation of the data matches with a sinusoid. If the angle between the internal z-axis and the global z-axis is called ϑ, the function that finds the value of the offset with the respect of the ϑ value is:

$$Offset = mean - amplitude * \sin(\vartheta + traslation)$$

Where $mean$ is the arithmetic average of the offset data founded during the tests, and $amplitude$ is the difference between the maximum value assumed by the offset and $mean$. To fit the data, the sinusoid needs a little translation.

Figure XY displays the offset of the two tests (red line and blue one) and the offset calculated by the function (green line). The result is considered satisfactory.

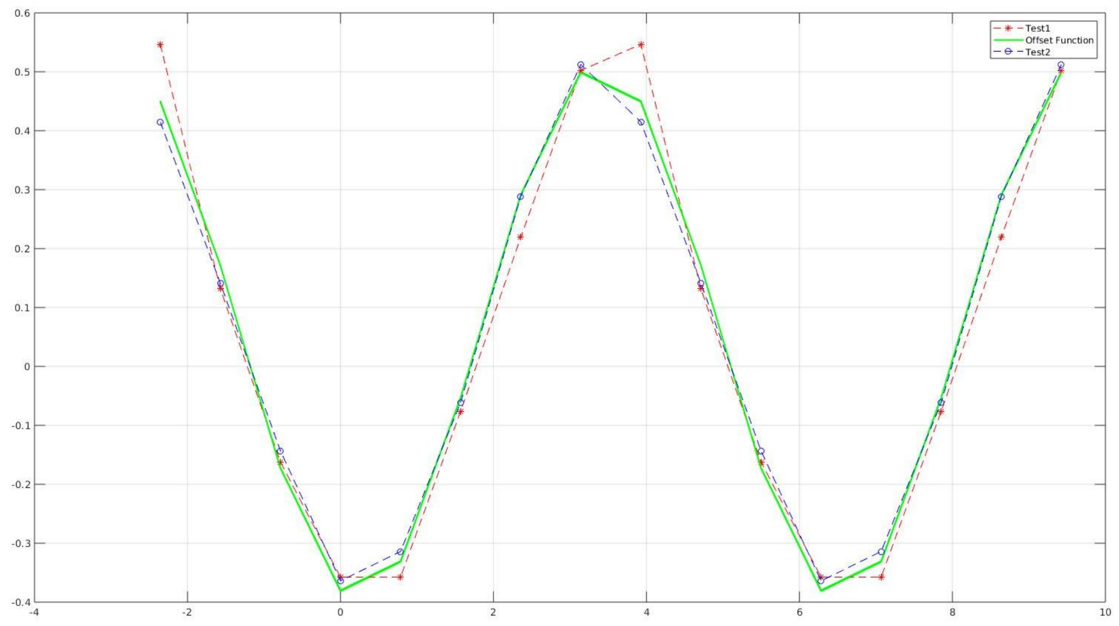


Figure 3-9: Sinusoidal offset pattern

The value of ϑ has to be known to apply the relation between the offset and ϑ. The section from lines 10 to 22 of the code below finds the transform between the global reference system and the Myo inner reference system. The command at line 24 turns the transform into Eigen::Vector3d. Vectors in the global reference system describe the axes of the Myo inner reference system.

The scalar product of two vectors is the cosine of the angle between the two. This allows finding only angles between 0 and π. To determine a complete rotation of 2π, a second reference axis is needed. Considering how data have been taken in this case, if the angle between the internal y-axis and the global z-axis is greater than $\pi/2$, $\vartheta = -\vartheta$, otherwise it does not change (lines from 30 to 36). Those relations are obtained experimentally.

Line 37 is the implementation of the arithmetic relationship among the offset and ϑ.

```
 1  double findMyoError(tf::TransformListener& listener)
 2  {
 3    ros::Duration(0.01).sleep();
 4    double media=0.0593;
 5    double amplitude=0.4531;
 6    tf::StampedTransform transform;
 7    Eigen::Affine3d T_gs;
 8    Eigen::Vector3d z_g(0,0,1);
 9
10    for (int itrial=0;itrial<10;itrial++)
11    {
12        try{
13           listener.lookupTransform("/imu_global", "/myo_raw",
14                                    ros::Time(0), transform);
15           ROS_INFO("ok");
16           break;
17         }
18         catch (tf::TransformException ex){
19           ROS_ERROR("%s",ex.what());
20           ros::Duration(1.0).sleep();
21         }
22    }
23
24     tf::transformTFToEigen(transform,T_gs);
25
26     Eigen::Vector3d Zs=T_gs.linear().col(2);
27     Eigen::Vector3d Ys=T_gs.linear().col(1);
28     double cos_theta=Zs.dot(z_g);
29     double cos_gamma=Ys.dot(z_g);
30     double theta=acos(cos_theta);
31     double gamma=acos(cos_gamma);
32
33     if(gamma>(3.1416/2.0))
34     {
35       theta=-theta;
36     }
37     double offset=media-amplitude*sin(theta+(3.1416*0.5)-0.25);
38     return offset;
39  }
```

4 FINAL RESULTS

As mentioned in the first chapter, the control strategy has been tested on two different collaborative robots: the UR10e and the FourByThree. Both have six degrees of freedom and use only rotary actuators.

The first section of the present chapter is a comparison between the two collaborative robots.

Each developed program has been experimented, first of all, with the simulator on RViz. Tests were carried out on robots when they gave appreciable results on RViz. This chapter presents the experimental results of the three control methodologies implemented in the last program. Each section deals with the results of a specific control method. Different tests were done to check the control system's correct behavior on both robots. The result of one trial for each robot is given as an example.

In the present chapter are introduced the results of the single methodologies on both robots. The automatic control system transitions from a methodology to another work properly. It gave very satisfactory results during the trials.

4.1 UR10E AND FOURBYTHREE: TWO DIFFERENT COBOTS

The UR10e and the FourByThree used for the testing are similar. The main difference between them is the actuators' type. SEAs are described in the first chapter. They differ from the traditional actuators for the presence of a spring coaxial with the couple motor-gearbox. The elasticity makes the FourByThree safer than the UR10e, but, at the same time, more challenging to control and less accurate.

Consider a cartesian displacement of the gripper between two points. The cartesian trajectory is linear. MoveIt calculates the direct kinematic and dynamic of the robot. Joints' position is passed to the robot, which has to track the targets.

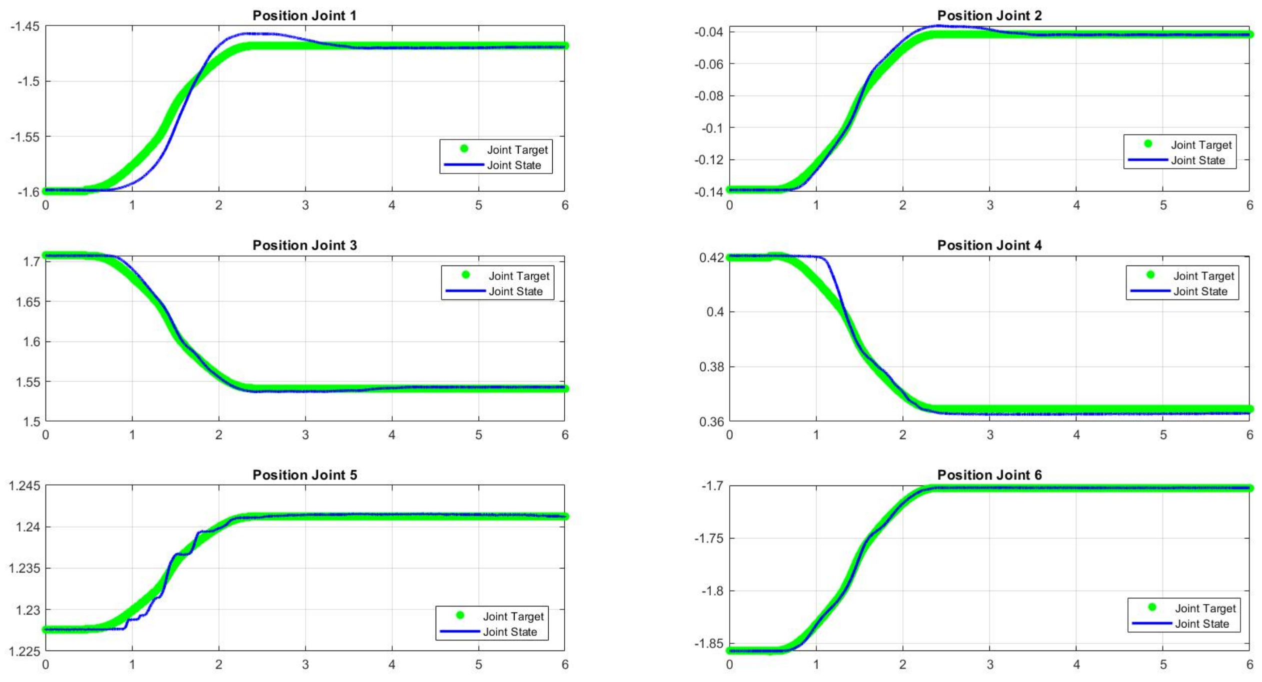


Figure 4-1:FourByThree – Joints' targets tracking

Figure 4-1 displays the FourByThree's joints' position (blue line) compared to the joints' target (green line). The manipulator follows the target no accurately. Overshoot can be observed in the first two joints graphs. The third joint becomes stable after few moments. The other joints oscillate while following the target, but overshoot does not occur.

When the UR10e performs the same kind of movements, the result changes. Figure 4-2 shows the UR10e's joints' position (blue line) compared to the joints' target (green line). Overshoot never occurs. There is a little delay between the target and the actual position. Data transmission requires some moments, therefore causes that delay. The UR10e accuracy decreases when it has to execute a very little joint movement. This fact is observable looking at the graphs of joints 3 and 5.

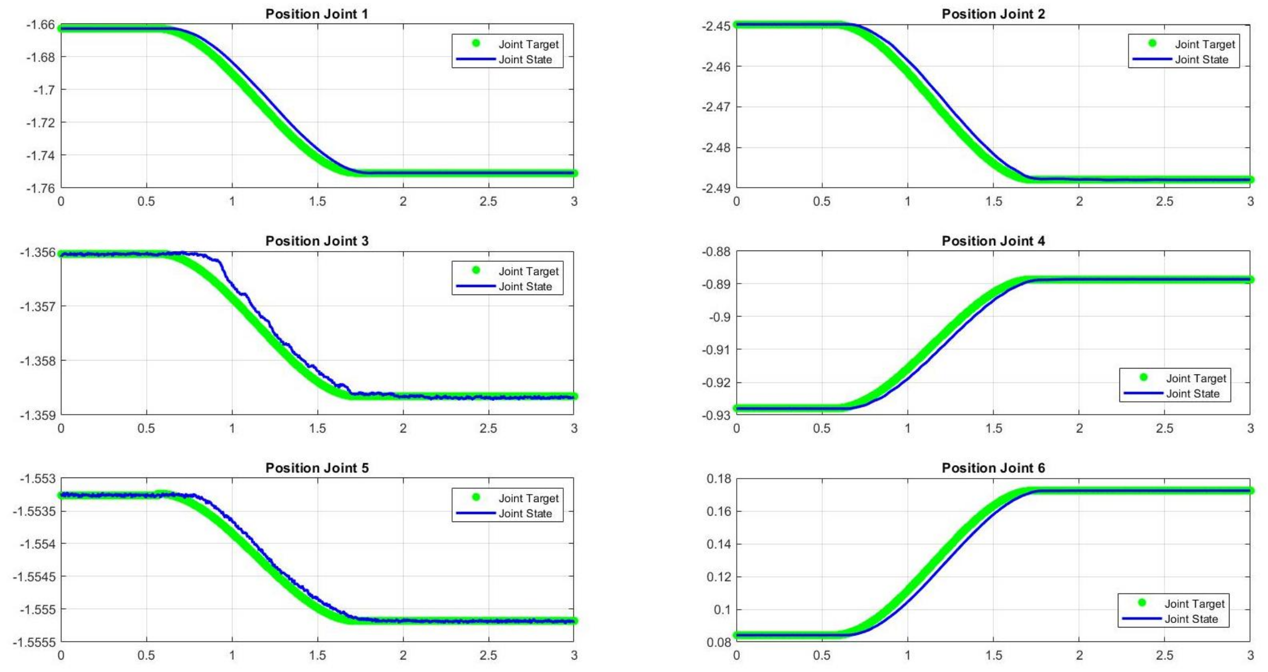


Figure 4-2: UR10e - – Joints' targets tracking

The difference between SEA and standard joint behavior is more evident by observing their velocity.

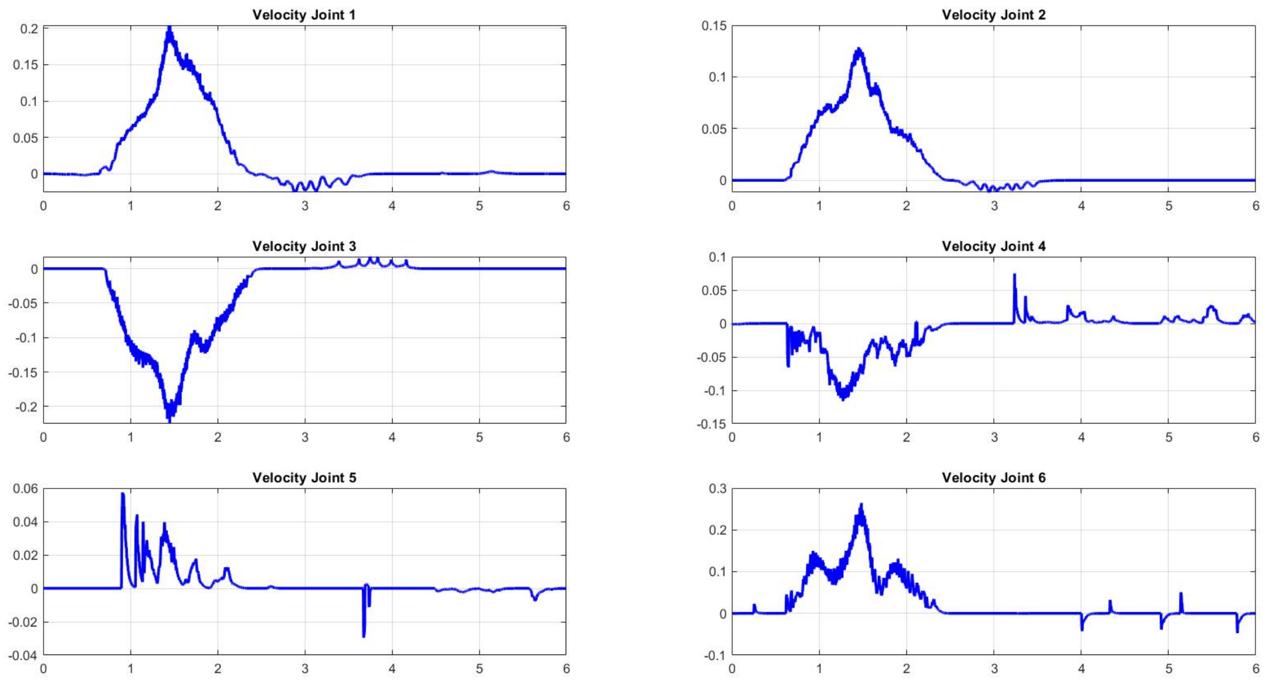


Figure 4-3: FourByThree - Joints Velocity

Figure 4-3 represents the joints velocity of the FourByThree. Joints curb suddenly and cause oscillations. The FourByThree has a specific controller that dampens the effect of the springs. The robot stabilizes its position faster than without the controller. Nevertheless, it needs a very long time to stabilize all the joints.

Figure 4-4 displays the joints velocity of the UR10e. Only the second joint requires a few instants to stabilize itself. Other joints are stables.

