TIAGo robot teleoperation via Body Machine Interface for cervical Spinal Cord Injured subject assistance

Ermanno Girardo

DIBRIS - Department of Computer Science, Bioengineering,

Robotics and System Engineering
University of Genova

In partial fulfillment of the requirements for the degree of
Doctor of Robotics Engineering
October, 2022

Acknowledgements

Maura **Casadio**[1], Camilla **Pierella**[2], Fulvio **Mastrogiovanni**[3], Alessandro **Carfì**[4].

[1]**Associate Professor** @ DIBRIS "Department of Computer science, Bioengineering, Robotics and Systems engineering", Università degli Studi di Genova, Via Opera Pia 13, Genova, 16100, Italy.

[2]**Research Fellow** @ DINOGMI "Department of Neuroscience, Rehabilitation, Ophthalmology, Genetics and Maternal-Infant Sciences;Teaching Fellow @ DIBRIS Università degli Studi di Genova, Via Opera Pia 13, Genova, 16100, Italy.

[3]**Associate Professor** @ DIBRIS "Department of Computer science, Bioengineering, Robotics and Systems engineering", Università degli Studi di Genova, Via Opera Pia 13, Genova, 16100, Italy.

[4]**Postdoctoral Researcher and Member** @ TheEngineRoom Università degli Studi di Genova, Via Opera Pia 13, Genova, 16100, Italy.

To all the Master and PhD students of Robotics Engineering at the University of Genova.

Abstract

Support/Assistive robotics is becoming more and more popular thanks to innovation of technologies and computer science. Support and assistive robots are used today in hospitals and hospices to support and assist older people and subjects with mobility problems. In particular cSCI (cervical Spinal Cord Injury) subjects are often forced to lead a static life unable to move legs and upper body, in some cases also the arms. The scope of this article is to expose a model and a re-educational path to be able cSCI subjects to teleoperate through body gestures an assistive robot TIAGo, with the aim to stimulate the subjects recovering motor functions and to provide assistance for activities of daily living (**ADL**). Body-Machine Interfaces (**BoMI**) have been proven to be capable of harnessing residual joint motions to control devices such as computer cursor and virtual simulated or physical wheelchair and to promote motor recovery. BoMI will be the starting point to develop an architecture entirely ROS (Robotic Operating System) based, that will allow a cSCI subject, to teleoperate the mobile base and the arm of TIAGo. The wanted final result is to allow a cSCI subject to teleoperate autonomously TIAGo robot in order to reach a target object, thanks to his body residual motion, always taking in mind the constraints related to cSCI subject.

Contents

List of Figures

Chapter 1

Introduction

People that unfortunately have suffered cervical spinal cord injury (**cSCI**) are forced to lead a static life since they have lost motor and/or sensory functions below the level of the lesion. cSCI is a devastating event that change the life style of a person forever, not only physically but also psychologically. However the majority of cSCI people maintain limited motions of upper body. For these reasons physical rehabilitation is a key point for cSCI people,since today no curative treatments exist, in order to regain some motor skills and also supporting them psychologically. In the last decades researchers developed many software architectures and technologies to support and assist dependent people for example: frail elderly, blind, cSCI, cerebral palsy and rheumatoid arthritis people. The result is the birth of systems, known as assistive robots, able to collect and analyze data from the real environment and perform actions that benefits people with disabilities and seniors. They can be divided into two main groups: fixed-base or mobile robots. Since the purpose of this work is to provide a way to assist cSCI people in an open area, a mobile base robot is more suitable because it is able to move in the surrounding environment. One possible and well consolidate way to control robots is teleoperation. Robot tele-operation has been studied for many centuries across application areas ranging from search-rescue (9) and space exploration (10) to surgery (11). The idea is to remotely operate a robot in dangerous condition, unreachable environment or simply in order to perform strenuous or impossible actions for humans. In the last decades many interfaces using different devices have been designed; think for example joystick, pedals, haptic feedback (12), vision sensors, voice commands, gestures (5) or simply mouse cursor (7) in order to teleoperate supportive and rehabilitative robots. A great number of existing technologies are the proof that not every type of interface can be used for cSCI patient. **El-E Robot**, one of the first assistive robot that is aimed to help people suffering from certain disabilities and motor impairments, developed by Georgia Institute of Technology , see (18). It is able to pick up and bring objects

from the floor thanks to its gripper. It is not the ideal one for a cSCI since the targets are selected using a laser pointer. Another example could be **LIO Robot**, developed by F&P Personal Robotics in 2018, see (19) . It is fully autonomous mobile robot with a multi-functional arm explicitly designed for human-robot interaction and personal care assistant tasks. It was intended, at first, for provide support and assistance to employees in farms but its potential is extendable to other type of final users (i.e elderly people). An example of assistive robot could be **TEK RMD**, developed by MATIAS Robotics in 2020, see (20). Equipped with GPS is able to reach source signal, TEK RMD is thought to enable people in wheelchair to goes around in the environment. It is ideal for paraplegics or those with compromised walking and standing abilities but as cons it can be used only from people that have good upper limb and hand functions. Then it is not ideal for cSCI subjects. The last example that I want to cover is **HAL5 - Hybrid Assistive Limb**. Experts in computer science,robotics mechanics and biomedical have developed exosuit for different purposes. HAL is a powered exoskeleton suit developed by Japan's Tsukuba University and the robotics company Cyberdyne that exploits two important functionalities, Cybernetic Voluntary Control (CVC) & Cybernetic Autonomous Control (CAC), allowing people with motor impairments to walk.

Keeping in mind that teleoperation will be carried out by cSCI people, a lot of the solution above listed shall be discarded as a result of cSCI's motor functions. It is needed an interface that could be accessible to every type of disable people, also from people with great motor disabilities. In the last years a new ad hoc interface for people with impaired mobility has been designed. BoMI "Body Machine Interface" convert high-dimensional body signal into lower-dimensional commands, allowing the user to control an external device trough IMU sensors (1), such as mouse cursor, an electric wheelchair (4) or an assistive manipulator. It gives BoMI great flexibility and the possibility to adapt it at every type of cSCI subjects, selecting the most suitable body joints. Recent studies, see (3), prove that integrating BoMI with physical therapy can be a way to remapping motor skills and also a way to stimulate psychologically the subjects to do better.

(a) *El-E Robot*

(b) *LIO Robot*

(c) *TEK RMD*

(d) *HAL5*

Figure 1.1: Some existing Rehabilitative and Supportive Robots

Chapter 2

Body Machine Interface

Summary

As previously stated, BoMIs enable individuals with restricted mobility to extend their capabilities by mapping their residual body movements into commands to control an external device. During this work you will see the application of two type of BoMIs in order to teleoperate TIAGo robot. The first one -*see* chapter 2.2 - is a video-based marker-less interface that can track the position of different parts of the body (e.g nose, shoulder, eyes and fingers) using only a camera. The second one - *see* chapter 2.3 - is based on IMU sensors placed on different parts of cSCI's body in order to acquire motion data.

2.1 Dimensionality Reduction Techniques

Real-world data, such as speech signals, digital photographs or in this case data acquired from body motion usually has high dimensionality. In order to handle such real-world data, its dimensionality needs to be reduced. Several dimensionality reduction techniques (**DR**) have been studied and developed in the last decades by machine learning researchers.

2.1.1 Principal Component Analysis

The most classical one is Principal Component Analysis (**PCA**). PCA is a linear technique for DR, which means that it performs DR by embedding the data into a linear subspace of lower dimensionality. In data analysis, the first principal component of a set of **n** variables, presumed to be jointly normally distributed, is the derived variable formed as a linear combination of the original variables that explains the most variance. The second principal component explains the

most variance in what is left once the effect of the first component is removed, and we may proceed through **n** iterations until all the variance is explained. In other words given a dataset X of n observations of dimensionality D, we want to find a matrix M that map X into a lower dimensionality d, such that $d < D$. In mathematical terms, PCA attempts to find a linear mapping M that maximizes the cost function trace $M^T cov(X)M$, where $cov(X)$ is the sample covariance matrix of the data. This linear mapping is formed by the d principal eigenvectors (i.e., principal components) of the sample covariance matrix of the zero-mean data. The eigen problem is the following:

$$cov(X)M = \lambda M \tag{2.1}$$

The eigenproblem is solved for the d principal eigenvalues λ. Finally the low-dimensional data representations y_i of the datapoints x_i are computed by mapping them onto the linear basis M,such that:

$$Y = XM \tag{2.2}$$

2.1.2 Autoencoder

Autoencoders are a family of neural network encoding a lower dimensional representation of the input space. The wanted behaviour is the following: train an artificial neural network model in order to obtain at the output units a pattern of activities as close as possible to the pattern on the input units. The general idea of autoencoders is pretty simple and consists in setting an encoder and a decoder as neural networks and to learn the best encoding-decoding scheme using an iterative optimisation process. After the creation of the architecture - *see* article (17) for more details, the autoencoder is then trained in order to find the weights that minimize the cost function E, defined as the mean square error between the input data and the reconstructed data in the output units:

$$E = \sum_{i=1}^{N} \|y_i - x_i\|^2 \tag{2.3}$$

where N is the number of neurons in each layer, x_i is the input of the first layer and y_i is the output of the last layer. Autoencoders with linear activation function perform the same operation as PCA, but of course it can be also non-linear.

2.1.3 Variational Autoencoder

Standard autoencoders learn to generate compact representations and reconstruct their inputs well, but asides from a few applications like denoising autoencoders,

they are fairly limited. The fundamental problem with autoencoders, for generation, is that the latent space they convert their inputs to and where their encoded vectors lie, may not be continuous, or allow easy interpolation. Variational Autoencoder (VAE) has one fundamentally unique property that separates them from autoencoders, and it is this property that makes them so useful for generative modeling: their latent spaces are, by design, continuous, allowing easy random sampling and interpolation. It achieves this by making its encoder not output an encoding vector of size n, rather, outputting two vectors of size n: a vector of means, μ, and another vector of standard deviations, σ.

2.2 Marker-Less BoMI

In the last years, several studies were carried out with regard to Computer Vision and markerless systems able to track body motion (16). These type of systems avoid the problem that a cSCI subject is not able to autonomously wear sensors. Therefore video-based markerless BoMI are potentially suitable for subjects with restricted mobility, since they allow a more natural user-friendly interaction with an external device and they are less invasive and cheaper than a sensor-based one. As a cons have emerged that:

- model training is sensible to the light of the environment
- equal colors of the surrounding environment and of the subject's clothing can bring to bad results in calibration

In particular I want to consider an already existent video-based markerless BoMI developed recently by a Team of Università degli Studi di Genova, see (13). For semplicity this program will be called later as 'Main Program'. The idea is simple: control a mouse cursor on the screen through body motion. Taking in mind that the application is oriented towards cSCI subjects, the body parts to track are the ones whose mobility is most likely retained even after a high level cSCI - *i.e* shoulders and head (nose and eyes) and some cases fingers. The architecture has to provide the following functionalities:

1. automatic acquisition of images of the user from a computer webcam
2. detection of landmark points - *e.g* eyes, nose, shoulders and fingers
3. encoding of the extracted signals to a lower dimensionality (control) space via application of DR techniques
4. handling of a graphical user interface (GUI) for providing BoMI users with visual feedback of the cursor via a computer monitor.

In particular the automatic body markers acquisition is based on a pre-trained machine learning model developed by Google, **MediaPipe** - *see* (15) - for further informations. You can download and install into your computer the markerless BoMI from (14), in order to try and see how the initial version works and the relative GUI. On top of this markerless BoMI will be implemented all the modules for teleoperating TIAGo, see chapters 4 and 5 . A pretty nice and useful tool that can be added on top of the original version is the webcam real-time feedback. In this way the user has the possibility to check if the camera acquisition setup is correctly tuned - *i.e* environment light and video acquisition area.

2.2.1 Mathematical Explanation

The first phase consists of a body dance (60 seconds) where user motion is recorded. After the first calibration phase, where MediaPipe model acquire autonomously data from joints selected, the BoMI map has to be constructed. Considering what has been said in section 2.1, acquired data from MediaPipe have dimension that is equal to the number of selected joints n, multiplied for two (position x,y of the joints in the image plane). Then if the user want to control the cursor into 2D plane, the initial $2 * n$ dimensional body vector have to be reduced to two dimensional control vector, (in equation: 2.1 where $\lambda = 2$, first to eigenvectors of the covariance matrix), since we want to control the position x&y of the cursor into 2D plane. Then for example, if we consider to select eyes and shoulder (number of joints equal to four) we have $2*4 = 8$-dimensional body vector.After computing matrix M with one of the DR techniques in section 2.1 (PCA/AE/VAE), the general final control vector is obtained by the following equation:

$$p^{(i)} = \begin{bmatrix} m_{1,1} & \dots & m_{1,n} \\ m_{2,1} & \dots & m_{2,n} \\ \vdots & \ddots & \vdots \\ m_{i,1} & \dots & m_{i,n} \end{bmatrix} \cdot q^{(n)} = M \cdot q^{(n)} \tag{2.4}$$

where $q^{(n)}$ in our example is the 8-th dimensional body vector and $p^{(i)}$ is the extracted 2-th dimensional control vector. After the calibration and map construction phases, the user has the possibility to customize the control vector tuning some important parameters as:

- horizontal and vertical offset p_0, if the position of the cursor is not satisfactory in resting position
- gain α along x and y directions to modify the amplitude of the control signals

- rotation matrix R in order to rotate the matrix M

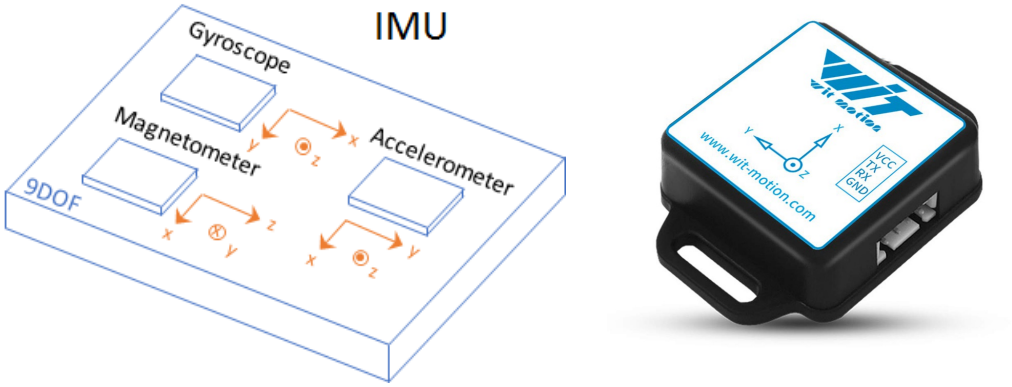
The final control vector customization can be seen in the following equation:

$$p^{(i)} = R \cdot \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_i \end{bmatrix} \cdot \begin{bmatrix} m_{1,1} & \dots & m_{1,8} \\ m_{2,1} & \dots & m_{2,8} \\ \vdots & \ddots & \vdots \\ m_{i,1} & \dots & m_{i,n} \end{bmatrix} \cdot q^{(n)} + \begin{bmatrix} p_{0,1} \\ p_{0,2} \\ \vdots \\ p_{0,i} \end{bmatrix} \tag{2.5}$$

The final phase consists of a series of reaching point task, in which the user can pratice BoMI and become familiar with it.

2.3 BoMI with IMU sensors

The idea and the structure behind is the same for the video-based one with the only difference that the acquisition of body signals are carried out by Inertial Measurements Units. Typically IMU sensors are placed on shoulders and forearms (the ones whose mobility is most likely retained even after a high level cSCI). Each IMU, combining the information of a triaxial gyroscope, accelerometer and compass sensors embedded in the IMU, provide in real time signals that can be sampled. Data can be processed as Euler Angles (yaw, pitch and roll) or as quaternions. Both of them provide a convenient mathematical notation for representing spatial orientations and rotations of elements in three dimensional space. It is well known in literature that usually yaw angle is difficult to be processed, for this reason it is disregarded. Therefore each IMU provide a 2-dimensional signal vector (roll and pitch). In order to compute the control vector the following steps are equal to the video-based one, explained in chapter 2.2



(a) *IMU embedded sensors* (b) *IMU hardware*

Figure 2.1: Inertial Measurement Unit sensor

Chapter 3

TIAGo Robot

Summary

This chapter has the scope to give a general review on TIAGo Robot specifications and functionalities. Moreover will be explained some teleoperations techniques that will be implemented in the final software architecture.

3.1 General Description

Developed by PAL-Robotics, TIAGo is a mobile robot used for assistive and supporting tasks. Recently Università degli Studi di Genova, purchased TIAGo ++ version, the one with two arm in the Figure 3.1. The price not relatively high makes it suitable for commercial applications.

3.2 Use cases

TIAGo demonstrates its potential during COVID-19 emergency, alongside cleaning and sanitation staff in hospitals, see (21) TIAGo is also used as social robot in the project CARESSER for social assistance to elderly people, see (22).

3.3 Technical Specifications:

3.3.1 General Features:

- Height: 110 - 145 cm
- Weight: 70kg
- Footprint: Ø54 cm
- Battery Autonomy: 4-5 hrs

Figure 3.1: TIAGo++ PAL-Robotics

3.3.2 DEGREES OF FREEDOM (DoF)

- Torso Lift: 1 prismatic joint
- Mobile Base: 2
- Arm: 7
- Head: 2

3.3.3 Audio:

- 2 x 5 W audio speaker
- 2x microphone array with stereo output 50-8000Hz

3.3.4 Sensors

- Base: laser 5.6 m / 10 m / 25 m range, rear sonars 3x1m range
- IMU (Base): 6 DoF
- Motors: Actuators current feedback
- Head: RGB-D camera

3.3.5 Computer:

- CPU:Intel i5
- RAM: 8GB
- SSD: 250GB

3.3.6 Operating System:

- Ubuntu LTS 64-bit
- RT Preempt real-time framework

3.3.7 Robotic Middleware

- Orocos
- ROS LTS

3.3.8 Simulation:

- Gazebo dynamic simulation
- URDF model
- RViz

3.3.9 Teleoperation:

- base
- torso lifter
- head
- end-effector

3.3.10 Ros Controllers:

Controllers implemented as ros control plugins running in the real-time control loop. Supported control modes:

- Wheels: velocity control
- Lifting torso and head: position control
- Arm motors: position and effort mode
- Joint trajectory controllers on groups of joints
- QT GUI to move individual joints
- Head Action Server to control the robot's gaze

3.3.11 Autonomous Navigation:

- Laser-based mapping and self-localization
- Navigation to a map point
- Obstacle avoidance

3.3.12 Robot Sensor Visualization:

- Rviz plugins for camera
- lasers
- sonars
- IMU
- force/torque sensor

3.3.13 Whole body control:

Hierarchical quadratic solver providing:

- On-line inverse kinematics of the robot's upper body (7 DoF arm, torso prismatic joint, 2 DoF head)
- Self-collision avoidance
- Joint limit avoidance
- Gaze control

Chapter 4

Software Architecture - Base Teleoperation

Summary

As previously explained in Chapter 2, the choice of implementing BoMI as a way to teleoperate TIAGo is not only due to its effectiveness with cSCI subject but also because it could be seen as a source of distraction. In this chapter you will see the implementation of the overall architecture for the mobile base, starting from the graphical user interfaces, designed to help the user to teleoperate TIAGo and to represent useful information, going to explain the Finite State Machine developed, arriving to TIAGo's control system implemented.

4.1 General Description

Having understood how well BoMI fits with cSCI subjects, it is now time to deal the most important part, how to process body signals in order to teleoperate TIAGo robot, going into technical details. You have to imagine the architecture composed principally by two cores. The first runs into a machine that we will call the main program, responsible to acquire body signals and convert it into control signals as explained in section 2.2.1. A second computer is connected to the first one, thanks to a socket TCP/IP communication. In this second computer, operative system Ubuntu is mandatory, runs all the components and modules needed to teleoperate TIAGo , both in simulation both real hardware.

4.2 Main Program

Starting from the Marker Less BoMI version described in section 2.2, additional state has been added in order to teleoperate TIAGo.