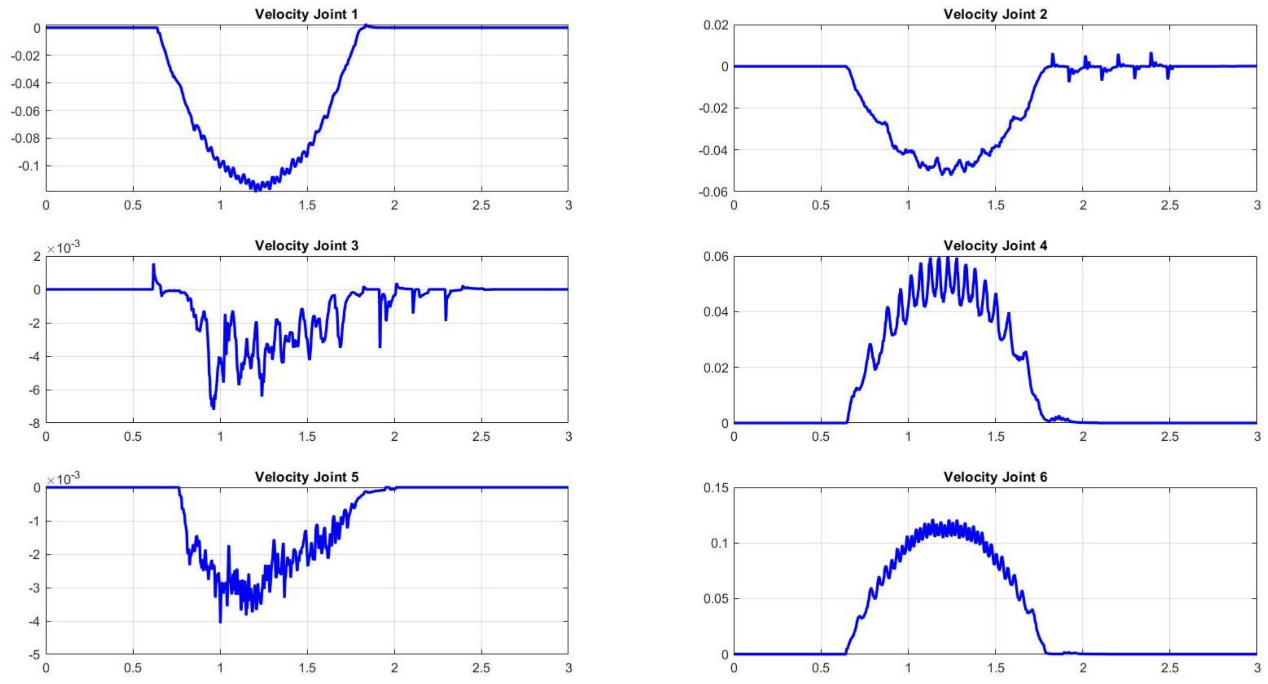


Figure 4-4: UR10e - Joints Velocity

4.2 Cartesian Velocity Control

The main critical issue is armband behavior. The accelerometer inside the Myo armband used does not always give a signal coherent with the arm movement. Myo armband detects correctly wide, smooth, and clear movements. If the arm does little and sudden movements, the accelerometer signal appears very noisy and unprecise. In that case, also the direction detected by the armband can result incorrect.

Even gesture recognition is not always precise. It can be unconstant during the arm motion. Thus, the trajectory may not be wholly acquired.

The program works properly according to the signal received by the armband. The processing of the acceleration data gives satisfactory results. The resulting acceleration is sufficiently clean. The velocity obtained from the integration saturates correctly.

The UR10e is more accurate and fast in the execution than the FourByThree. For both the following tests, the arm movement is a spiral.

4.2.1 UR10e

Figure 4-5 shows the acceleration received by the accelerometer (in green line) and the output obtained by applying the filter and the dead-band (in red line).

Figure 4-6 displays the velocity obtained by the integration (in blue). Those are the data passed to the Configuration Manager. Besides, there is the representation of the Gripper Cartesian velocity (in magenta), obtained as finite-difference of the gripper's Cartesian position. The robot interface has a topic in which publishes Gripper Cartesian position. Velocity is obtainable from those pieces of information.

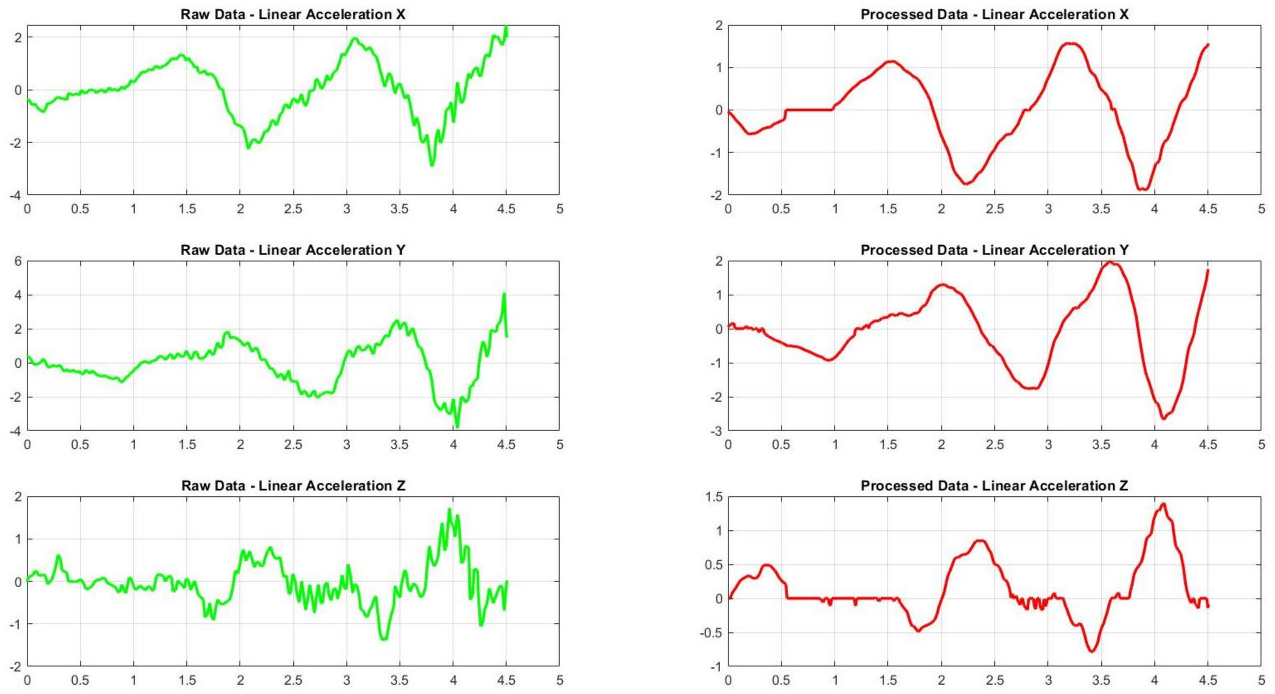


Figure 4-5: UR10e – Cartesian Acceleration

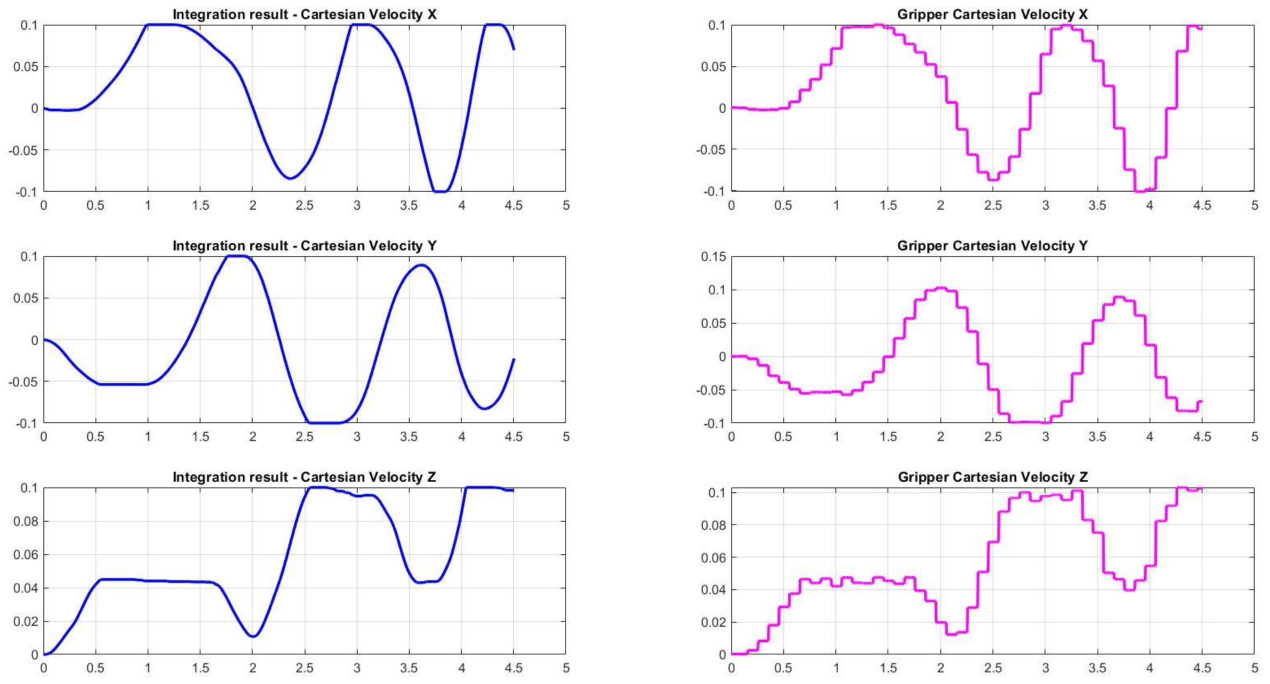


Figure 4-6: UR10e – Cartesian Velocity

Compare the velocity send to the Configuration Manager and the Gripper Cartesian velocity. The two signals have the same trend, assuming approximately the same values, showing the functionality's effectiveness. As expected, the gripper velocity is lagging behind the target one.

It is possible to observe the performance of the joints during the robot movement. Joints receive a position target. They precisely track the target (Figure 4-7). The delay is due to to the data transmission time and the position control reaction time.

Figure 4-8 shows the joints' angular velocity. The joints' effort is exhibited in Figure 4-9.

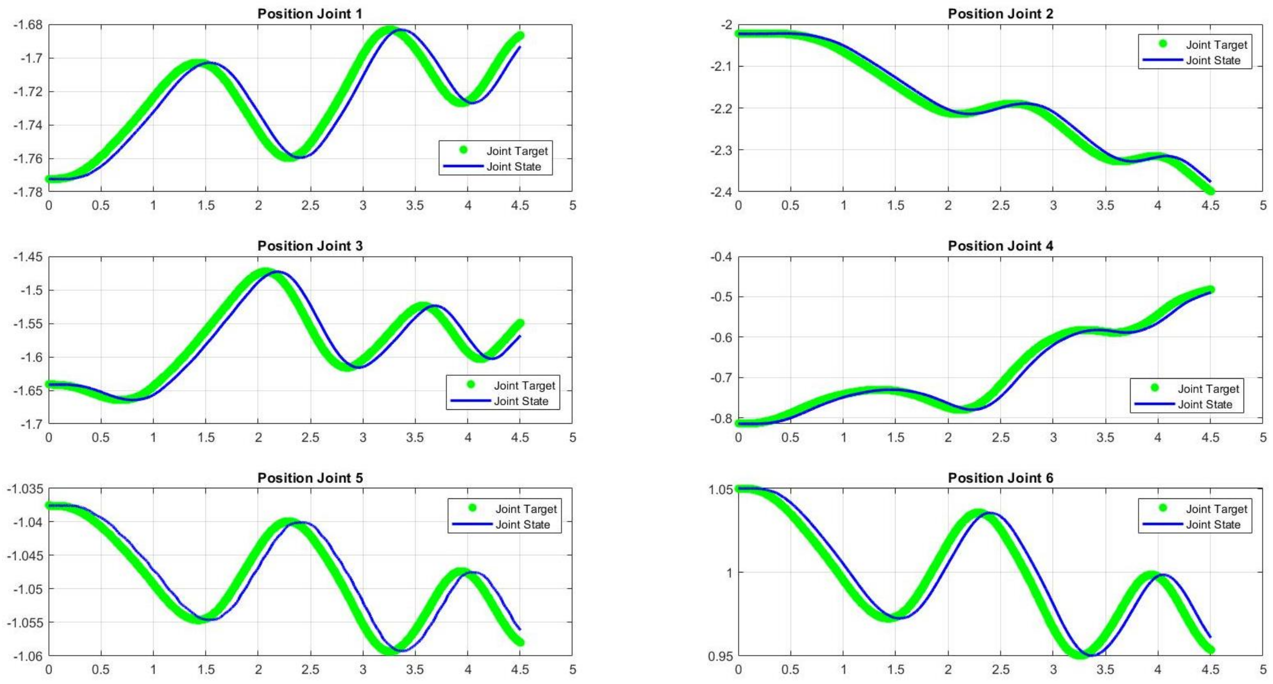


Figure 4-7: UR10e – Joint Position

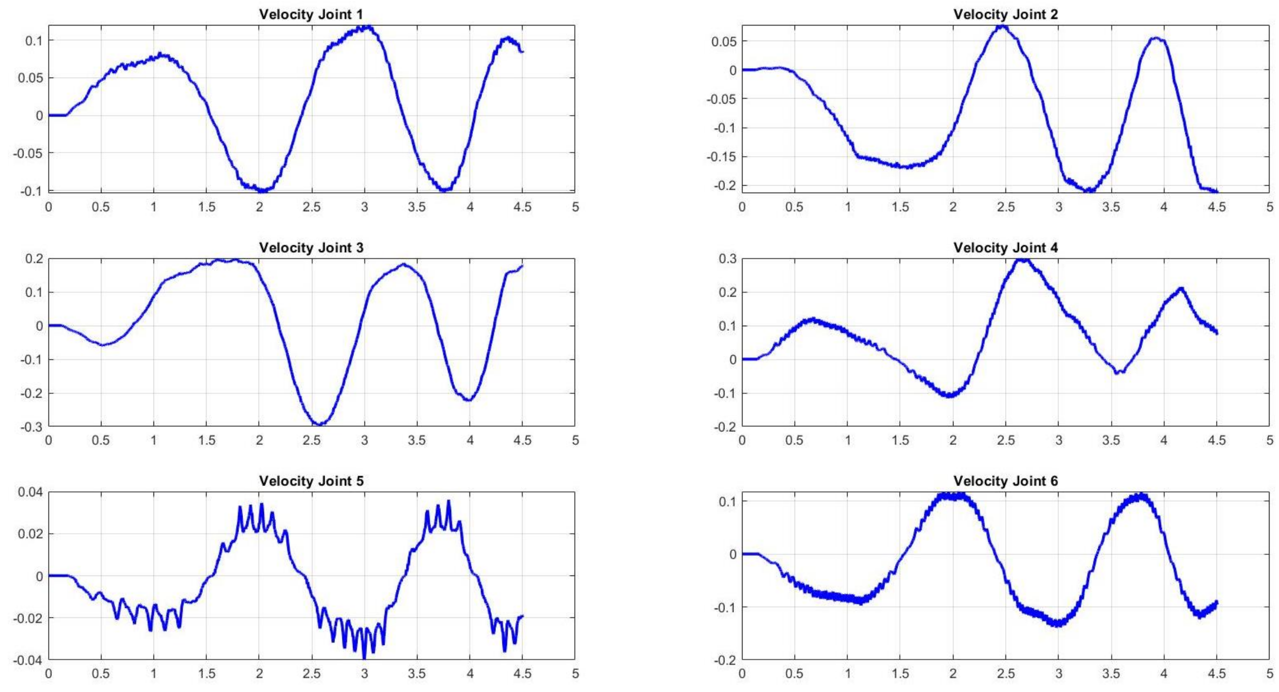


Figure 4-8: UR10e – Joint Velocity

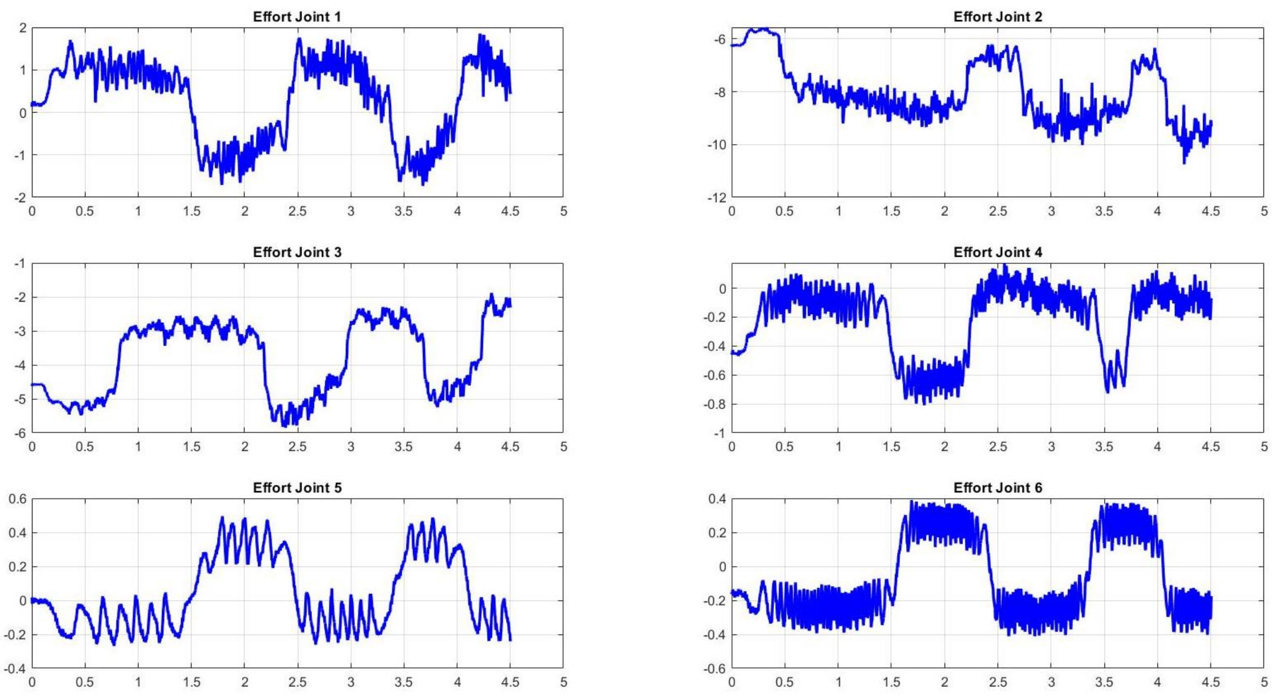


Figure 4-9: UR10e – Joint Effort

4.2.2 FourByThree

Acceleration data (Figure 4-10) are processed correctly as in the previous case.

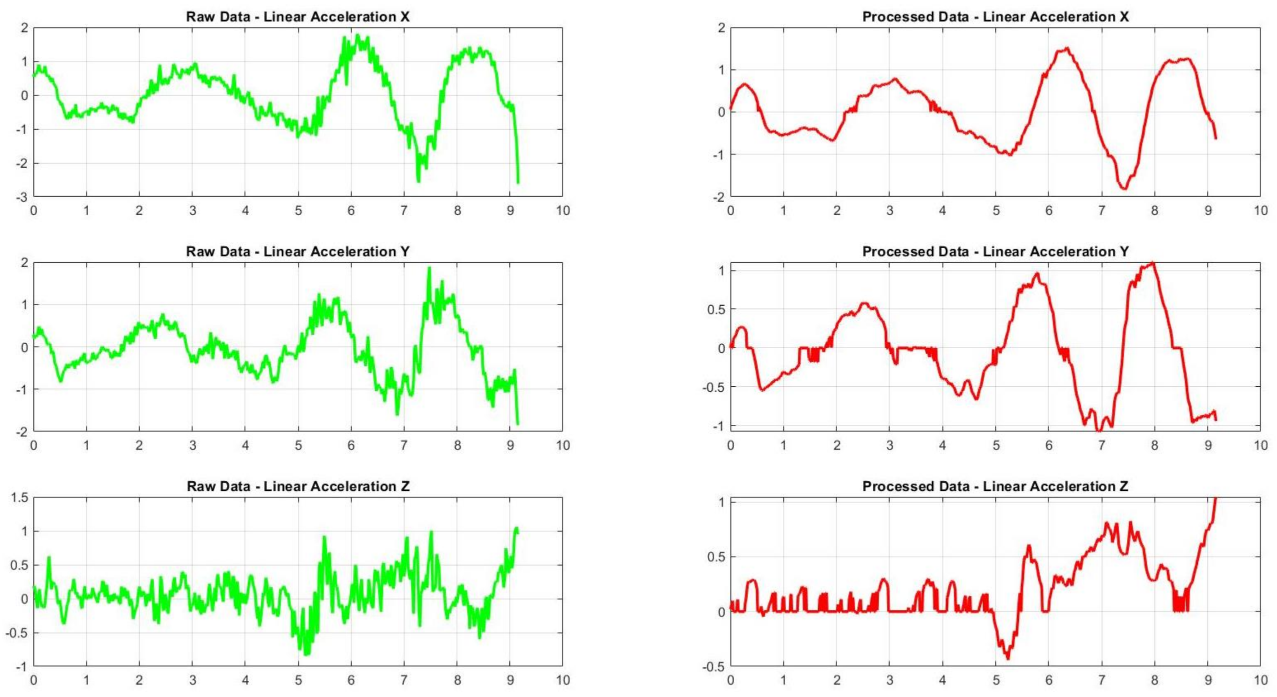


Figure 4-10: FourByThree – Cartesian Acceleration

The main difference is in the cartesian velocity graph Figure 4-11. The gripper Cartesian velocity has a trend similar to the target velocity but different values on the x and y axes. It does not follow the trend on the z-axis. There is a delay of almost one second from the information sent to the Configuration Manager and corresponding gripper velocity, due to the limited bandwidth of the position control loop.

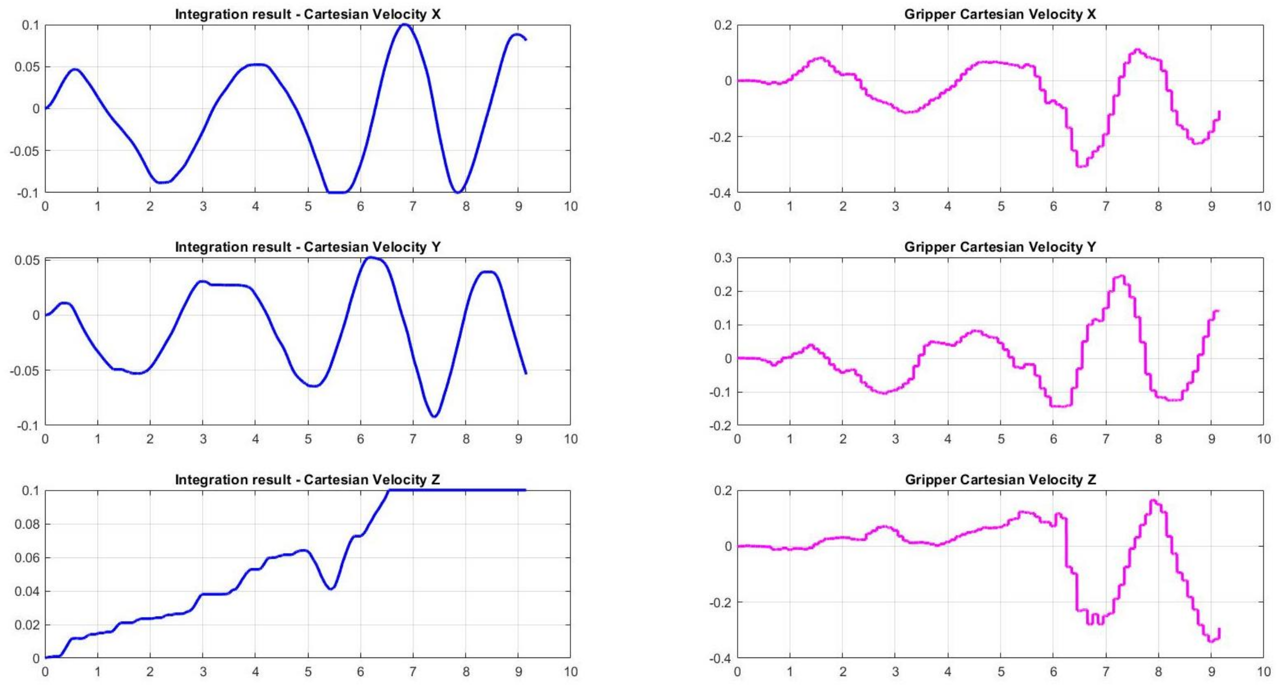


Figure 4-11: FourByThree – Cartesian Velocity

The delay is more evident observing the joints' position compared with their target (Figure 4-12). The oscillation amplitude of the joints' position is greater than the one of the target. Joints 3 and 5 reach their limit, affecting robot performance. A more reactive position control could improve the result.

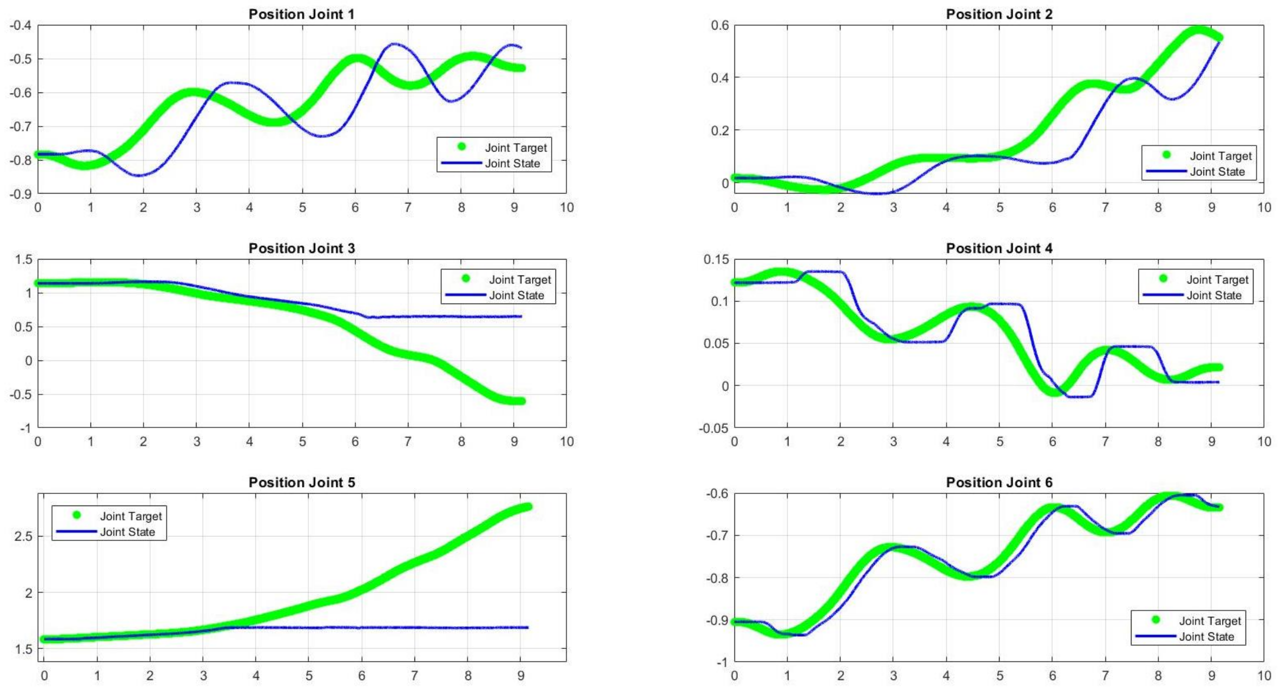


Figure 4-12: FourByThree –Joint Position

Figure 4-13 displays the angular velocity of the joints and Figure 4-14 their effort.

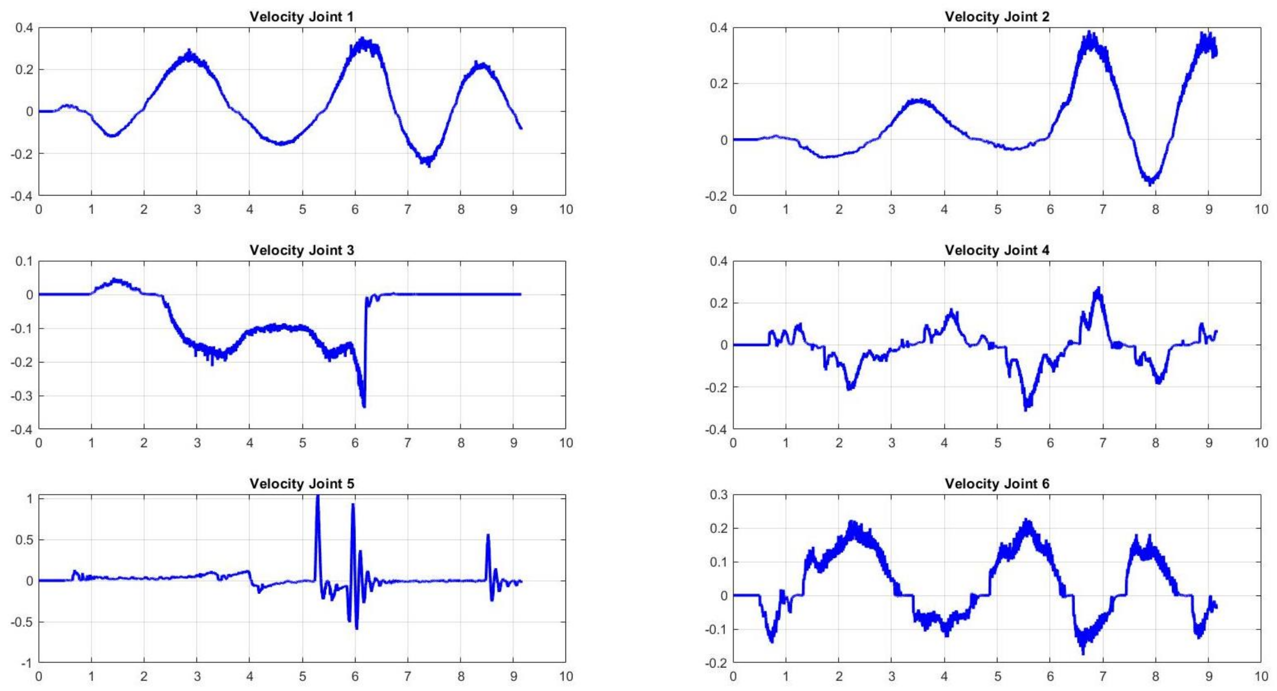


Figure 4-13: FourByThree –Joint Velocity

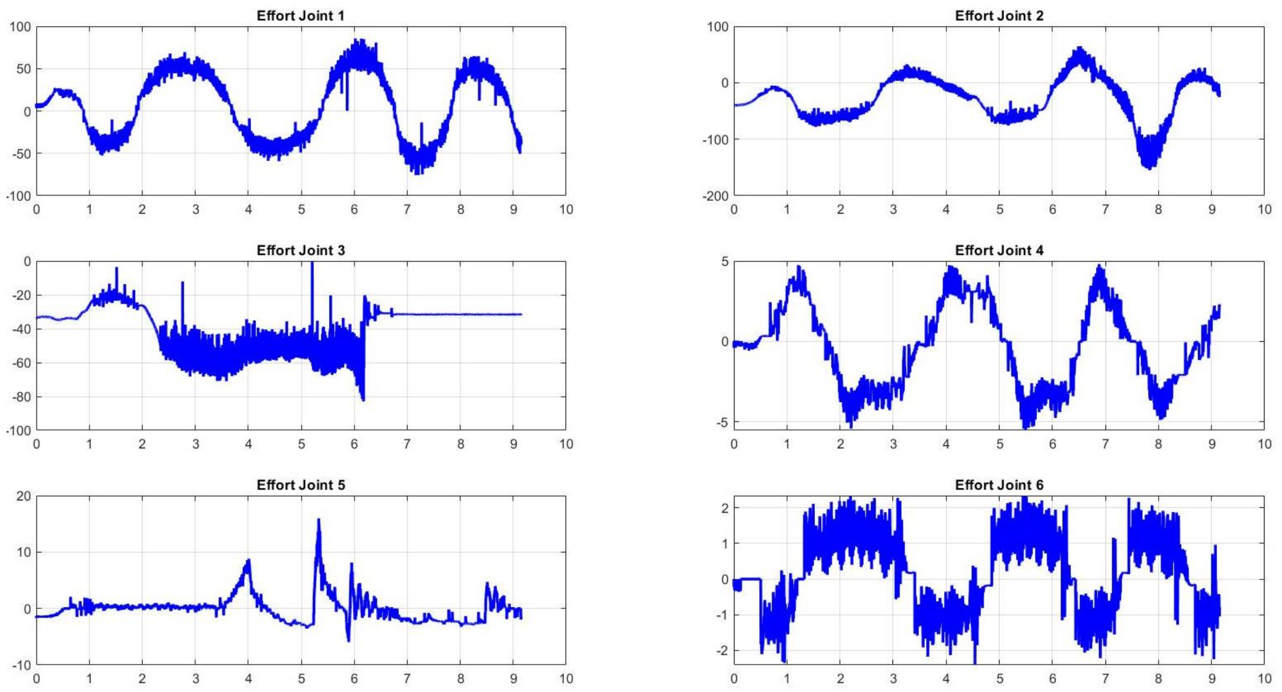


Figure 4-14: FourByThree –Joint Effort

4.3 Teaching By Manual Guidance And Path Repetition

The considered control strategy uses only gesture detection. It is the control method that gave the best results for both the robots. The robot configurations are memorized correctly and they are properly passed to MoveIt. All memorized points are reached during execution and the movement repetition for both the robots.