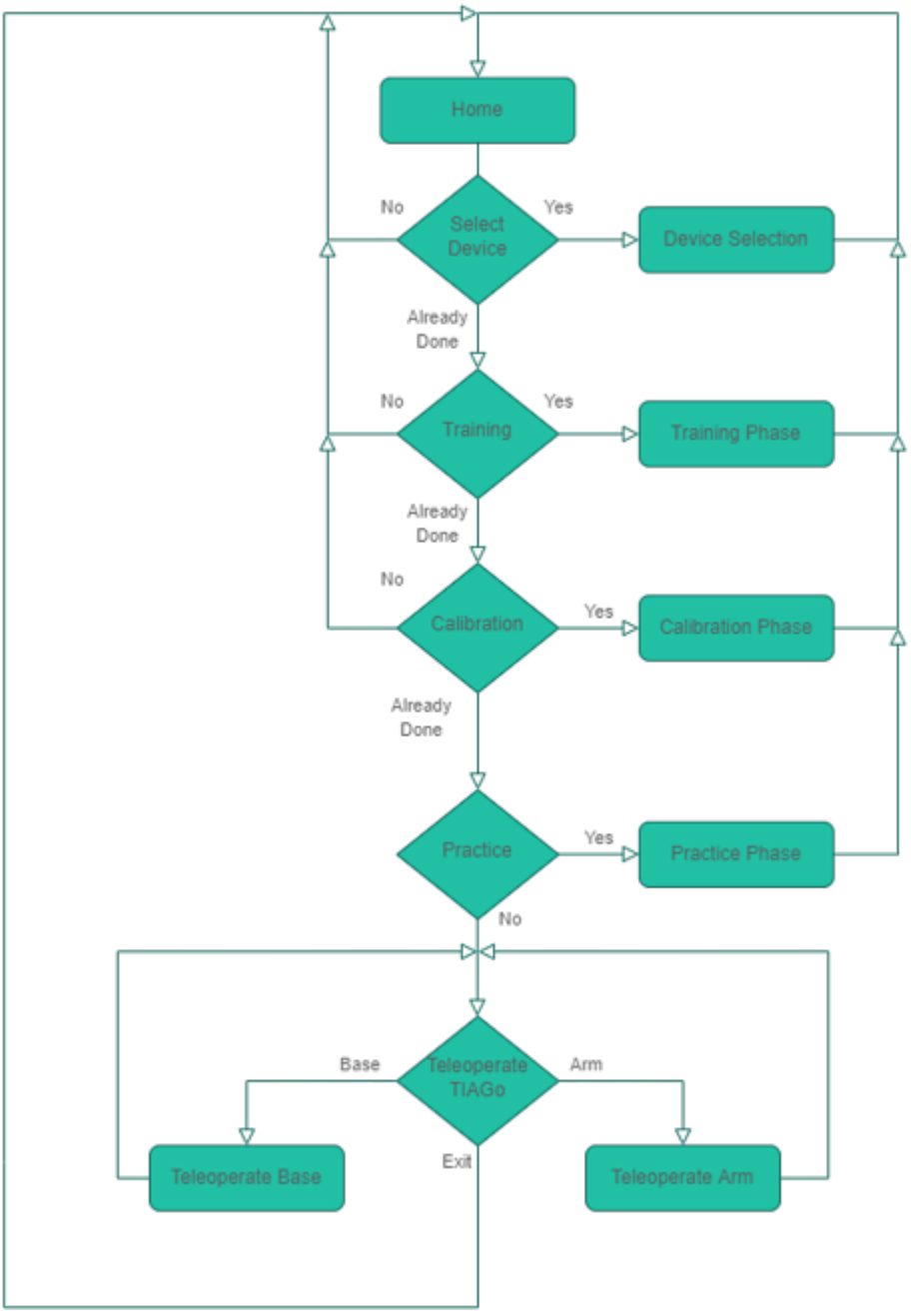


Figure 4.1: State Diagram of the Main Program

In particular, as you can see in the following state diagram, has been added to the original version of the MArker-Less BoMI the state demanded to teleoperate the mobile base and the state demanded to teleoperate the arm.

4.3 Twist message composition

In order to control differential drive mobile base a Twist message has to be send in real time. Since TIAGo is not-holonomic it cannot translate across the direction of the wheels. In particular a Twist message is the composition of linear and angular velocity. The ROS topic demanded to publish the message is :/mobile_base_controller/cmd_vel. For example, the message to have TIAGo moving at 0.5 m/s forward and at the same time rotating at 0.2 rad/s about its vertical axis can be seen in the figure 4.2.

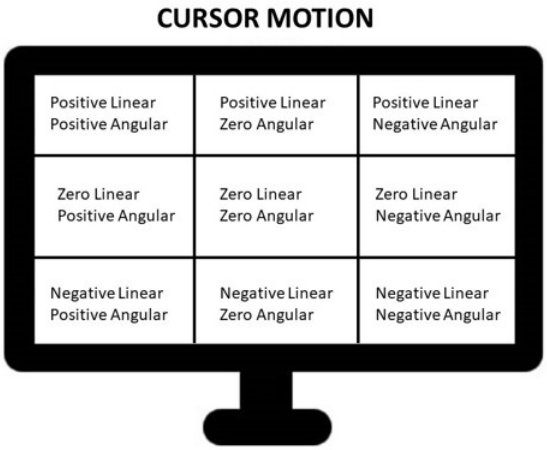
```
rostopic pub /mobile_base_controller/cmd_vel \
geometry_msgs/Twist "linear:
  x: 0.5
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.2" -r 3
```

Figure 4.2: Twist Message

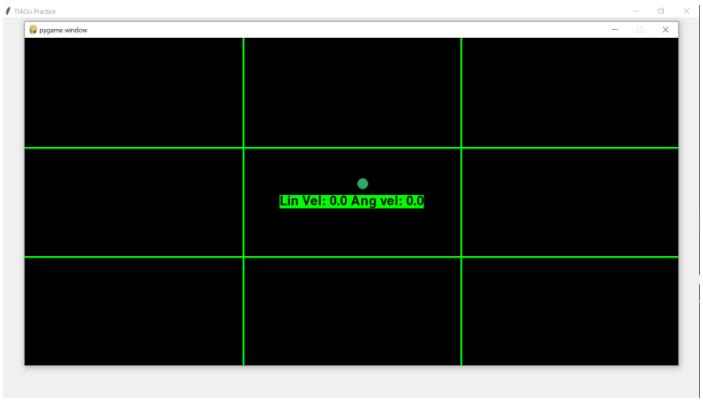
Only after that the user has maintained the cursor fixed on a specific region for some instants (i.e 500ms should be enough) the associated Twist message is sent. During transient periods no message will be send, in order to prevent dispatching wrong messages. Before the submission of the Twist message a mapping is needed. In order to makes possible the teleoperation also indoor, the values chosen are: ± 0.2 m/s for linear velocity and ± 0.5 rad/s for angular velocity.

4.4 Nine Regions GUI

The idea is to modify the original video-based markerless BoMI version presented in chapter 2.2 in order to control the mobile base of TIAGo. You have to imagine



(a) *Nine Regions GUI*



(b) *Nine Regions GUI drew with Pygame*

Figure 4.3: Nine Regions GUI

that all this part is contained inside the 'Teleoperate Base' state, see figure 4.1 First of all the screen will be divided in nine areas. Each area corresponds to a different composition of the Twist message, as you can easily understand seeing Figure:4.3. In this way the user is able to differentiate the areas of the screen. As a result the subject should be able to intuitively teleoperate TIAGo moving the cursor into the appropriate area. The final behaviour will be:

- Maintaining the cursor on the top left corner, positive linear and positive angular velocities will teleoperate TIAGo left and forward
- Maintaining the cursor on the top central region, positive linear and zero angular velocities will teleoperate TIAGo only forward

- Maintaining the cursor on the top left corner, positive linear and negative angular velocities will teleoperate TIAGo right and forward
- Maintaining the cursor on the middle left region, zero linear and positive angular velocities will teleoperate TIAGo only left
- Maintaining the cursor on the middle central region, zero linear and zero angular velocities will maintain TIAGo still.
- Maintaining the cursor on the middle right region, zero linear and negative angular velocities will teleoperate TIAGo only right.
- Maintaining the cursor on the bottom left region, negative linear and positive angular velocities will teleoperate TIAGo left and back
- Maintaining the cursor on the bottom central region, negative linear and zero angular velocities will teleoperate TIAGo only back
- Maintaining the cursor on the bottom right region, negative linear and negative angular will teleoperate TIAGo back and right.

4.5 Odom GUI

In section 4.4 you have seen the first GUI to compose the twist message in order to do a manual teleoperation of TIAGo's mobile base. Since maintaining the same body position for a long time could be a problem for people with motor impairments, a second ad hoc GUI is implemented. Thanks to a simple Cartesian 2D plane the user has the possibility to select a target on the 2D screen (x and y coordinates) that represent the new goal that TIAGo will has to reach. In order to cover all possibles map coordinates the final target position will be:

$$FinalTargetPosition = TIAGoPosition + SelectedCoordinate \tag{4.1}$$

How to selected the coordinates? How to switch between Nine Regions GUI and Odom GUI? This will be deal in the next section.