The control method program works very well and gives excellent results, allowing a reliable and natural way to program the robot by demonstration.

4.3.1 UR10e

Figure 4-15 displays the joints' position (in blue). The red dots in the graphs are the memorization instants. The robot is manually held in place during the memorization of the robot configuration. When the manual guidance is active, the robot control system simulates the presence of a return spring. Its effect is visible in the joints' position and joints' velocity (Figure 4-16) graphs. The execution command is launched when the robot returns to the starting position of the manual guidance. This avoids the abrupt blocking of the actuators.

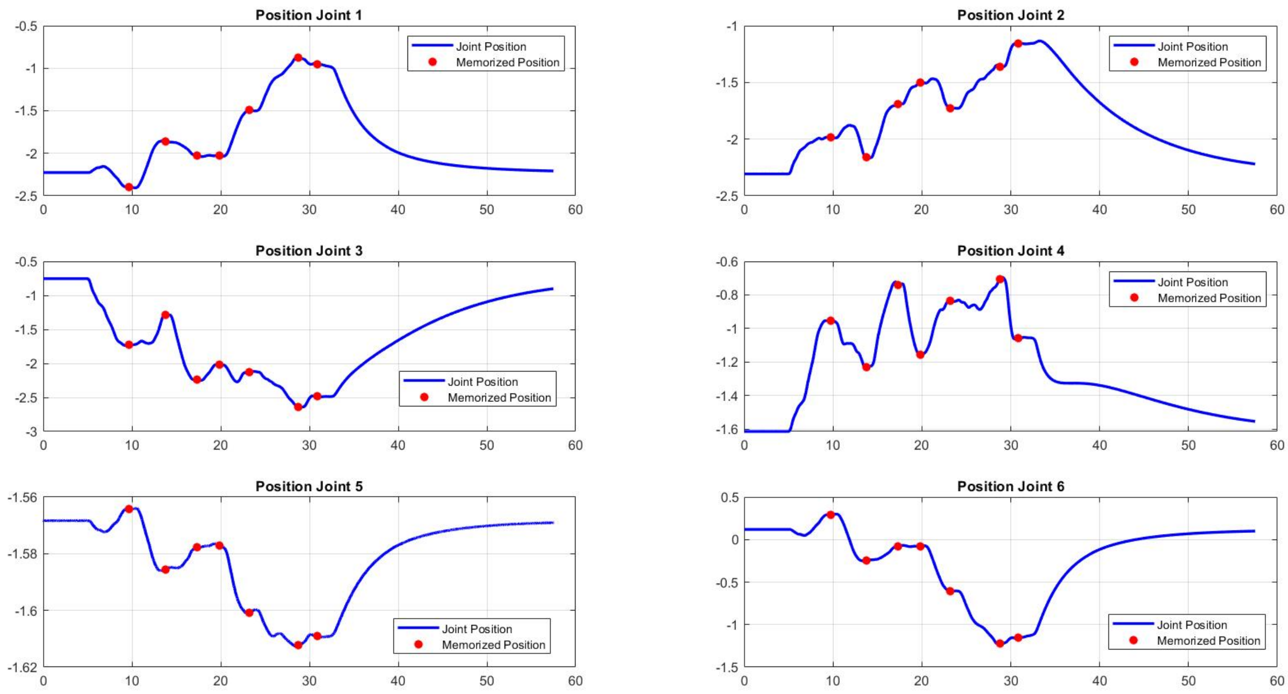


Figure 4-15: UR10e - Teach phase - Joint Position

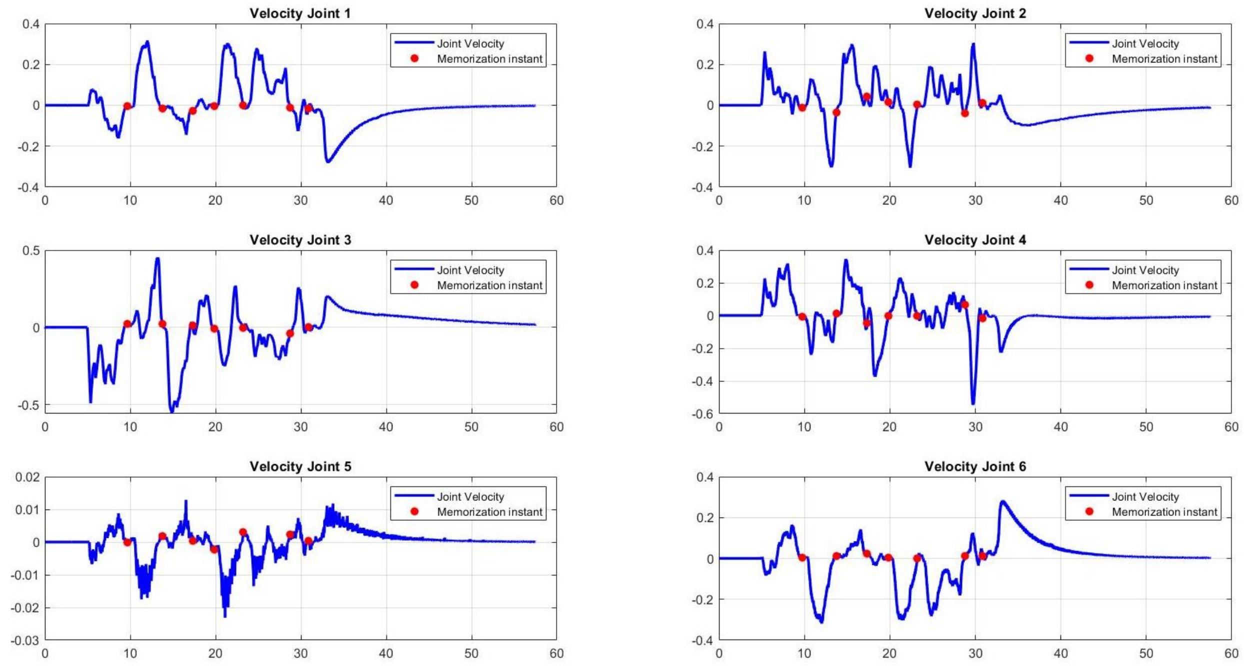


Figure 4-16: UR10e - Teach phase - Joint Velocity

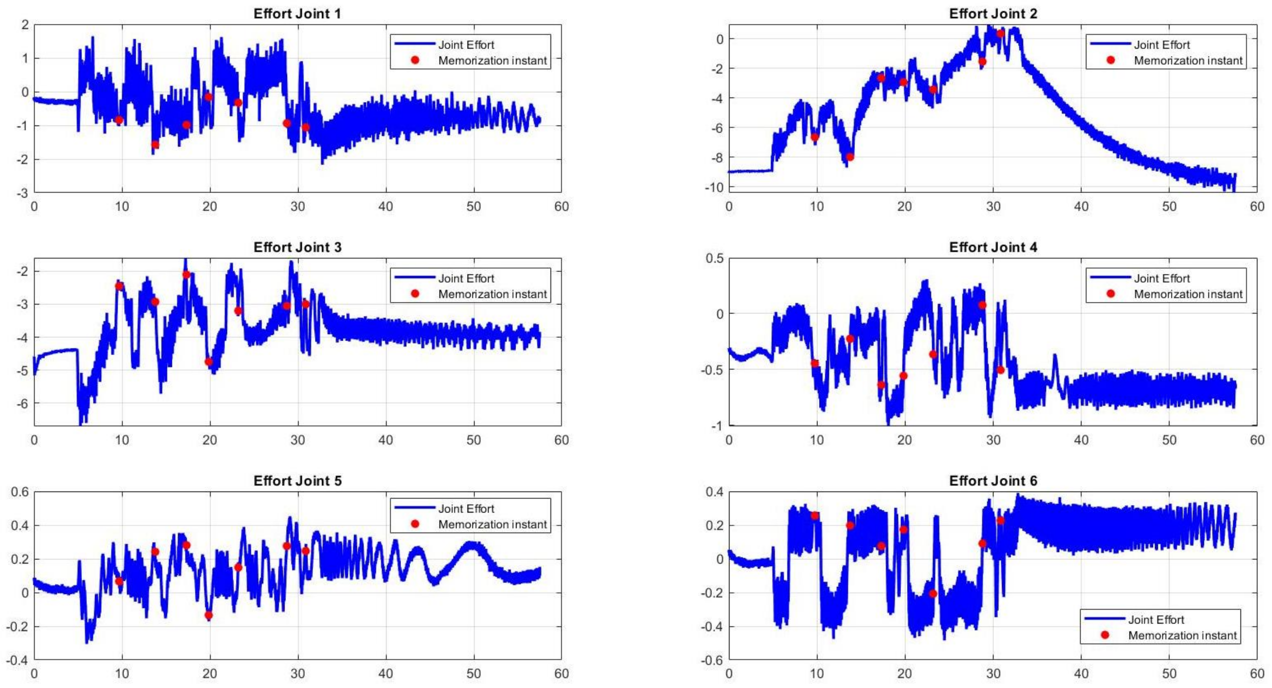


Figure 4-17: UR10e - Teach phase - Joint Effort

Figure 4-18 shows the comparison between joints' positions and their targets during the execution. The triangles indicate the instants in which the robot reaches the memorized configurations.

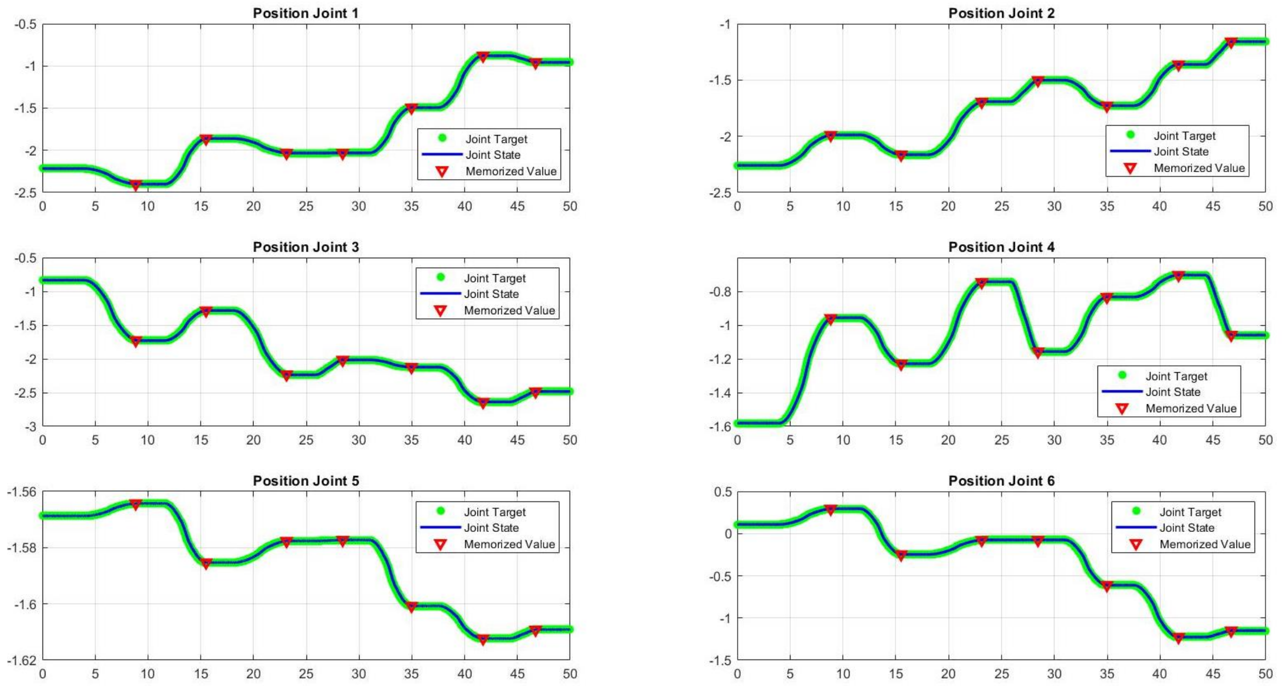


Figure 4-18: UR10e - Execution phase - Joint Position

The robot reaches a configuration when its joints' velocity is equal to zero, as observable in Figure 4-19.

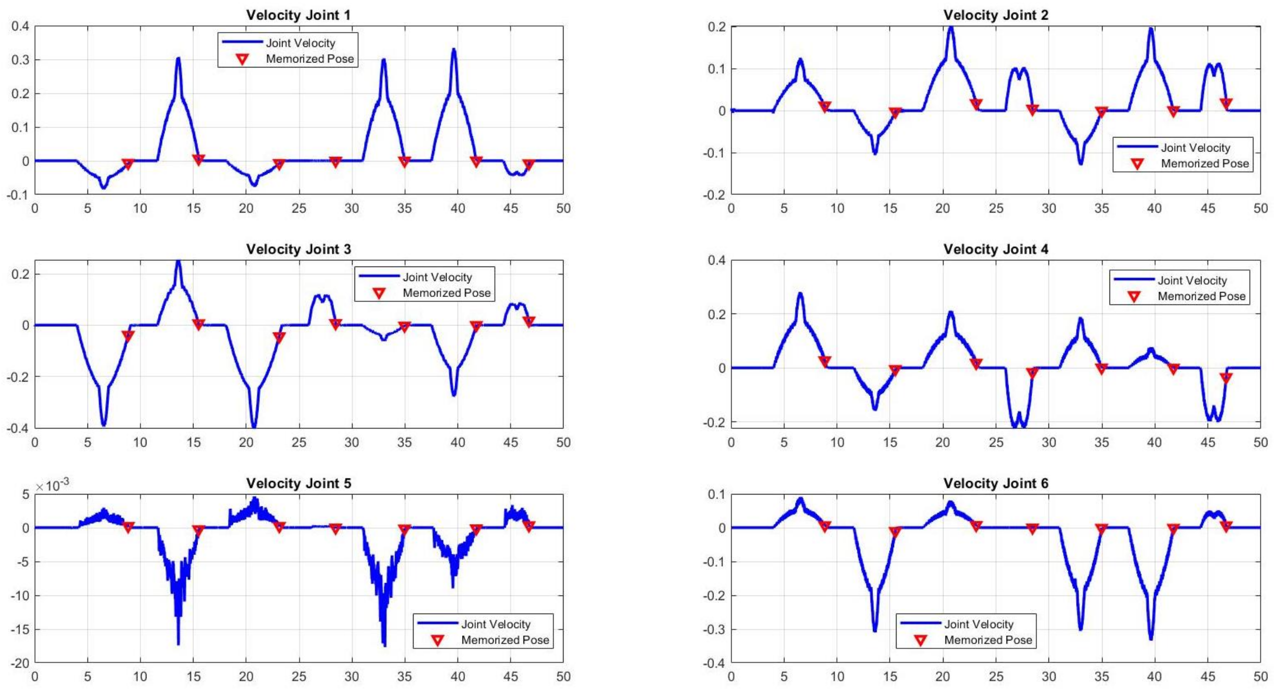


Figure 4-19: UR10e - Execution phase - Joint Velocity

It is possible to observe that the effort signal is less noisy when the actuator actively applies a force (Figure 4-20), than when an external force is applied (Figure 4-17).