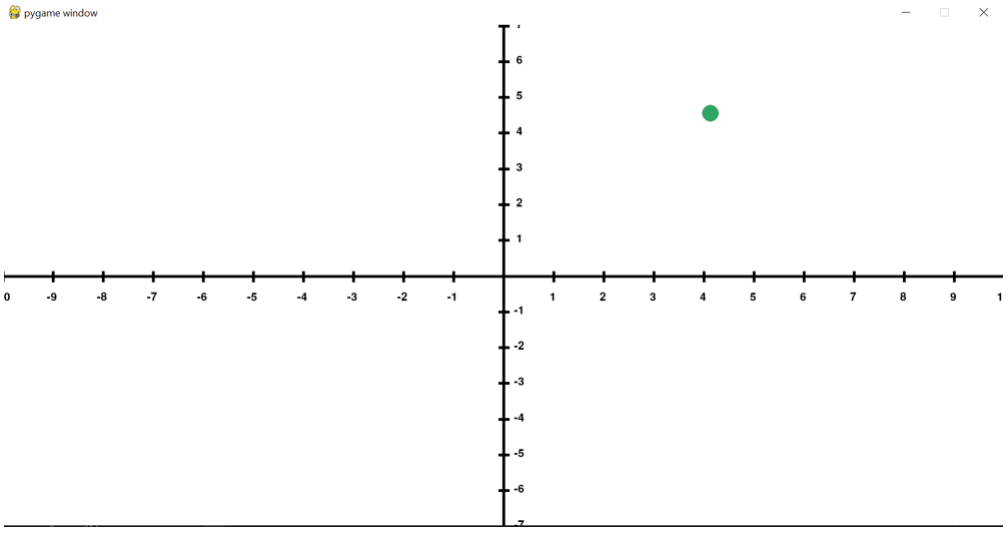


Figure 4.4: Odom GUI drew with Pygame

4.6 Mobile Base - ROS Side

In this section we will talk about **ROS** (Robot Operating System) modules needed to control TIAGo's mobile base. As it is usual to do at the design stage, it is important to test the behaviour of the architecture in simulation, before testing into the real robot. ROS have two important framework that usually are used:

1. **Gazebo** - It is demanded to import the URDF (Unified Robot Description Format), an XML file for representing the robot model (links, joint, motors, sensors and controllers). Gazebo spawns the robot on a specific environment that is usually specified in a .world file. Gazebo has the scope to simulate the physics and the dynamics of the simulation

2. **RViz** - It is a ROS graphical interface that allows you to visualize a lot of information, using plugins for many kinds of available topics. For example it is possible to import the robot model, it is possible to visualize the map that the robot is creating thanks to LIDAR and SONAR sensors (if any), it is possible to see trough the robot camera (if any) ecc..

We have seen in the previous sections how the GUIs for the mobile base works. Is now time to explain how these data are used to control TIAGo ROS side. Referring to the 'Nine Regions GUI' for a manual teleoperation, will be sufficient to publish a Twist message on a topic. The topic demanded to acquire twist message to move TIAGo in the environment is **/mobile_base_controller/cmd_vel**,

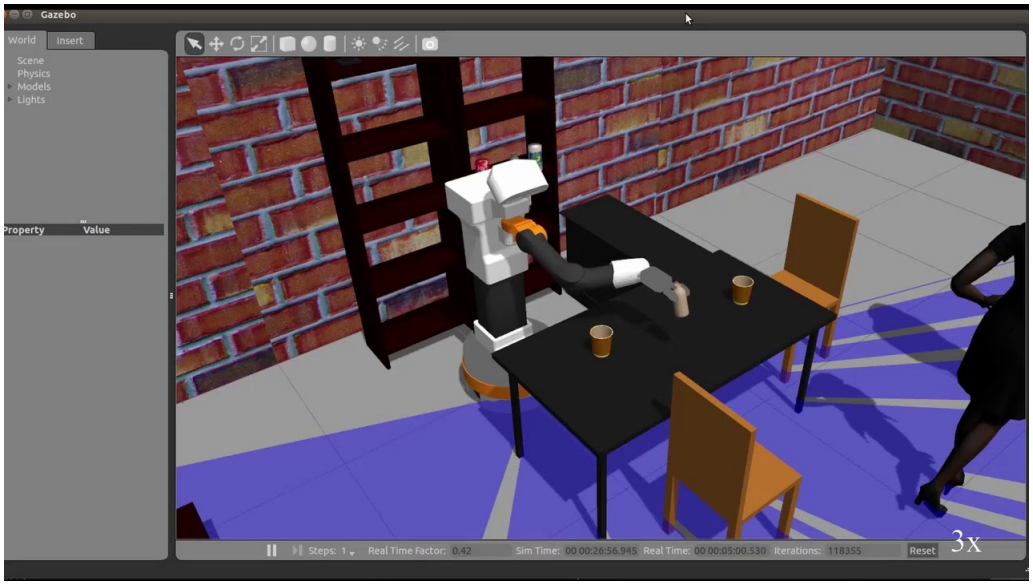


Figure 4.5: TIAGo in Gazebo Simulation Environment

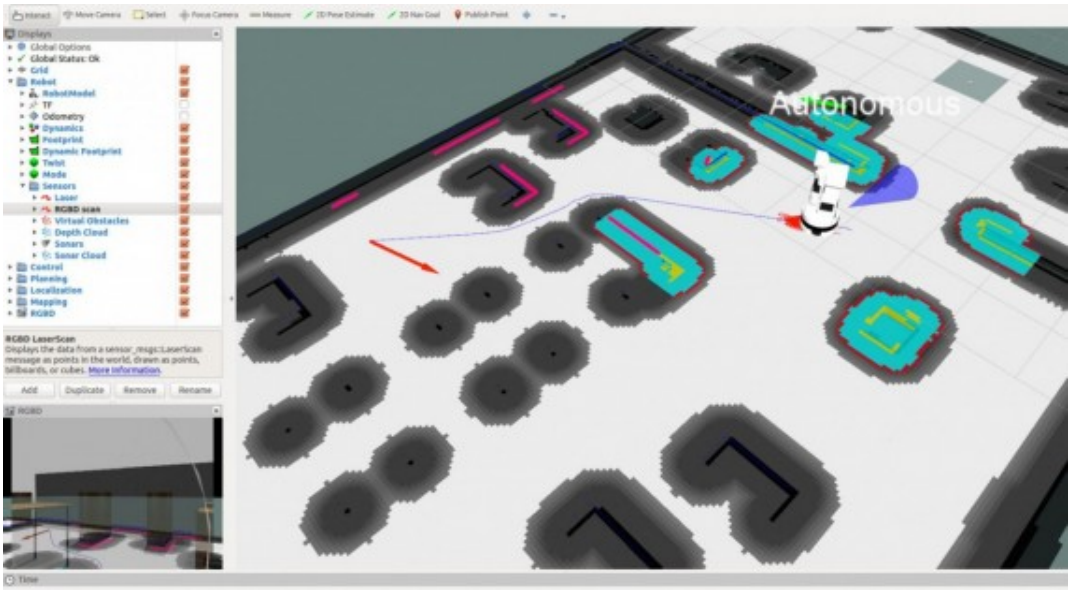


Figure 4.6: TIAGo in Rviz Simulation Environment

see figure: 4.2. After that the twist message is composed on the main program, depending on the position of the cursor in the 'Nine Regions GUI', a publisher on topic cmd_vel will move TIAGo. Talking about 'Odom GUI', for an autonomously teleoperation, things get a little more complicated. Before entering in detail I have to explain to you the concept of **mapping**, how robot are able to localize itself in

the environment and **autonomous navigation**. In general in ROS, in order to create a map in the environment it is used GMAPPING. This package contains a ROS wrapper for OpenSlam's Gmapping. The gmapping package provides laser-based SLAM (Simultaneous Localization and Mapping), as a ROS node called slam_gmapping. Using slam_gmapping, you can create a 2-D occupancy grid map (like a building floorplan) from laser and pose data collected by a mobile robot. In order to use slam_gmapping you need a mobile robot that provides odometry data and is equipped with an horizontally-mounted, fixed, laser range-finder. Once the map is created the robot will be able to localize itself in the environment, it will be the starting point for the autonomous navigation. Usually in order to allow a robot to move it self autonomously in the environment in ROS it uses move_base package. The move_base package provides an implementation of an action that, given a goal in the world, will attempt to reach it with a mobile base. The move_base node links together a global and local planner to accomplish its global navigation task. As you can see in the following figure 4.7, (taken from Move Base Documentation), it is mandatory for move_base to have available:

- a map published on topic /map of type nav_msgs/GetMap
- odometry source of type nav_msgs/Odometry
- a sensor source of type sensor_msgs/LaserScan
- a sensor transform of type tf/tfMessage

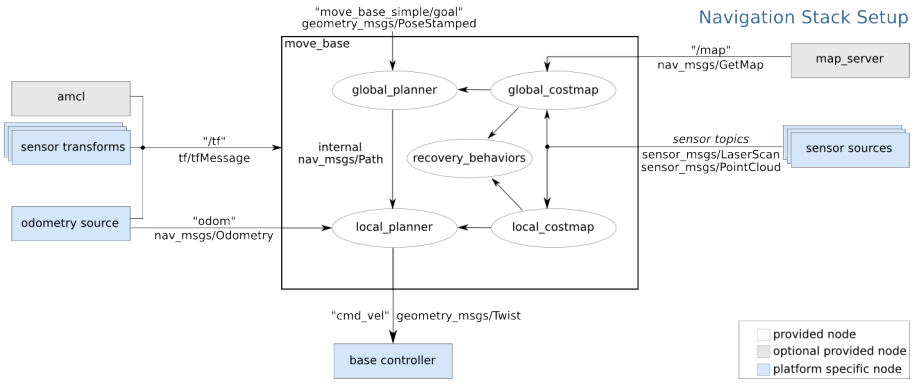


Figure 4.7: How Move Base internally works

If the packages slam_gmapping and move_base work well, publishing a goal on the topic **move_base_simple/goal** you will see the robot moves trying to reach the goal, avoiding obstacles, in a completely autonomous way. This is possible thanks to path planning algorithm such as Dijkstra or A*. For example using

Dijkstra it is possible to compute the minimum path, from a node to another, in a graph with or without sorting, in this context from the robot to the goal. Once a path is compute move_base will guide the robot through Twist messages.

4.7 Eye Blinking Detector

As we have discussed cSCI subject maintains a limited motion of the upper body depending on the high of the lesion's level. For this reason different subjects can have very different residual motor skills. In order to make available TIAGo teleoperation to every type of subjects, it was decided to implement a script that detects the eyes blinking or closure in order to allow every subject to switch the state of the finite state machine. People that have lost hands or arms motor skills are able to control the FSM thanks to their eyes in a fully autonomous way. The script was built on the basis of a model developed by Google, MediaPipe offers open source cross-platform, customizable ML solutions for live and streaming media. Among the many functionalities it offers (object detection, hair segmentation,body pose and holistic, etc), there is the possibility to detect an human face. Of course in order to acquire the information it is mandatory the use of a webcam. In particular it is able to create a graph of the face that you can use for your purposes. Each node of the graph has its own ID that will be used to track the wanted region of the face, in this case the eyes.

Eye Blinking Detector script operation is rather simple and intuitive. Once Mediapipe has extracted the face graph, that you can see in figure 4.8, it computes the distance between upper and lower eyelid. This distance is the euclidean distance from point 145 to point 159 in figure: 4.9, that for simplicity is called σ.

This distance will establish the state of the eye: open or closed on the basis of a threshold. In order to compute this threshold a preliminary calibration phase is mandatory. In particular this phase has the duration of 10 seconds where the subject has to open and squint the eyes. All σ values, frame by frame, are stored into a list. When the calibration phase is finished $\sigma_{threshold}$ for each eye is computed in the following way:

$$range = max_{\sigma} - min_{\sigma} \tag{4.2}$$

$$\sigma_{threshold} = min_{\sigma} + range * 0.3 \tag{4.3}$$

Simply for each eye is computed the range equation 4.2, then 30% of the range is added to the minimum as in equation 4.3. In this way if σ is minor than $\sigma_{threshold}$ the eye is considered closed. Thanks to appropriate stop watch the script meets the following functionalities:

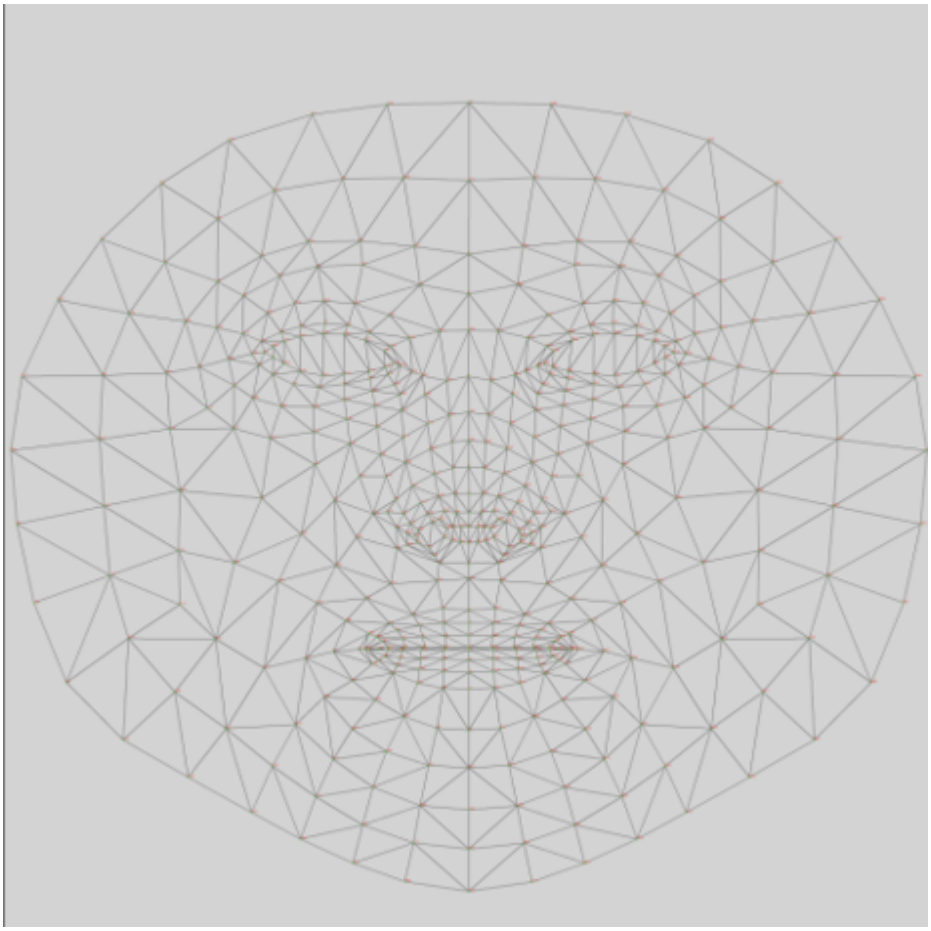


Figure 4.8: Extracted Face Graph with Mediapipe

1. Detect three eyes closure in 1.5 second
2. Detect eyes closure for 1.5 second duration
3. Detect winking

How these functionalities are used will be explained later.

4.8 Nose Detector

Taking in mind what previously sad about Eye Blinking detector a node has been implemented in order to track nose tip and use them as a trigger to change the state of the FSM. Referring to Fig:4.8 you can see that the nose tip has the ID 94. Via a preliminary calibration phase a threshold was computed with the

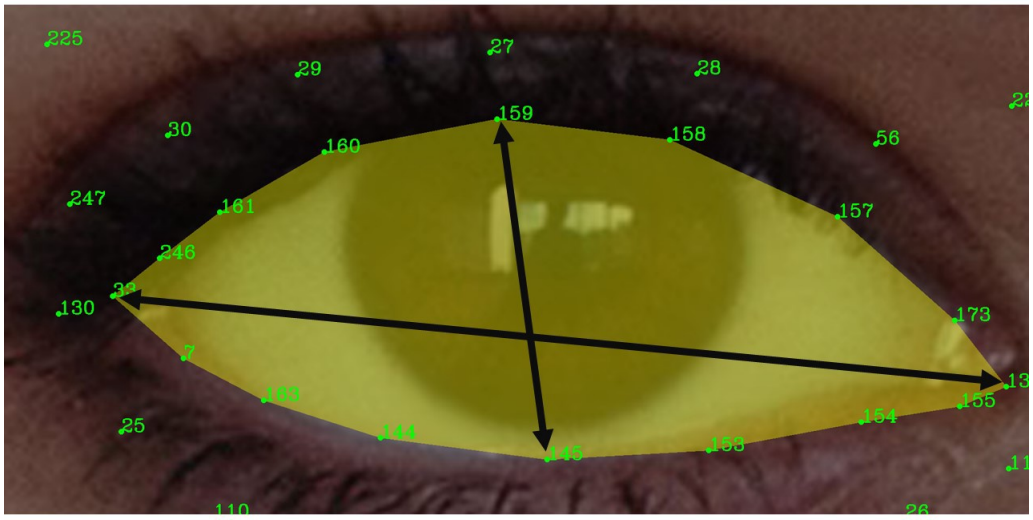


Figure 4.9: Eye Closure Distance

aim to detect the nose tip right cross this threshold. This threshold is computed similarly to Eq:4.2. In particular using Mediapipe and OpenCV the coordinates of the nose tip into 2D image plane are extracted, call it for simplicity $nose_x$ and $nose_y$. During the 10 seconds of the calibration the coordinates are stored into two list in order to calculate the threshold as follow:

$$nose\_range = max\_nose_x - min\_nose_x \tag{4.4}$$

The wanted trigger signal is following: move the nose tip two times in 2 seconds, in such a way to override on the right the threshold.

Taking this in mind the threshold is computed as follow:

$$nose\_threshold = min\_nose_x + nose\_range * 0.15 \tag{4.5}$$

As you can see the equations above consider only the x coordinate of the nose, indicating that only the horizontal motion of the nose is taken in mind. The purpose of this trigger will be treated later.

4.9 Socket TCP/IP Communication - A bridge between Main Program and ROS Side

A TCP socket is the means by which a destination packet is provided with all the information necessary to leave for the recipient host, as well as the main

responsible for establishing the connection between two hosts and maintaining the session and then regenerating the connection upon sending of additional packages. The IP in the name means that the connection between the client and the server is done thanks to the internet address of server and its port. Notice that is important to specify that the two process, client and server have to be under the same LAN. As you can see in the following figure the connection is established thanks to the following sequence:

1. Initialize client and server processes
2. The client process sends the connection request to the server indicating the socket
3. The server accepts the request and creates a virtual channel that will use for data transmission

The choice of implementing a socket communication between the main program and ROS side has many reasons:

- Establish a connection between two distinct machines, the one that runs the BoMI main program and the second machine running the simulation and all the nodes needed to teleoperate TIAGo.
- Do not stress too much the machine splitting data computations into two different machines, in case the subject wants to teleoperate TIAGo in a simulation environment as Gazebo.
- The subject has the possibility to deal with the interfaces into the main program and simultaneously see the simulation in Gazebo.

In this context it has been implemented a server socket in order to receive data into the machine demanded to control TIAGo robot and this server has to be executed from the ROS machine. Moreover a client has been implemented and has to be exectuted from the machine that runs BoMI main program. The server is executed at the start of the simulation and it listens until client socket requests the connection with the server. This events from the client occurs when the subject wants to start to teleoperate TIAGo robot, both in simulation both the real robot. If everything went well, client and server should be connected through the IP address and port specified by the server machine.

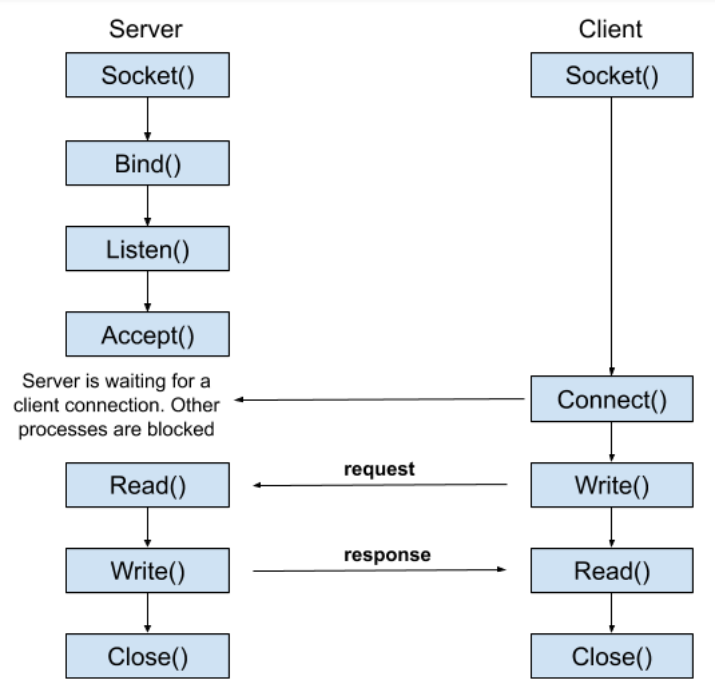


Figure 4.10: Socket Communication Sequence

4.10 Mobile Base Implementation

We have seen how the teleoperation of the base has been designed. The main program contains a dedicated mode for teleoperating only the base of TIAGo. In this way the subject has the opportunity to practice exclusively the teleoperation of the base. Taking in mind what has been sad before about socket TCP/IP communication, chapter 4.9, the subject has the possibility to use both the interfaces developed and explained in chapters 4.4 and 4.5. Comes into play here, the **three times eye blinking trigger** captured from the Eye Blinking Detector Thread. In particular it is used in order to change the interface wanted, switching between the 'Nine Regions GUI' and the 'Odom GUI'. The behaviour is clear looking the following simple state diagram.