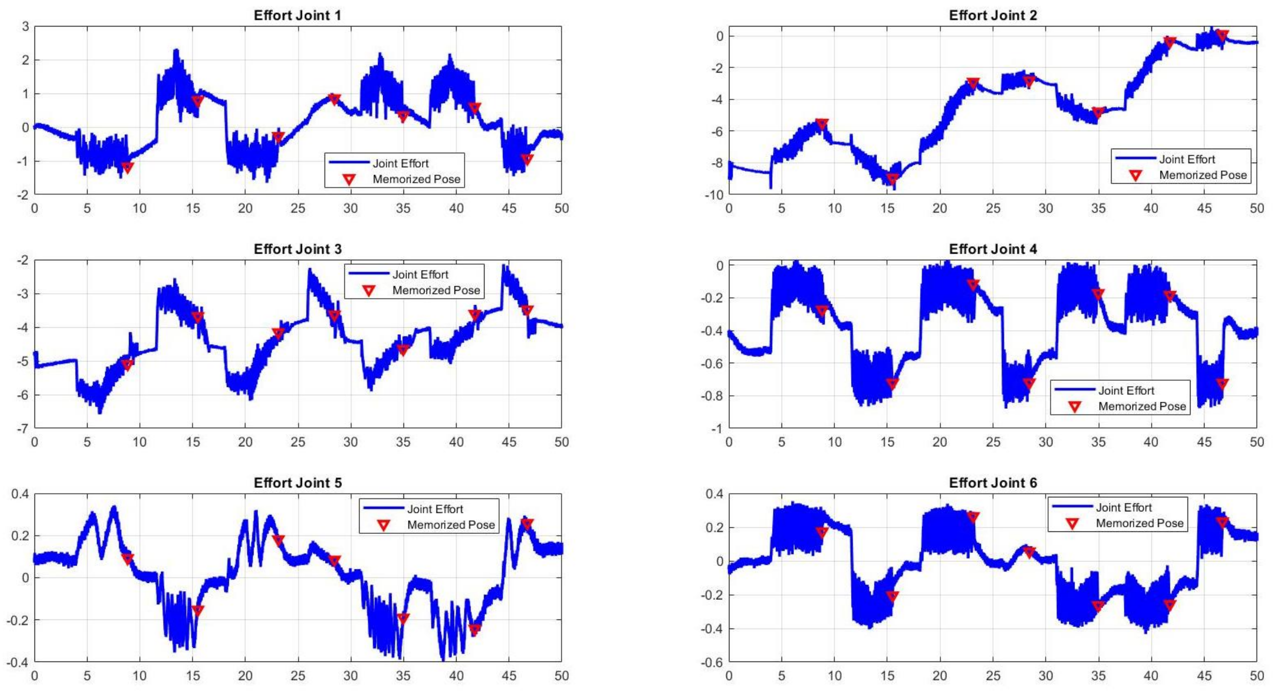


Figure 4-20: UR10e - Execution phase - Joint Effort

4.3.2 FourByThree

The absence of a force sensor on this robot does not allow the use of a manual guidance controller. Thus, the robot was moved from the computer interface. This does not affect the memorization process.

Figure 4-21 shows the joints' position (in blue) and the memorization instants (the red dots).

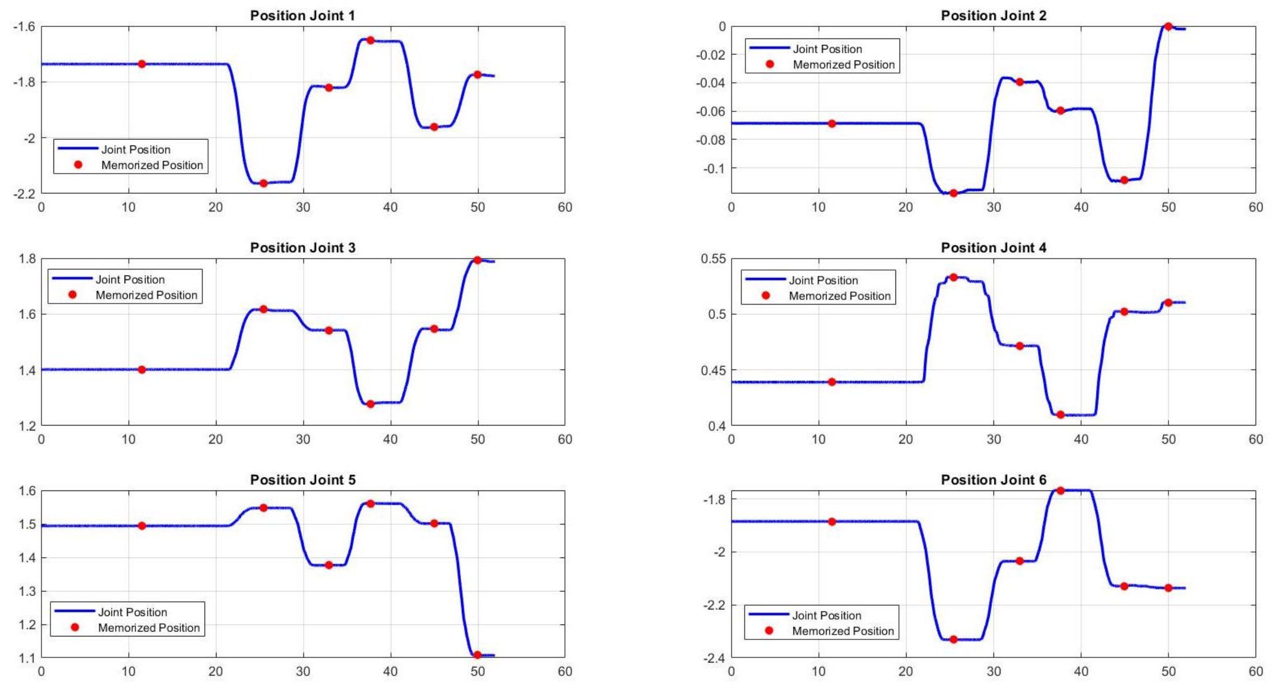


Figure 4-21: FourByThree - Teach phase - Joint Position

As observable from the joints' velocity (Figure 4-22), the manipulator oscillates a little bit after a movement.

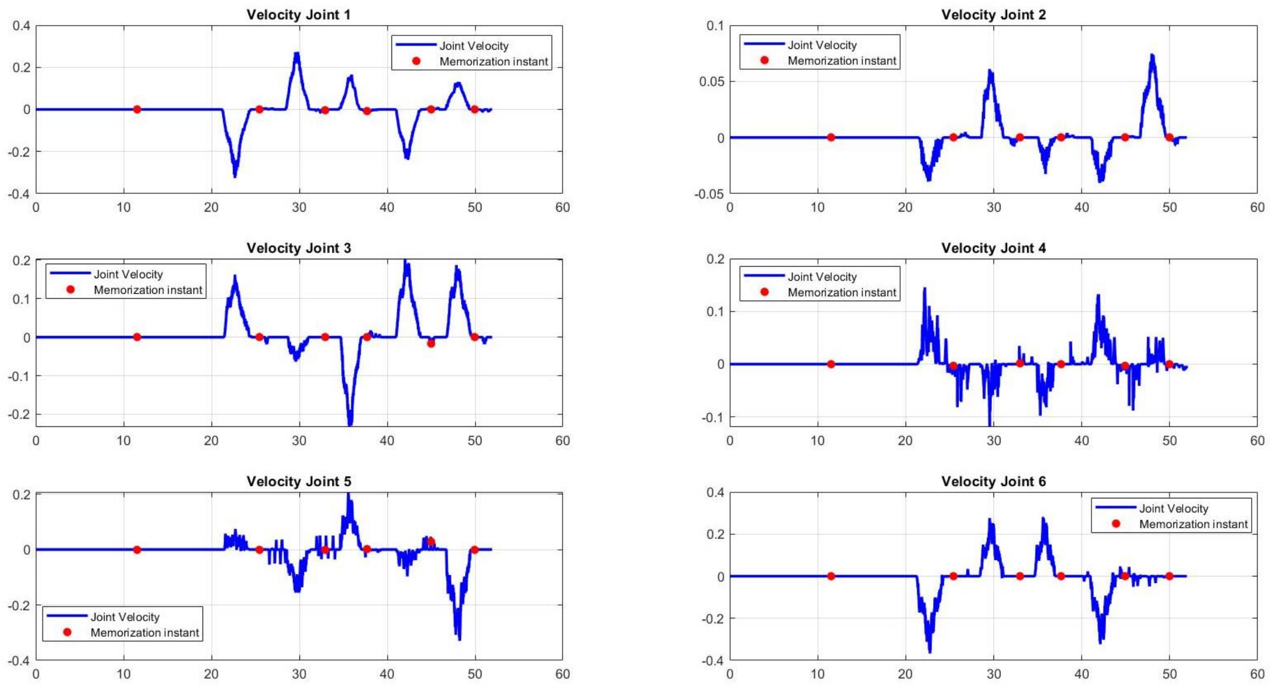


Figure 4-22: FourByThree - Teach phase - Joint Velocity

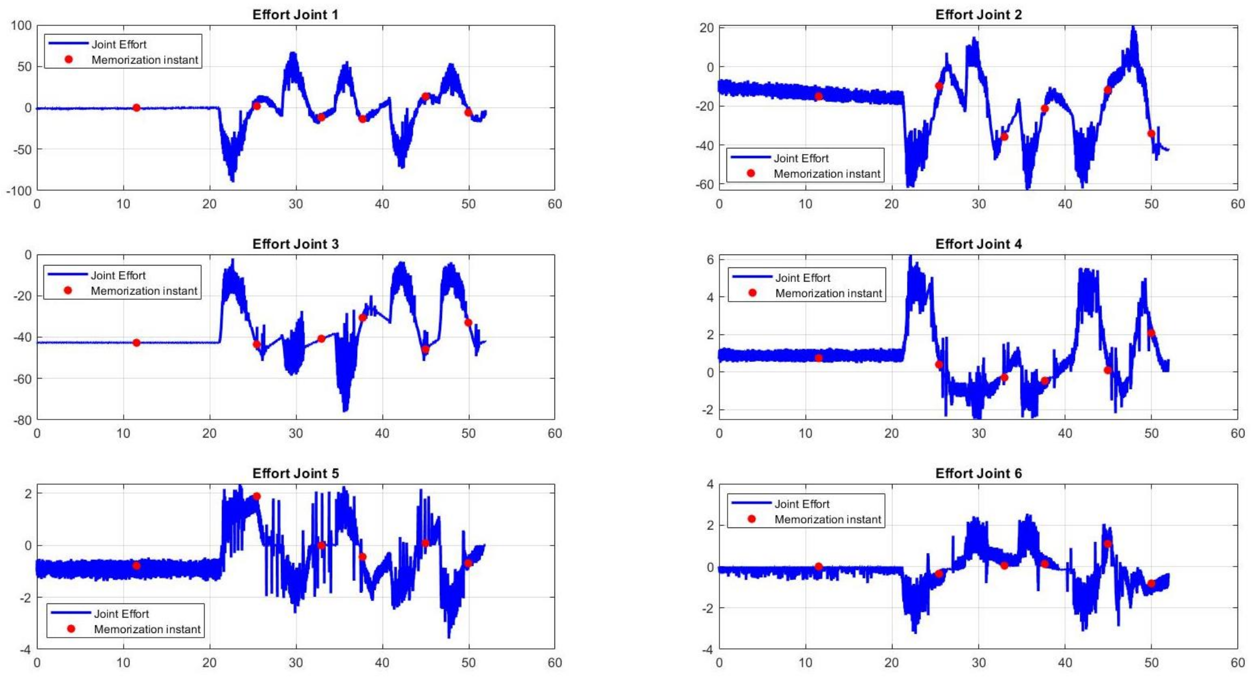


Figure 4-23: FourByThree - Teach phase - Joint Effort

Figure 4-24 displays the joints' positions and their targets during the execution. The robot reaches all the memorized points.

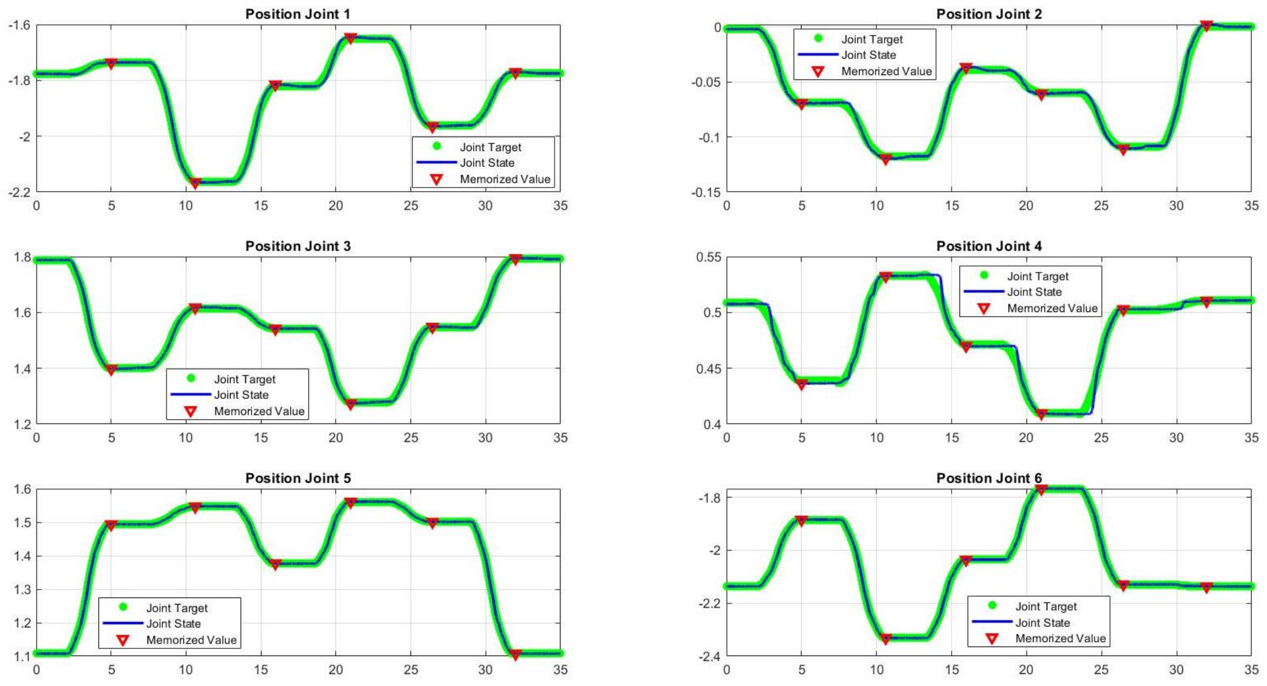


Figure 4-24: FourByThree - Execution phase - Joint Position

The robot oscillations are more evident looking at the velocity represented in Figure 4-25.