Using his residual body motion, the subject will have the possibility to teleoperate manually the robot using the 'Nine Regions GUI', blinking three times the eyes he will have the possibility to teleoperate TIAGo in a partially autonomous way using the 'Odom GUI' and so on As you can see, if the user is teleoperating TIAGo with Odom GUI, closing eyes for 2 seconds he has the possibility to assign a target position, which TIAGo will reach autonomously. Should be now clear the overall architecture structure in order to teleoperate TIAGo base. The following diagram, developed with Diagrams.net, summarizes the main important

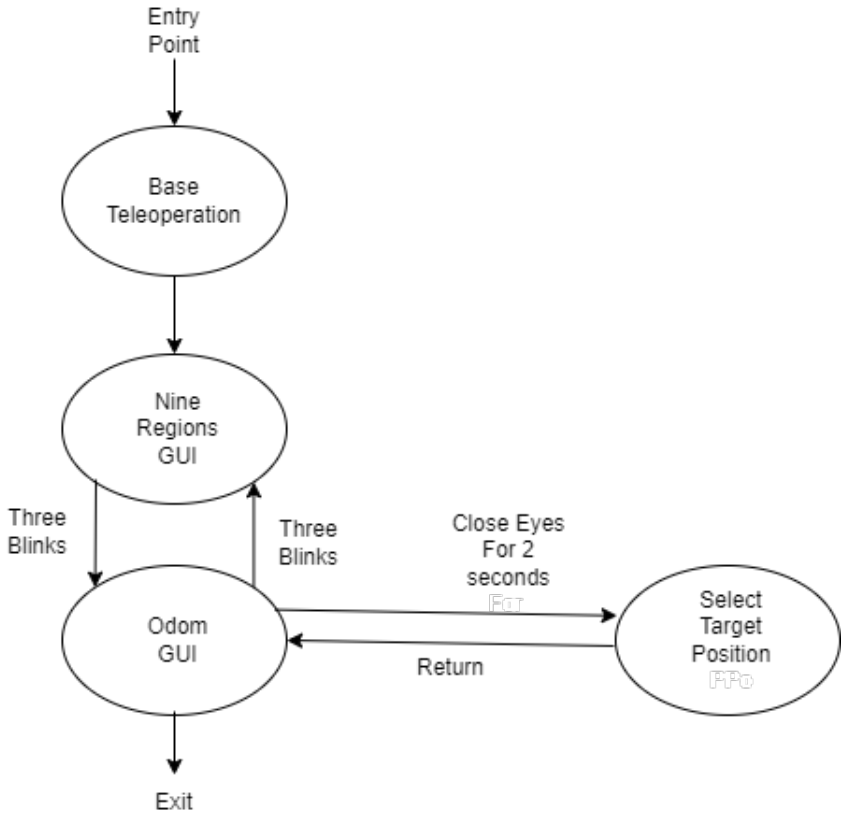


Figure 4.11: Base Teleoperation State Diagram

components of the base teleoperation architecture.

In the figure above you can see the three main components of the architecture, Into the **Blue** box in figure:4.12 there are all the components executed by Marker-Less BoMI main program. Temporally speaking, its pipeline flows is the following:

1. Frame acquisition from webcam
2. Body Landmark detection from Mediapipe in order to store body joints coordinates into the body vector q, (see section: 2.2.1)
3. Apply dimensionality reduction techniques (see chapter 2.1), thanks to BoMI map previously computed and customized, to body vector in order to extract the control vector p, in this case p has two dimensions.
4. The two extracted dimensions of p are used in order to control mouse cursor into 2D monitor plane.

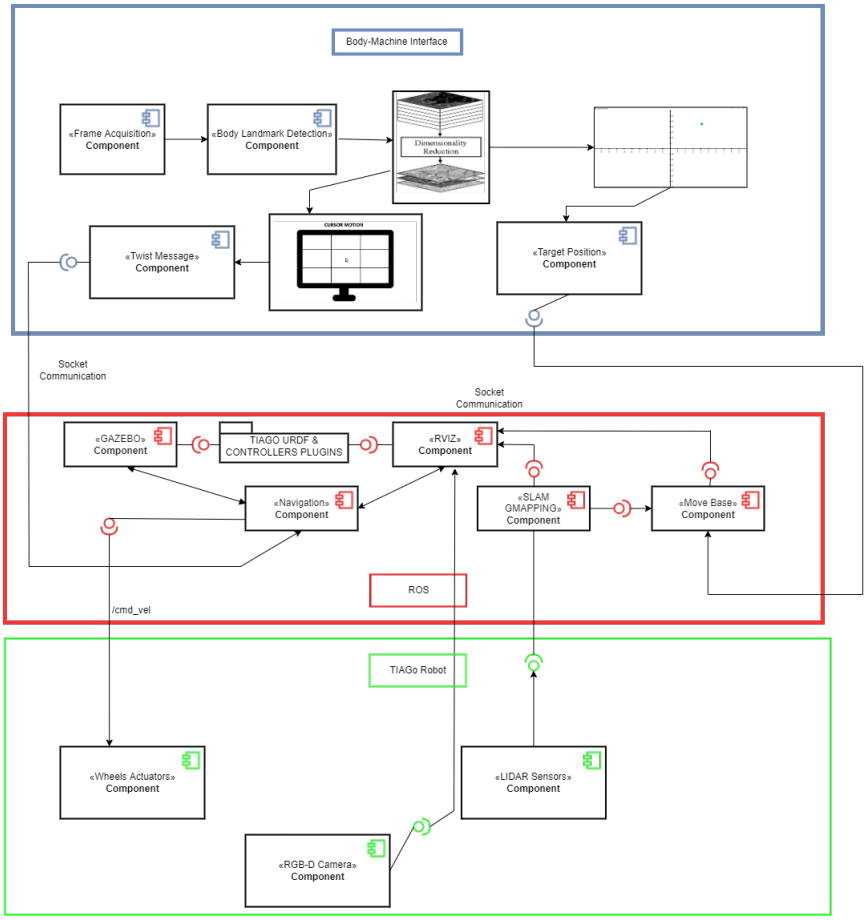


Figure 4.12: Base Overall Architecture

5. The subject has the possibility to teleoperate TIAGo via Twist message composition using 'Nine Regions GUI' or via move base algorithm sending a position goal using 'odom GUI'.

Into the **Red** box there are all the components executed by the second machine, which runs the simulation (see chapter 4.6) and the nodes demanded to control TIAGo ROS side.

- The node **Navigation** takes the twist message composed by 'Nine Regions GUI' via socket communication and directly publishes the velocity command on the topic **cmd_vel** in order to control TIAGo's wheels actuators.
- The node **Move Base** (see figure: 4.7 and related chapter), takes the goal x and y coordinates (the orientation is established by default) via socket communication and publishes it on the topic **move_base_simple/goal**. Thanks to the informations provided by LIDAR and sonar sensors, move base internal planner is able to plan a path, that will be respected sending **cmd_vel** to TIAGo's wheels actuators.

Finally the components inside the **Green** box are related to the physical modules present on TIAGo (wheels actuators, RGB-D camera and LIDAR sensors).

Chapter 5

Software Architecture - Arm Teleoperation

Summary

We have seen how to move TIAGo from the start location to the target location, how the navigation is implemented and how a tetraplegic subject has the possibility to teleoperate TIAGo's base. At this point the subject should have the possibility to reach a target object, teleoperating TIAGo thanks to the architecture explained in the earlier chapter. Once arrived in front of the object, how to teleoperate TIAGo's arm in order to reach the object with the end-effector, how to make teleoperation possible taking in mind that the subject is a tetraplegic person with limited body motion. You can find all the answers to questions soon in this chapter.

5.1 Explained Variance of the body vector q

Until now we have extracted a control vector p with n=2 dimensions from body vector with at least q=4 dimensions (shoulders positions) applying BoMI map, (see chapter: 2.2.1). This choice has been done on the assumption that TIAGo's mobile base has 2 DoF and thanks to the developed GUIs and control systems 2 variables are enough to teleoperate the base. Regarding arm teleoperation this choice remains valid? In order to control TIAGo arm there are two main solutions:

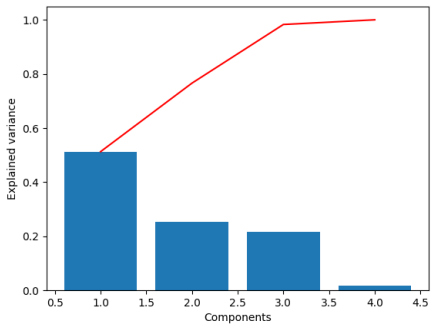
- Planning in **Cartesian Space**: specifying the end-effector position and orientation in a 3D space with respect to fixed frame, for example the base one.

- Planning in **Joint Space**: specifying the position of all 7 joints of TIAGo's arm plus the position of the prismatic torso one.

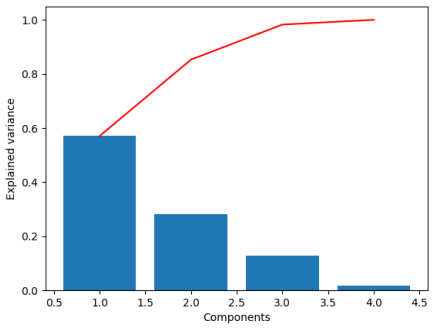
The second solution has to be discarded for the following reasons:

- Seven independent and controllable variables are too much for a tetraplegic subject.
- Assign a position for each joint should be very difficult and slow, it is better to demand this task to a planner.