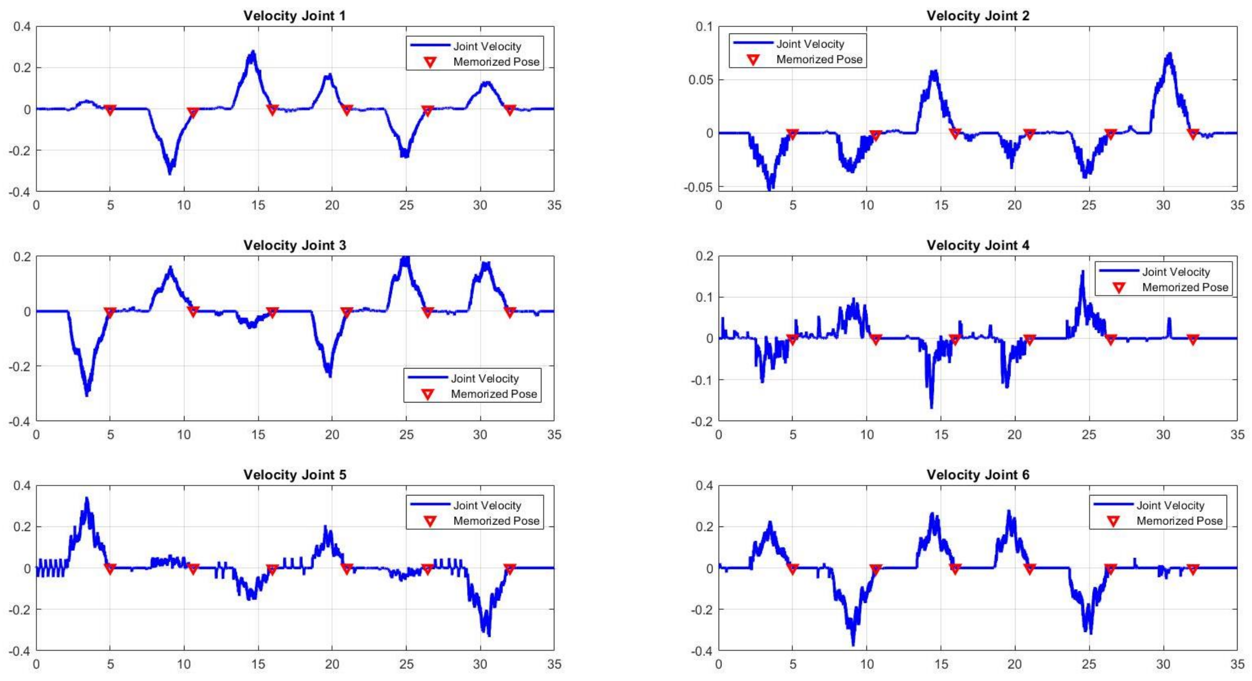


Figure 4-25: FourByThree - Execution phase - Joint Velocity

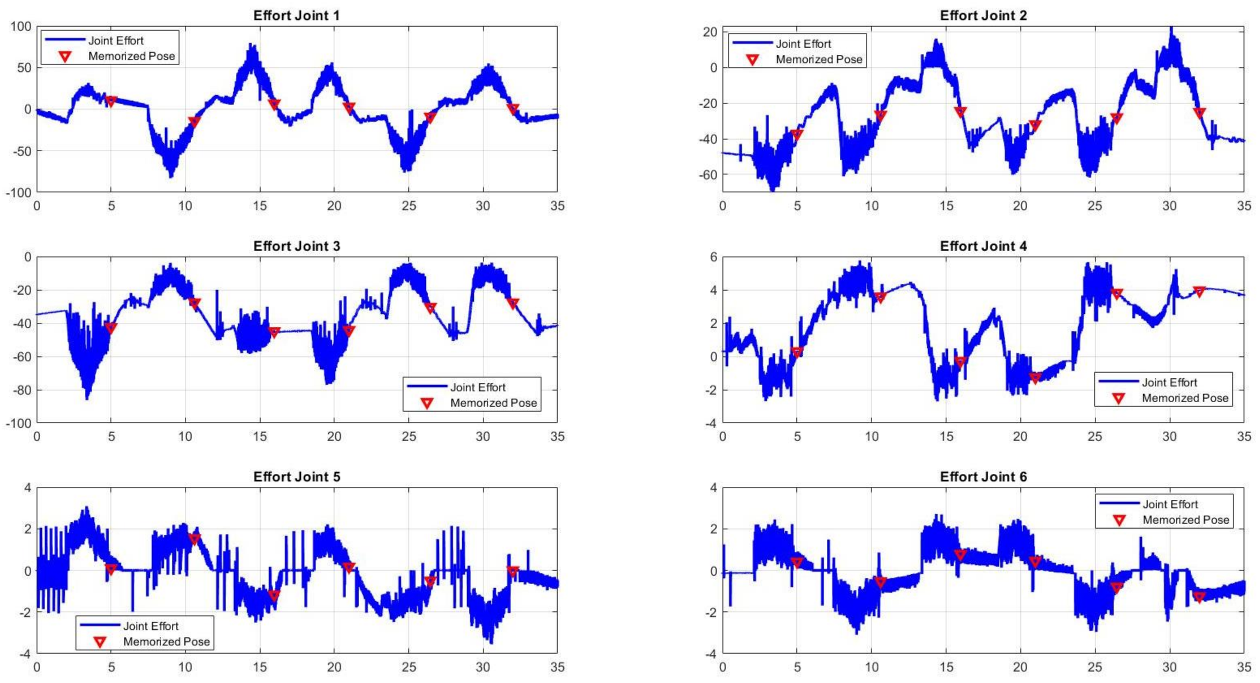


Figure 4-26: FourByThree - Execution phase - Joint Effort

4.4 Direction Control

This control strategy is affected by the Myo armband's same issues for the cartesian velocity control. The program correctly processes the data received from the accelerometer. Both the robots reach the indicates point. As for the cartesian velocity control, the UR10e gives the best result between the two robots. The control method gives satisfactory results.

The control method can be divided into two phases: the acquisition of the direction and the execution.

4.4.1 UR10e

Figure 4-27 compares the acceleration data given by the Myo and the filtered one. Figure 4-28 displays the Cartesian velocity. Those passages are the same done for the cartesian velocity control. It works properly.

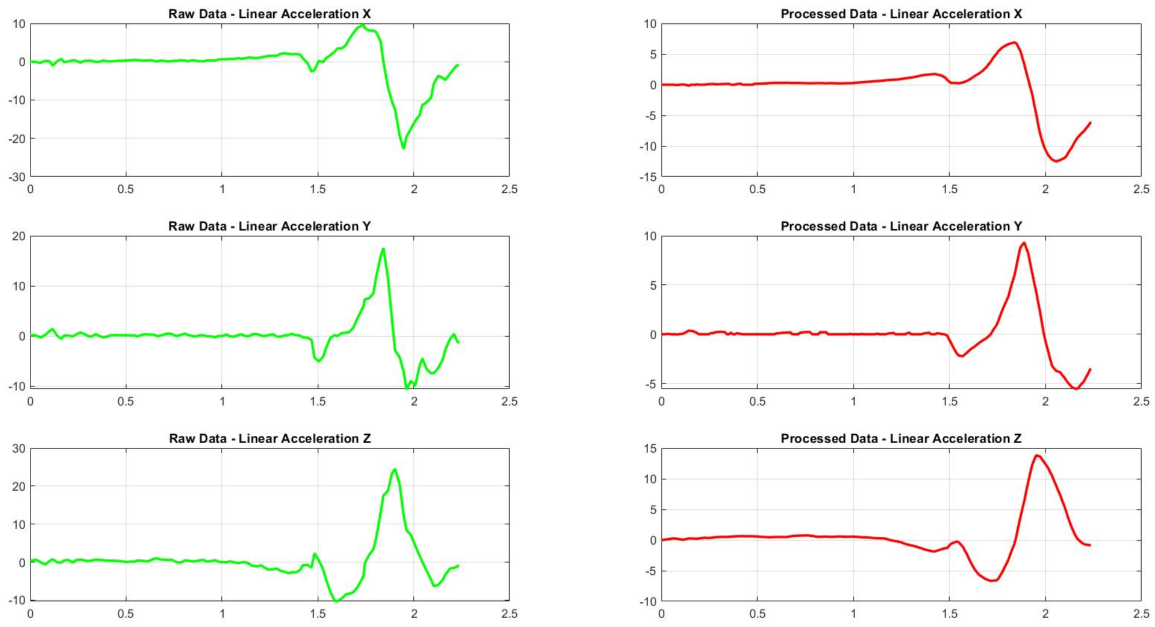


Figure 4-27: UR10e - Acquisition phase - Cartesian Acceleration

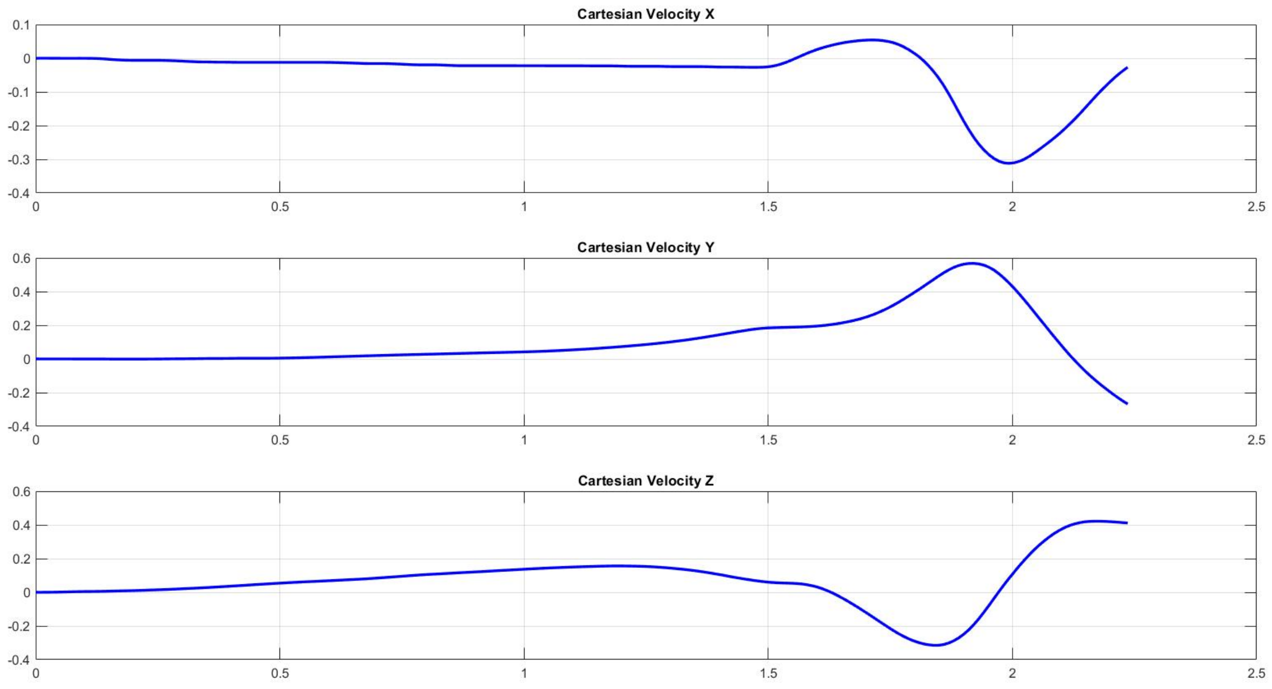


Figure 4-28: UR10e - Acquisition phase - Cartesian Velocity

The added passage is the integration of the cartesian velocity. The cartesian position is obtained and the versor of the displacement is calculated. Figure 4-29 displays the second integration results (in blue) and gripper cartesian position during the execution. The gripper Cartesian displacement is coherent with the versor value. The results show that the control system's effectiveness to reach the desired point.

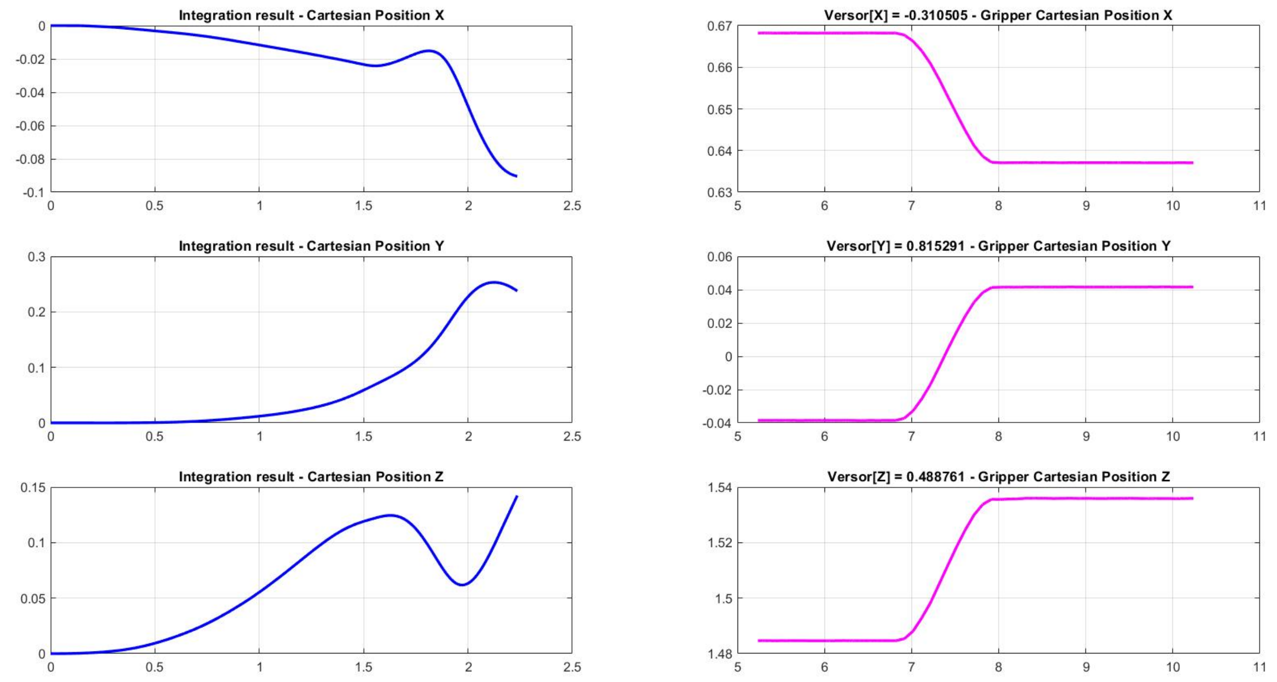


Figure 4-29: UR10e - Acquisition phase - Cartesian Position

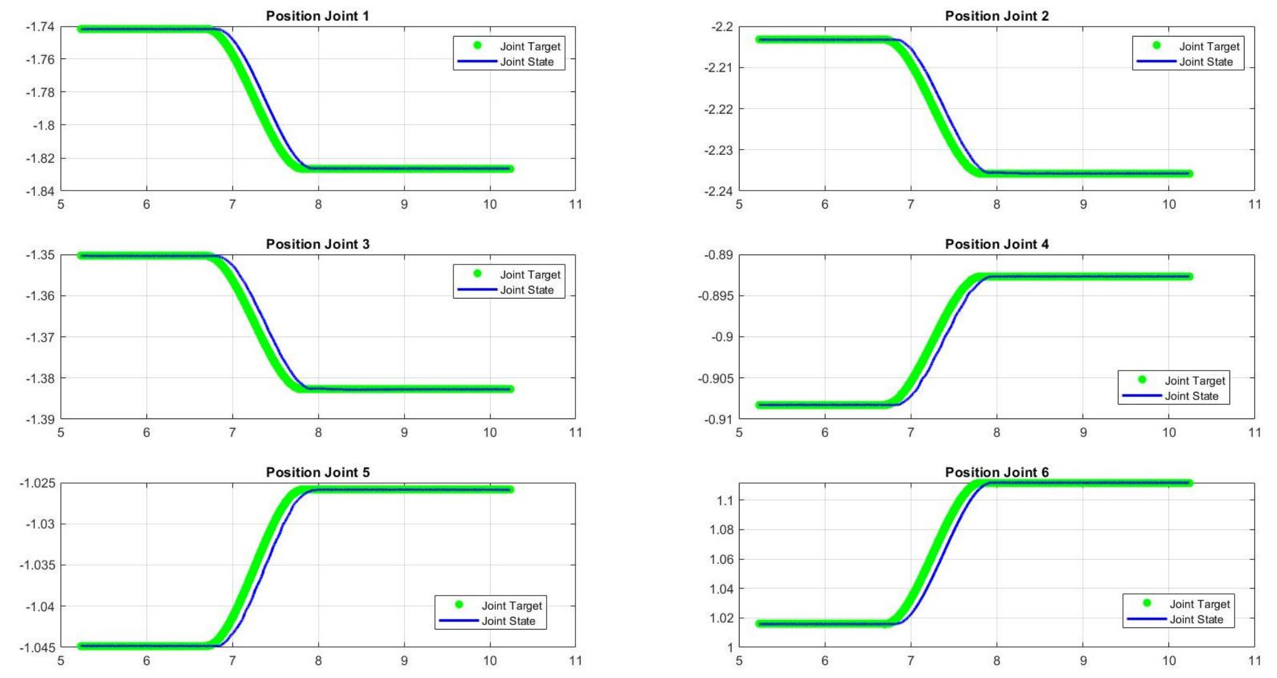


Figure 4-30: UR10e - Execution phase – Joint Position

As in the previous cases, the UR10e tracks precisely the joints' position target (Figure 4-30). The following graphs show the velocity (Figure 4-31) and the effort (Figure 4-32) of the joints.