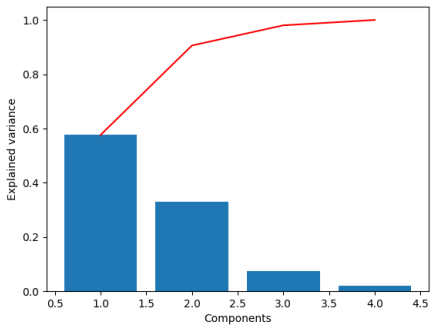
It is evident that the only accessible way is the **planning in Cartesian Space**. Since the arm workspace belongs to a 3D space the total amount of variable to be assigned is 6, three for the position and three for the orientation of the end-effector. Now if we assume that for the majority of objects (think for example to bottles and cups) the orientation of the e.e should be good if considered parallel to the ground, the total number of variables to be controlled decreases to three, that are the three coordinates of the end-effector's position in 3D space. The real question to ask yourself right now is: it is possible to extract a control vector p with i=3 dimensions? Theoretically, keeping the assumption that the subject can teleoperate TIAGo with even the movement of the shoulders, this is possible for healthy people and tetraplegic subjects with a lesion lower than level C2. For a tetraplegic person with a lesion at level C2 is not possible to extract three independent and controllable variables with the only shoulders motion but at most two. Although theoretically this is possible, performing several calibrations on healthy people, they showed that the explained variance along the third component is not enough to be controlled. As you can see in figure: 5.1, the variance along the third is too little to be controlled. The images in figure 5.1 shows how the variance is distributed on the components and they are labelled with the sum of the first two components. Only in the case VAF 74% the variance on the third component is sufficient to be controlled but this result was obtained only once in ten calibrations. The final solution, in order to meet the constraints of a person with a lesion at level C2 and the problem of the explained variance belongs to the third component, is to compose the information of three variables into two distinct GUIs and phases. In the next chapter we will see a solution to teleoperate TIAGo's arm.



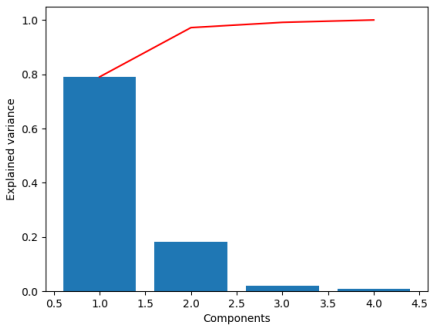
(a) *Explained Variance with VAF 74%*



(b) *Explained Variance with VAF 84%*



(c) *Explained Variance with VAF 90%*



(d) *Explained Variance with VAF 96%*

Figure 5.1: Explained variance plots of calibration with different VAF using only shoulders

5.2 MoveIt Motion Planning Framework

MoveIt is an easy-to-use open source robotics manipulation platform that performs some very interesting features. In particular it is able to:

- Generate high-degree of freedom trajectories through cluttered environments and avoid local minimums
- Analyze and interact with the environment with grasp generation
- Solve for joint positions for a given pose, even in over-actuated arms
- Execute time-parameterized joint trajectories to low level hardware controllers through common interfaces

- Connect to depth sensors and point clouds with Octomaps
- Avoid obstacles using geometric primitives, meshes, or point cloud data

These are some important features covered by MoveIt but other solutions are possible. In a general case MoveIt has the possibility to be configured in order to work with every type of robot. This is possible thanks to MoveIt setup assistant following step-by-step configuration wizard. In particular Moveit takes the robot URDF (Unified Robot Description Format), an XML format file which represents robot model (joints, links, motors, sensors,etc) all the components needed in order to simulate the robot physics and behaviour. MoveIt needs also a second file .gazebo that includes most of Gazebo-specific XML elements including the tags. MoveIt provide also a visual demonstration on Rviz with some nice features, thanks to Moveit motion planning:

- The user has the possibility to play with the robot seeing the possible configurations for the arm, simply dragging and dropping the end-effector
- Once a plan is computed, the user can see the precalculated path before apply it
- The user can assign a value for each joints and see the final result

In our context PAL Robotics provides us TIAGo URDF and MoveIt package already configured and ready to use. It is a good practice to group joints and links, in order to select the wanted group and planning only for the wanted joints and links. In our case study the group control the prismatic joint torso and the seven joints of the right arm is called **arm_right_torso**. Having made this brief introduction on MoveIt it is now time to deal with the arm teleoperation.

5.3 Arm Teleoperation with 2 DoF

Since this project want to build an architecture in order to teleoperate TIAGo's arm, it is important to understand how to acquire and handle data. In order to meet the constraints above listed the idea is to extract a control vector p with i=2, the first two components with the greatest variance, always using the BoMI Markerless version, acquiring shoulders position with Google Mediapipe, in order to teleoperate mouse cursor. The idea is to split the information from the control vector in two different states in order to control three DoF, the **position** in 3D space of the arm_tool_link frame with respect to the base_footprint frame, see figure: 5.3.

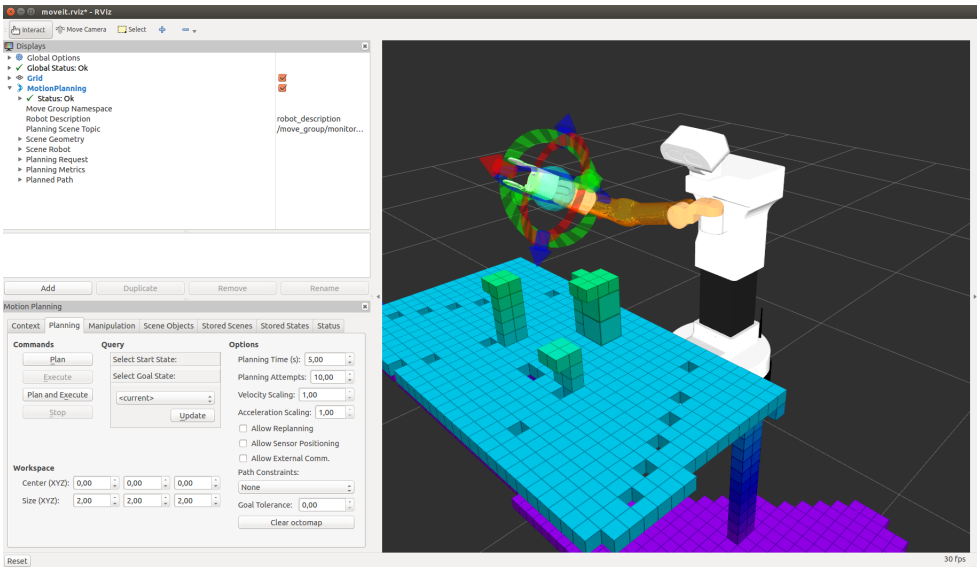


Figure 5.2: MoveIt Motion Planning in Rviz simulation Environment

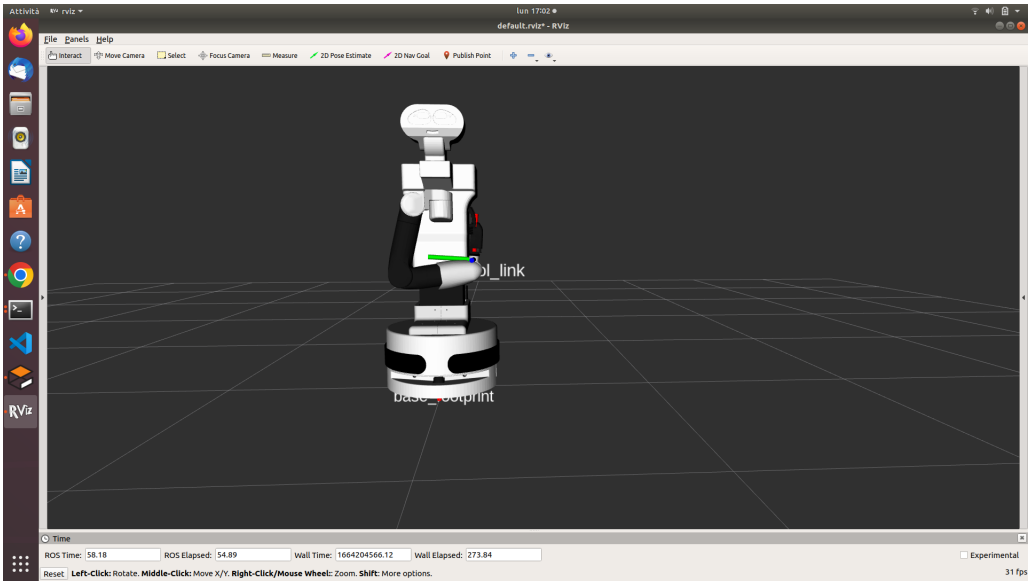


Figure 5.3: TIAGo arm_tool_link and base_footprint

One intuitive way to split the information is to control x and y coordinates of the arm_tool_link frame with respect to the base_footprint frame moving the cursor in a dedicated GUI called '2D vector GUI' and the z coordinate into another GUI called '1D vector GUI', always controlling the mouse cursor. The final result will be a new FSM node in order to teleoperate the arm, composed by two distinct states:

- Firstly the subject has the possibility to move the arm into a plane in the 3D space with a fixed altitude, changing only the x and y position coordinates.
- In order to select a vector the subject has to close eyes for 2 seconds
- Blinking three times the eyes the subject will have the possibility to change the altitude maintaining the same x and y gripper position coordinates, selecting 1D vector
- Take in mind the assumption done on the gripper's orientation.

Let's see now how the GUIs have been implemented.

5.4 2D vector GUI

Let's see now how to create a simple,intuitive and easy-to-use interface in order to change TIAGo's gripper position. In order to better understand how to teleoperate TIAGo's gripper, it is better to represent the translation and the gripper motion displaying a vector respect to a simple point. In fact vector brings with it the information on the magnitude and the direction, that visually help the user to understand how to move the body to teleoperate TIAGo's arm. A simple Cartesian plane with two axis will be sufficient for us. The subject moving his body will move mouse cursor into 2D screen plane. The concept is very similar to the 'Odom GUI' in figure: 4.4 but this time will be displayed the cursor that has the tip on the x and y cursor position and the tail in the origin of the Cartesian plane. This is simply a design choice in order to help visually the user how to teleoperate TIAGo's arm. In the following figure you can see the final result of the '2D vector GUI', developed with Pygame.