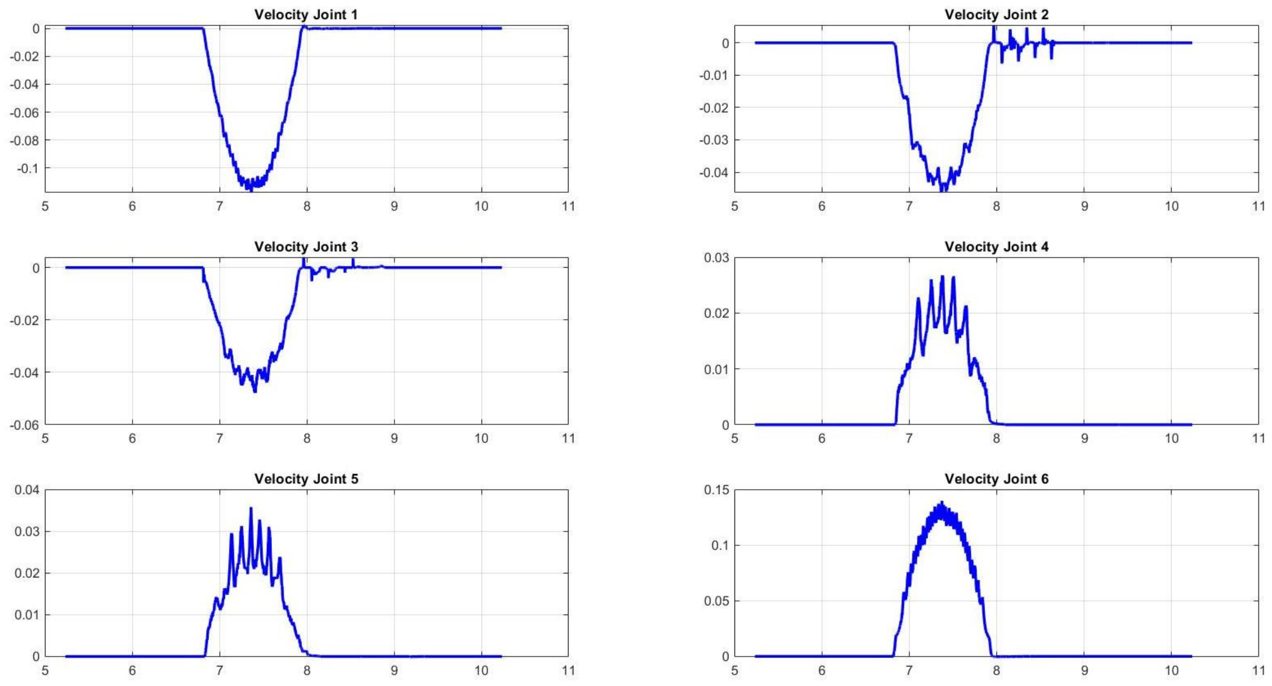


Figure 4-31: UR10e - Execution phase – Joint Velocity

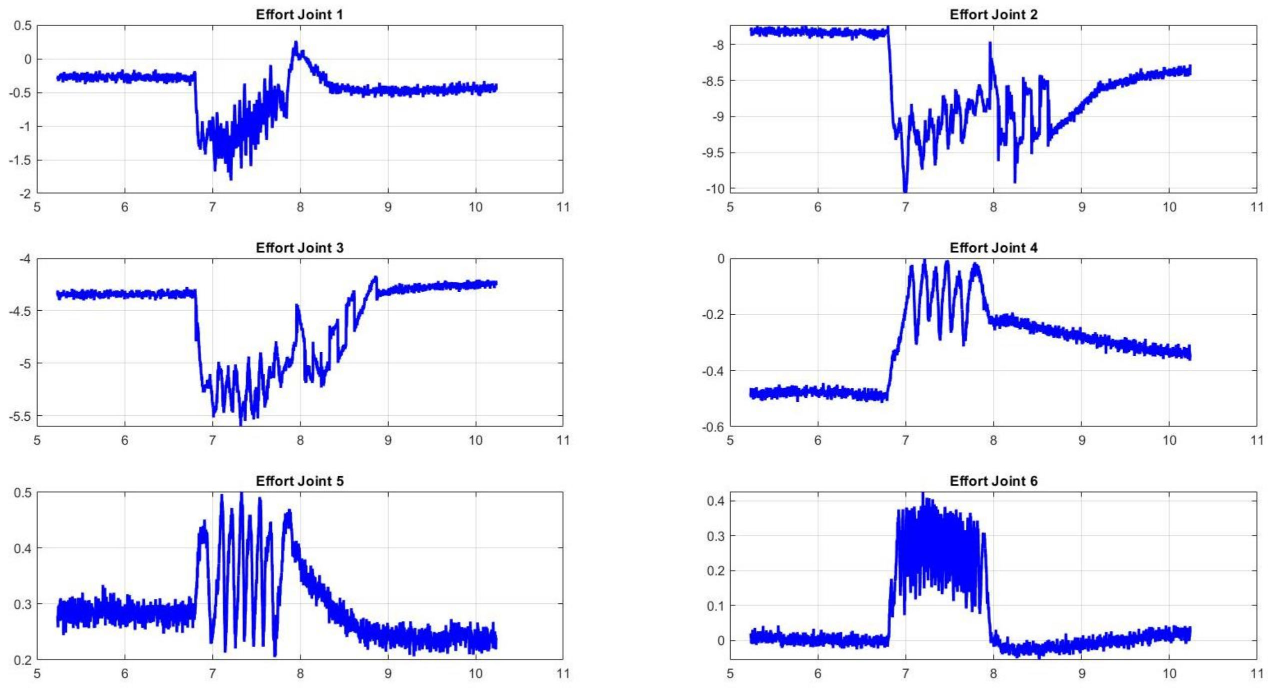


Figure 4-32: UR10e - Execution phase – Joint Effort

4.4.2 FourByThree

As for the UR10e, the acquisition phase performs very well. The graphs in Figure 4-33 and Figure 4-34 demonstrate the first part of data processing.

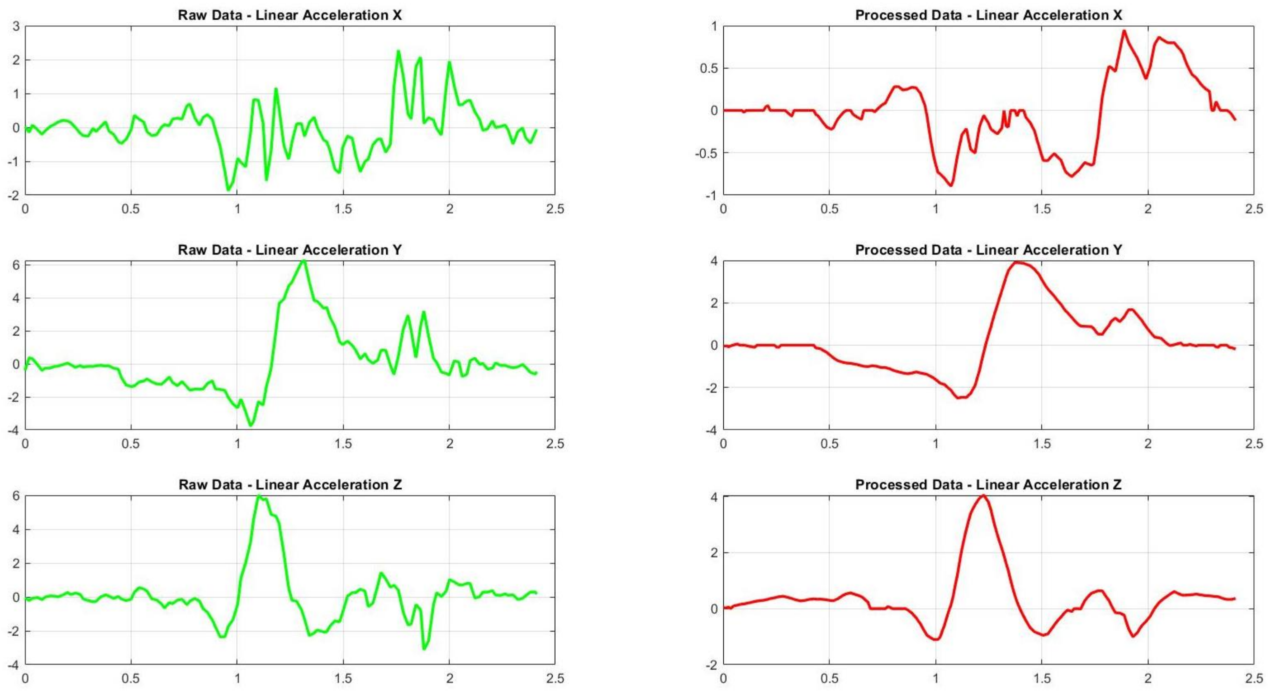


Figure 4-33: FourByThree - Acquisition phase - Cartesian Acceleration

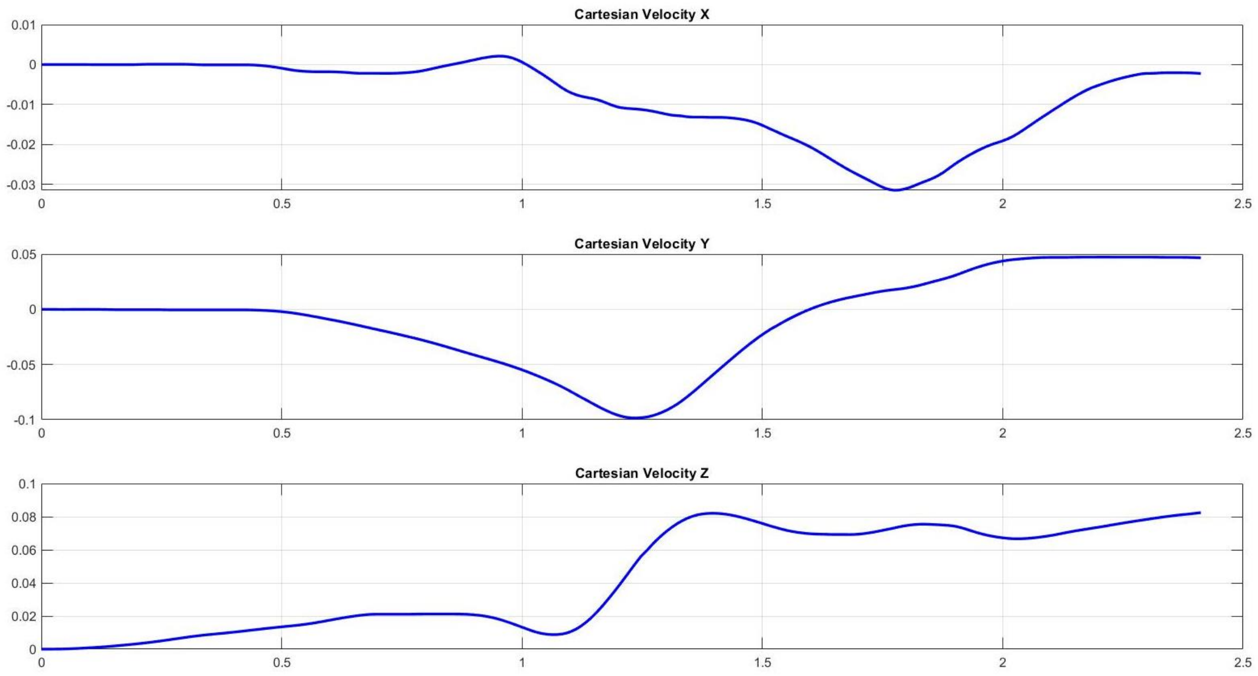


Figure 4-34: FourByThree - Acquisition phase - Cartesian Velocity

The FourByThree is not precise as the UR10e in the execution of the movement. However, the gripper Cartesian displacement is in the direction detected by the armband. Once again, an improved position controller could improve the results since the robot oscillates around the cartesian target point before reaching it.

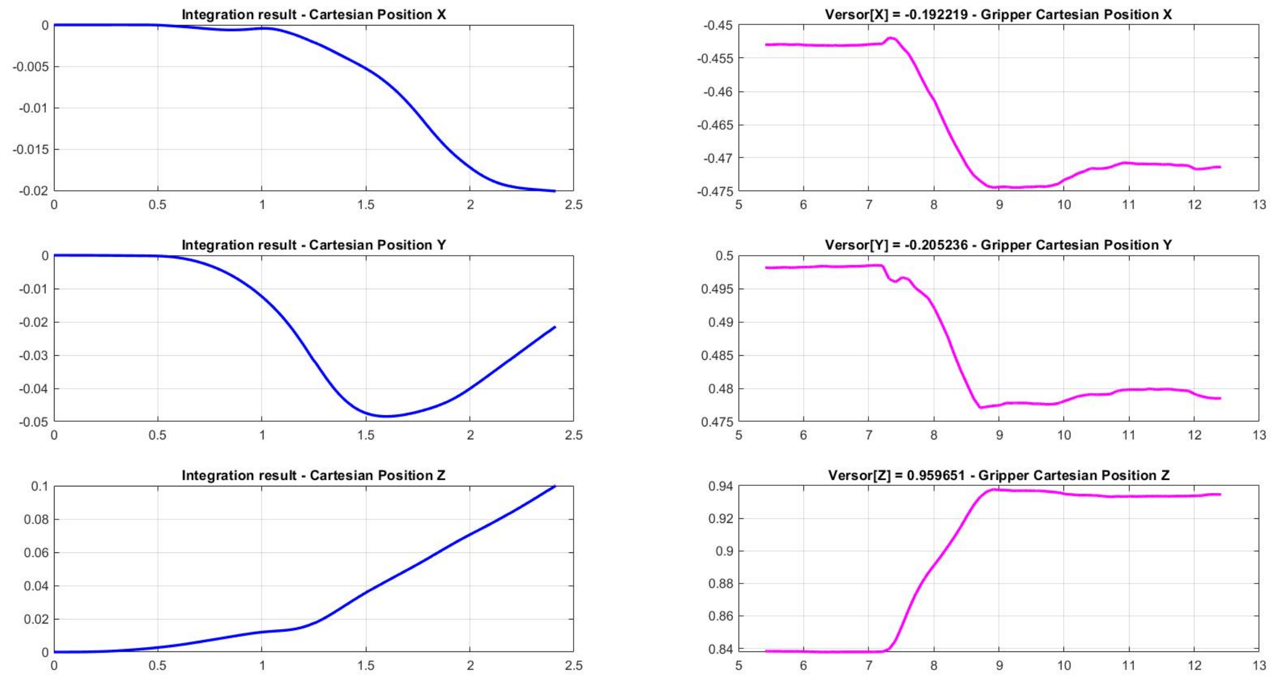


Figure 4-35: FourByThree - Acquisition phase - Cartesian Position

Oscillations and overshoots are evident by looking at the joints' position compared with their target (Figure 4-36).

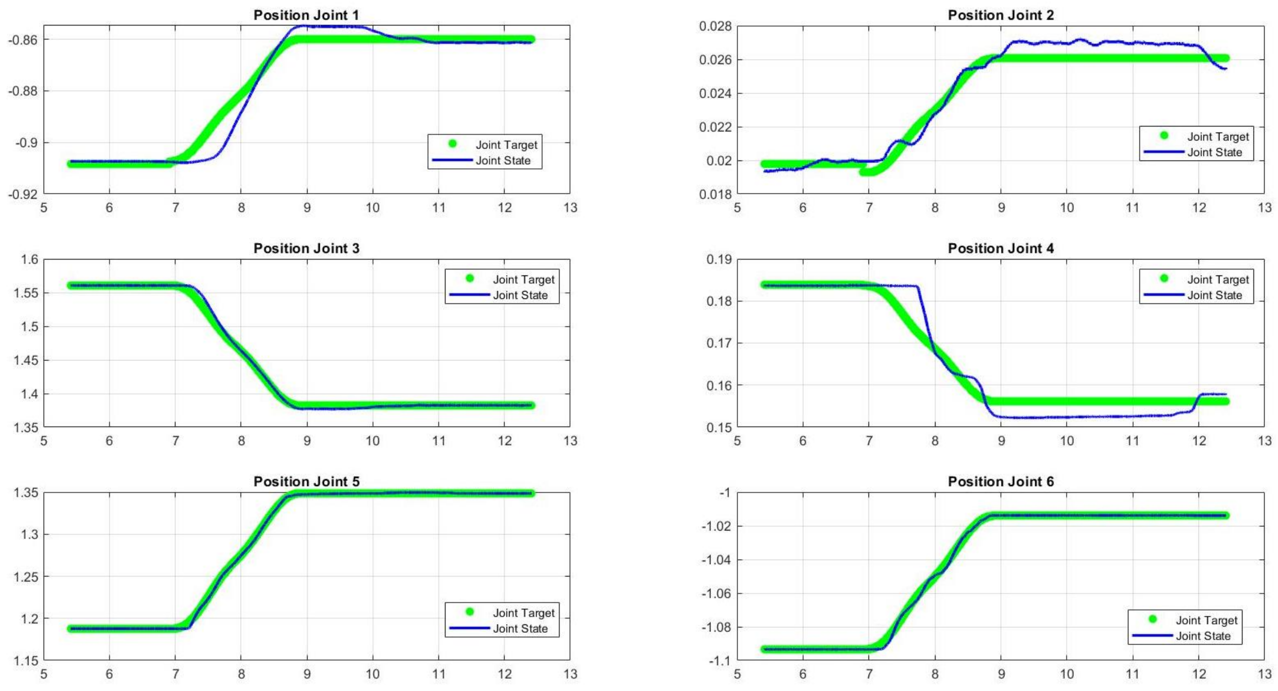


Figure 4-36: FourByThree - Execution phase – Joint Position

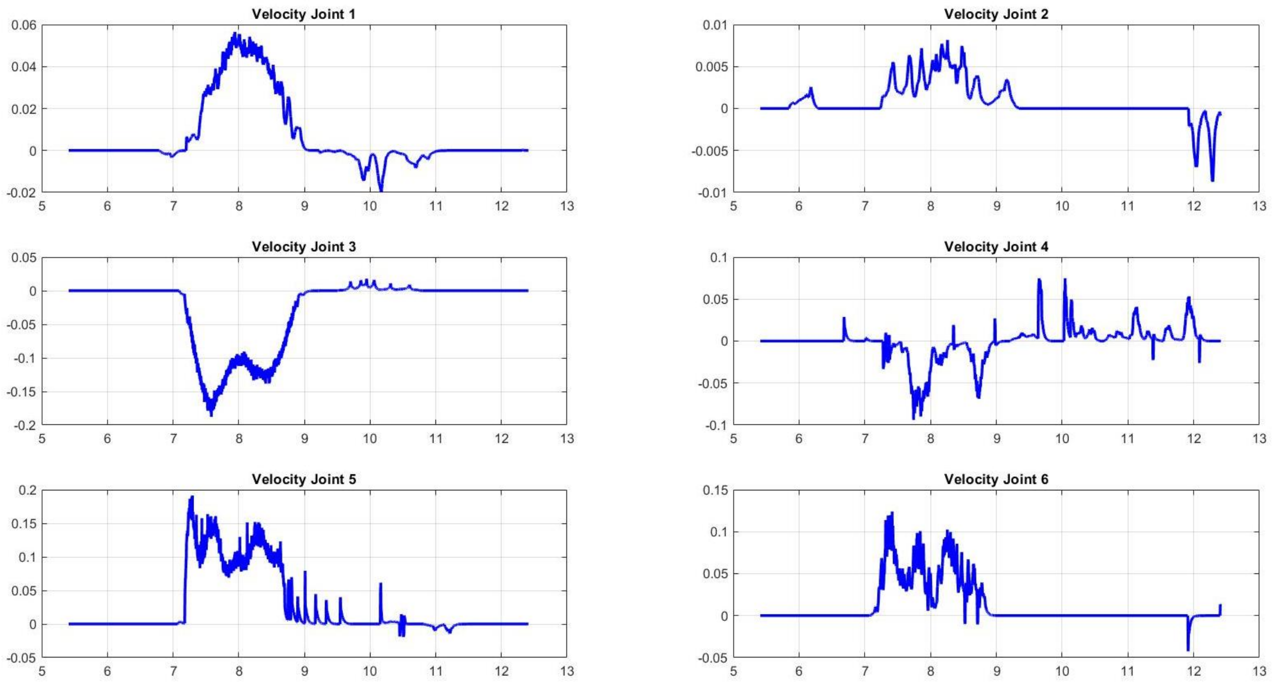


Figure 4-37: FourByThree - Execution phase – Joint Velocity

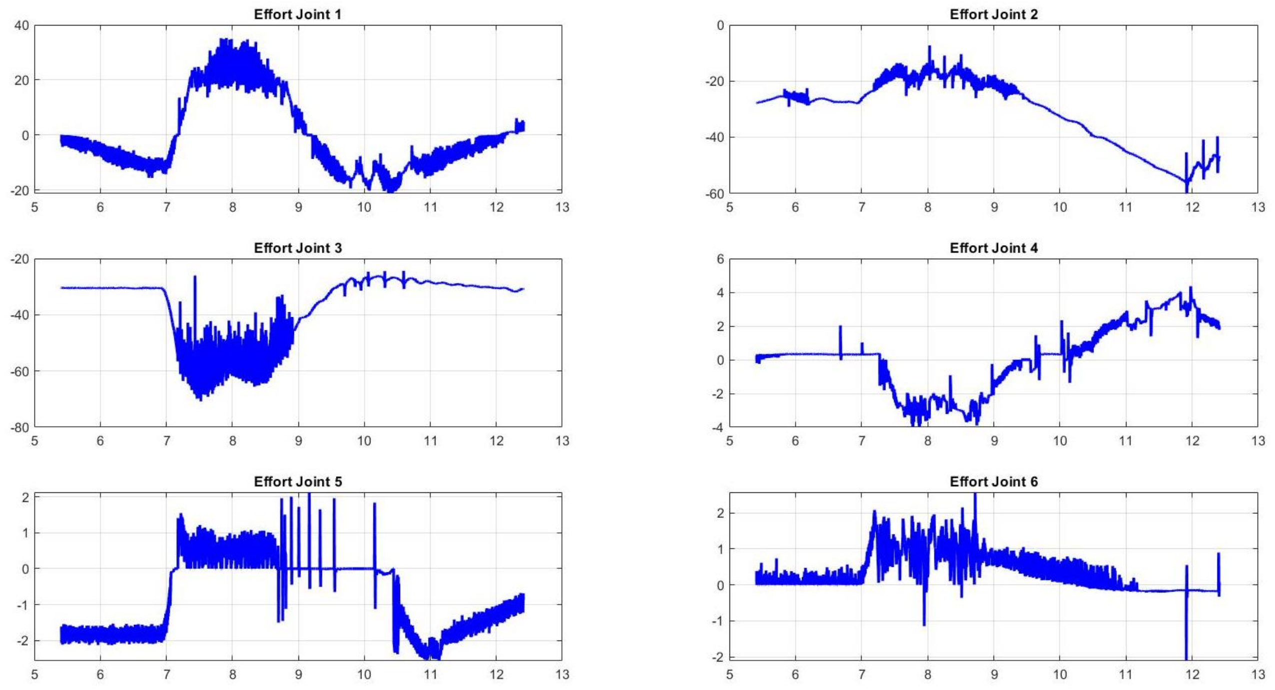


Figure 4-38: FourByThree - Execution phase – Joint Effort

CONCLUSION

The present thesis aimed to develop a control strategy for guiding collaborative robots using wearable EMG and IMU sensors. All the steps that lead to the development of the control system are described in the present work.

Three different control methods were developed:

- Cartesian velocity control: The IMU sensor is used to detect the linear acceleration of the arm. The linear velocity is obtained from it and then is used to control the cartesian velocity of the robot.
- Teaching by manual guidance with path repetition: The estimated gesture is used to save the waypoints of a path. The robot learns the path and can execute it when receives the execution command.
- Direction control: The robot is moved along the same direction as the arm movement. The direction is obtained from the arm acceleration.

The Myo armband is the sensing device. All the methods were tested on two different collaborative robots:

- The UR10e, which is an industrial robot with a force-torque sensor.
- The FourByThree robot, a prototype characterized by the presence of series-elastic actuators and, therefore by the presence of limited closed-loop bandwidth and oscillations.

The tests on the UR10e gave very satisfactory results. An improvement in the results is achievable by changing the sensing device. The Myo armband is a useful tool, but it is not precise in detecting all the possible movements. One solution could be the addition of other wearable IMUs on the arm that may improve the movement's recognition. Another solution could be the increase in the use of data from the EMG sensors.

Tests on the FourByThree gave good results but significantly improvable. The presence of SEAs actuators severely affects the robot's performance. The developed control system does not take them and the limited bandwidth of the inner controller into account. In the Cartesian velocity control case, performance may improve by adding at the control strategy a specific controller. It can use as input the arm velocity and return the robot kinematic and dynamic calculated to reduce oscillations. The more sophisticated calculation of the trajectory, kinematics, and dynamics could improve the other control methodologies' performance. The existing control system leaves this task to MoveIt. Future works could deal with designing an algorithm that calculates the best robot behavior and reduces oscillations.

ANNEX - DEVELOPMENT PHASES AND TEMPORARY PLANNING

The phases of development of this work are shown below in eight tasks:

- Task 1. Bibliographic review over collaborative robots and specifics control strategies.
- Task 2. Bibliographic review over Myo Armband applications.
- Task 3. Study over Myo Armband signals and amendment of its ROS interface.
- Task 4. Controller algorithm design.
- Task 5. Implementation as a C++ code of the algorithms.
- Task 6. Conducting tests with the simulator on RViz.
- Task 7. Conducting tests on collaborative robots.
- Task 8. Preparation of the report.

From mid-October to the end of November were dedicated 4 hours a day from Monday to Friday for 6 weeks. In Decembre and January an average of 5 hours a day from Monday to Friday for 4 weeks was worked. From mid-January to the end of February were dedicated 8 hours a day from Monday to Friday for 6 weeks. In May an average of 5 hours per 10 day was worked. This makes a total of approximately 510 hours of work. Table 1 shows the temporary planning of the tasks distributed over the months:

Tareas	Octubre				Noviembre				Diciembre				Enero				Febrero				Marzo			
1			🟧	🟧																				
2				🟧	🟧																			
3					🟧	🟧	🟧	🟧																
4								🟧	🟧	🟧	🟧			🟧	🟧	🟧	🟧							
5									🟧	🟧	🟧			🟧	🟧	🟧	🟧	🟧						
6										🟧	🟧			🟧	🟧	🟧	🟧	🟧	🟧					
7															🟧	🟧	🟧	🟧	🟧	🟧				
8																🟧	🟧	🟧	🟧	🟧	🟧	🟧		

Table 2: Gantt Diagram

USED COMPETENCES

The degree in Ingeniería Electrónica Industrial and the Laurea in Ingegneria dell'Automazione Industriale allows the students specialize in the field of Mechatronics and Automation, in addition to providing him with the necessary knowledge to develop his professional career in the industrial field.

Although there are differences in the definition of competences between both degrees as the systems in Italy and Spain are different, it can be stated that the main basic and transversal competences used in this work have been (using the UAL codes):

- CB1: That students have demonstrated to possess and understand knowledge in an area of study that is based on general secondary education, and is usually found at a level that, although supported by advanced textbooks, also includes some aspects that imply knowledge coming from the forefront of their field of study.
- CB2: That the students know how to apply their knowledge to their work or vocation in a professional manner and possess the skills that are usually demonstrated through the elaboration and defense of arguments and the resolution of problems within their area of study.
- CB3: That students have the ability to collect and interpret relevant data to make judgments that include a reflection on relevant social, scientific or ethical issues.

- UAL 1: Basic knowledge of the profession.
- UAL 2: Skill in the use of ICT.
- UAL 3: Ability to solve problems.
- UAL 7: Learning a foreign language.
- UAL 9: Ability to learn to work autonomously.

More specifically, knowledge in process control and automation has been applied, together with mechatronics design. The specific skills used in the field of Industrial Engineering (Mechatronics and Automation) have been:

- E-CT3: Knowledge in basic and technological subjects, which enables them to learn new methods and theories, and gives them versatility to adapt to new situations.
- E-CT4: Ability to solve problems with initiative, decision making, creativity, critical reasoning and to communicate and transmit knowledge, skills and abilities in the field of Industrial Engineering.
- E-CB3: Basic knowledge of the use and programming of computers, operating systems, databases and computer programs with application in engineering.
- E-CRI6: Knowledge about the basics of automatisms and control methods.
- E-CTEE7: Knowledge and capacity for modeling and simulation of systems.
- E-CTEE8: Knowledge of automatic control and control techniques and their application to industrial automation.
- E-CTEE11: Ability to design control systems and industrial automation.
- E-TFG: Original exercise to be carried out individually and presented and defended before a university tribunal, consisting of a project in the field of specific Industrial Engineering technologies of a professional nature in which the competences acquired in the teaching are synthesized and integrated.

BIBLIOGRAPHY

[1]. G. Legnani, I. Fassi. *Robotica Industriale*. Città Studi Edizioni, 2019.

[2]. I. Maurtua, N. Pedrocchi, A. Orlandini, J. de Gea Fernández, C. Vogel, A. Geenen. *FourByThree: Imagine humans and robots working hand in hand*. IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA), 2016. DOI: 10.1109/ETFA.2016.7733583

[3]. S. Ghidini. *Tesi di Laurea Magistrale – Development of advanced tecniques for the robust control of a compliance robot with serial elastic actuators*. Università degli Studi di Brescia. 2017.

[4]. S. Ghidini, M. Beschi, N. Pedrocchi. *A robust linear control strategy to enhance damping of a series elastic actuator on a collaborative robot.* Journal of Intelligent & Robotic Systems, 98, 627–641, 2020. DOI: 10.1007/s10846-019-01071-5

[5]. S. Ghidini, M. Beschi, N. Pedrocchi, A. Visioli. *Robust tuning rules for series elstic actuator PID cascade controllers*. IFAC (International Federation of Automatic Control) – PapersOnline, 51(4), 220-225, 2018. DOI: 10.1016/j.ifacol.2018.06.069

[6]. Universal Robots. *e-Series – Technical Brochure*. Universal Robot. 2019. https://www.universal-robots.com/media/1802432/e-series-brochure.pdf

[7]. B. Tveit. Master's Thesis in Applied Physics and Mathematics *- Analyzing behavioral biometrics of handwriting using Myo gesture control armband*. UiT – The Artic University of Norway. 2018.

[8]. P. Visconti, F. Gaetani, G. A., Zappatore and P. Primiceri. *Technical features and functionalities of Myo armband: an overview on related literature and advanced applications of myoelectric armbands mainly focused on arm prostheses*. International Journal on Smart Sensing and Intelligent Systems, 11(2), 1-25, 2018.

[9]. Github – ros_myo Package. *https://github.com/uts-magic-lab/ros_myo*

[10]. Robot Operating System: *http://wiki.ros.org/it.*

[11]. Robot Operating System. *Official Page: http://www.ros.org/*.

[12]. M. Mariotti. *Tesi di Laurea – Simulazione di robot industriali con simulatore V-Rep e connessione con ambiente ROS*. Università degli Studi di Brescia. 2018.

[13]. VMware. *Official Page: http://www.vmware.com/*.

[14]. Microsoft. *WSL Documentation - https://docs.microsoft.com/it-it/windows/wsl/*.

[15]. VirtualBox. *Official Page: https://www.virtualbox.org/*.

[16]. MathWorks. *MATLAB - Official Page: https://it.mathworks.com/*.

[17]. Itemis. *Yakindu Statechart Tools – Documentation: https://www.itemis.com/en/yakindu/state-machine/documentation.*

[18]. A. Quarteroni, F. Saleri, P. Gervasio. *Calcolo scientifico: esercizi e problemi risolti con MATLAB e Octave*. Springer Verlag. 2017.

UNIVERSIDAD DE ALMERÍA

Resumen/Abstract

El uso de robots está muy extendido en muchas aplicaciones industriales, pero pueden considerarse un riesgo si se requiere la colaboración con un trabajador. En los últimos años, las investigaciones y los estudios sobre la interacción hombre-robot (HRI) han aumentado y se ha desarrollado una nueva generación de robots: los robots colaborativos.
Los robots colaborativos nacen para trabajar con seguridad con humanos en el mismo espacio de trabajo, manteniendo la eficiencia y productividad de un robot industrial. Eso es posible gracias a una sinergia de estrategias de control y soluciones mecánicas y electrónicas específicas, como SEA.
La interacción hombre-robot provoca un nuevo problema: la interacción entre los trabajadores y el robot colaborativo.
El presente Trabajo de Fin de Grado trata sobre el control de robots y la interacción hombre-robot: en este trabajo se presenta una estrategia de control para guiar robots colaborativos utilizando sensores EMG y IMU portátiles. El objetivo final es permitir al operador humano el control del comportamiento del robot mediante el uso de los sensores.
Después de una primera sección descriptiva sobre la instrumentación, se presentan las etapas de diseño y implementación del sistema de control. El programa de control se implementa dentro del entorno ROS. El brazalete Myo es el dispositivo de detección.
Todos los métodos fueron probados en dos robots colaborativos diferentes que tienen seis grados de libertad: FourByThree, caracterizado por la presencia de actuadores elásticos en serie, y UR10e, producido da Universal Robot.
El sistema de control resultante puede dividirse en tres métodos de control diferentes:
• Control de velocidad cartesiana: la velocidad lineal del brazo se utiliza para controlar la velocidad cartesiana del robot.
•Enseñanza mediante guía manual y repetición de la trayectoria: el robot se mueve manualmente y se utiliza un comando gestual del trabajador para guardar la configuración del robot. De esta manera, el robot aprende un ejecutable de ruta por sí mismo. Otro comando de gesto permite la repetición de la última ruta memorizada.
• Control de dirección: el robot se mueve en la misma dirección que el movimiento del brazo.
La sección final del presente trabajo muestra los resultados de las pruebas con los robots.