Data that will be processed in order to teleoperate TIAGo's arm are: the amplitude of the vector and angle that the vector span with the x axis counterclockwise. Using some basic geometry, calling the x position of the mouse as x_mouse and y position y_mouse and taking in mind that the screen resolution chosen is $(screen\_width X screen\_height) = (1800 X 900)$ vector amplitude can be computed via Pythagorean theorem:

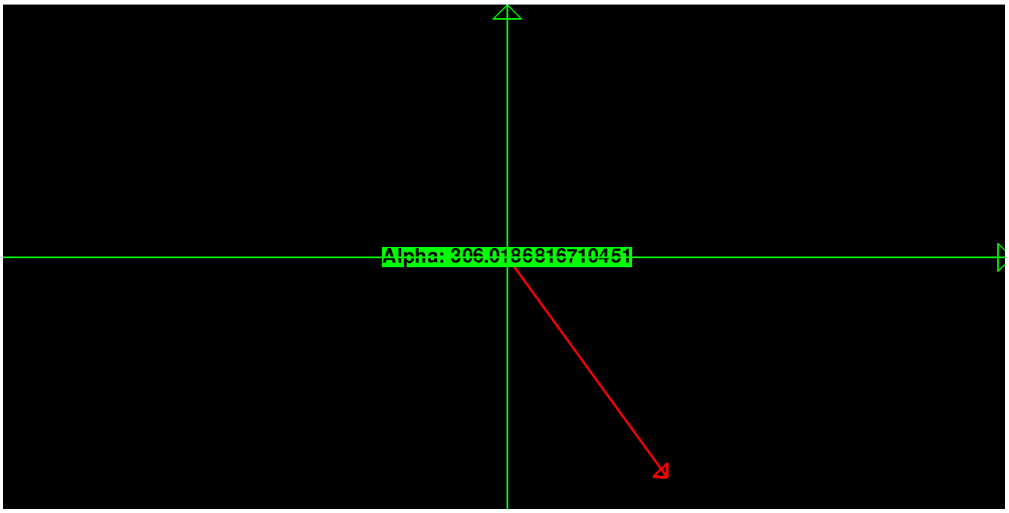


Figure 5.4: 2D vector GUI

$$vector\_amplitude = \sqrt{(screen\_center\_x - x\_mouse)^2 + (screen\_center\_y - y\_mouse)^2} \tag{5.1}$$

where screen_center_x and screen_center_y are respectively $screen\_width/2$ and $screen\_height/2$. Now that we have computed the hypotenuse of the right triangle spanned by the vector we can compute the internal angle of the triangle thanks to Cosine theorem for right triangle:

$$angle\_rad = \sin\left(triangle\_y\_component/vector\_amplitude\right) \tag{5.2}$$

where triangle_y_component depends on the quadrant where the vector has the tip, in particular it can be computed in the following python code:

```
1
2  #if the cursor tip is in the first quadrant
3  if (int(x_mouse) >= 900 and int(x_mouse) <= 1800) ...
4  ... and (int(y_mouse)>= 0 and int(y_mouse) <= 450):
5      triangle_y_component = screen_center_y - y_mouse
6
7  #if the cursor tip is in the second quadrant
8  elif (int(x_mouse) >= 0 and int(x_mouse) <= 900) ...
9  ... and (int(y_mouse)>= 0 and int(y_mouse) <= 450):
10     triangle_y_component = screen_center_y - y_mouse
11
12 #if the cursor tip is in the third quadrant
13 elif (int(x_mouse) >= 0 and int(x_mouse) <= 900) ...
14 ... and (int(y_mouse) > 450 and int(y_mouse) <= 900):
15     triangle_y_component = y_mouse - screen_center_y
```

```
16
17 #if the cursor tip is in the fourth quadrant
18 else:
19     triangle_y_component = y_mouse - screen_center_y
```

Listing 5.1: 2D Vector GUI

From equation 5.2, simply compute the angle in degree as:

$$angle\_degree = \frac{(angle\_rad * 180)}{\pi\_rad} \tag{5.3}$$

The information related to vector 's angle and amplitude will be sent via socket communication to ROS side.

In a scenario like in figure 5.4 how TIAGo's arm will move? Well at first sight, considering a reference system where TIAGo's head is placed on Cartesian origin with eyes facing the positive y axis, the arm will move far behind and little to the right. We will see later during this study how the information related to the amplitude and the vector angle will be dealt ROS side.

5.5 1D vector GUI

It is now time to see how to implement the interface to change TIAGo's arm altitude. This can be seen as a generalization of the previous case, where only the y position of the cursor is considered. In particular, always using the shoulder motion for changing cursor position, as in the previous case a vector is displayed which has the tip in the mouse coordinates and the tail in the axes' origin. This time mouse x coordinate is fixed to screen_center_x, the user will see a vector always with direction coincident with the y axis and verse that depends on the y mouse coordinate, see figure 5.5.

In this case the angle could be only $\frac{\pi}{2}$ or $\frac{3}{2}\pi$ depending on cursor y position, instead the amplitude can vary between 0 and screen_mouse_center_y:

```
1 if y_mouse < screen_mouse_center_y:
2     amplitude = screen_mouse_center_y - mouse_y
3     angle = pi
4 elif y_mouse > screen_mouse_center_y:
5     amplitude = mouse_y - screen_mouse_center_y
6     angle = 3/2pi
```

Listing 5.2: 1D Vector GUI

Also in this case vector's angle and amplitude will be sent via socket communication to ROS side. Selecting a vector as in figure 5.5 will drop TIAGo's arm.

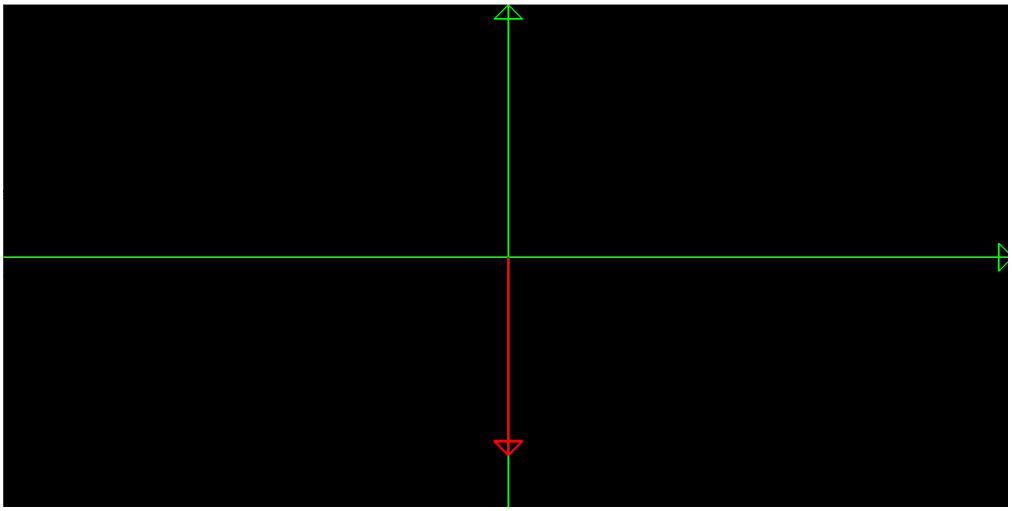


Figure 5.5: 1D vector GUI

5.6 Arm Teleoperation - ROS Side

We have seen the GUIs to help the subject to teleoperate visually TIAGo's arm. It is now time to see how the information related to vector's amplitude and angle are used ROS side to teleoperate TIAGo's arm. As anticipated in section 5.2, we will use MoveIt in order to plan path to move TIAGo's arm from initial position to end-effector goal position with respect to base_footprint frame. In order to do this, the first step is computing the available arm workspace, that is the space composed by all the available end-effector position in 3D space. Keep in mind that since the robot used in simulation environment is TIAGo one arm, instead in the real teleoperation the version is TIAGo++ with two arms, this step has to be repeated two times. This space could be computed using MoveIt motion planning and **tf**. tf is a ROS package that lets the user keeps track of multiple coordinate frames over time. tf maintains the relationship between coordinate frames in a tree structure buffered in time, and lets the user transform points, vectors, etc between any two coordinate frames at any desired point in time. In order to see real time the position and rotation of the arm_tool_link frame with respect to base_footprint frame will be enough to run this command in Ubuntu terminal is **rosrun tf tf_echo /base_footprint /arm_tool_link**

Once the available space is computed for both TIAGo versions it is time to see the second step, how to setup MoveIt in order to move TIAGo arm. The group demanded to move the torso and the arm concerning TIAGo single arm version is called arm_torso instead concerning TIAGo++ version it is called arm_right_torso in order to move the prismatic joint of the torso plus the 7 joints of the right arm. The following code can help you to setup MoveIt:

```
import rospy
from std_msgs.msg import *
import tf
from tf.transformations import quaternion_from_euler
from geometry_msgs.msg import PoseStamped
import moveit_commander
import moveit_msgs.msg
import math

#if is TIAGo single arm version
if single_arm:
    group_name = "arm_torso" #specify group name
#if is TIAGo dual arm version
elif dual_arm:
    group_name = "arm_right_torso" #specify group name
#assign the group to MoveIt Commander
group = moveit_commander.MoveGroupCommander(group_name)
group.set_goal_tolerance(0.01) #Set the goal tolerance
group.set_planner_id("SBLKConfigDefault") #Set the Planner
# Set the reference frame
group.set_pose_reference_frame("base_footprint")
```

Listing 5.3: MoveIt Setup

In this way MoveIt commander is ready to plan our trajectories.

In order to raise and lower TIAGo's arm the following function has been written:

```
def compute_1D_motion(translation):
    """
    This function, starting from the actual position of the
    arm_tool_link with respect to base_footprint
    and knowing the amplitude and the angle of the vector,
    computes the new position of the arm_tool_link frame
    Assumption: the orientation of the frame is constant and
    equal to [pi/2 , 0 , 0] (gripper parallel to the ground)
    In particular this function is used for the "1D vector" arm
    state, where our purpose is to control the z component
    Parameter:
        -translation: is a triple (x,y,z) that indicates the
    actual position of the arm_tool_link with respect to
    base_footprint
    Return:
        -translation: [x,y,z] with the z value modified
    """
    global teleoperate_arm
    angle = teleoperate_arm.vector_angle
    amplitude = teleoperate_arm.vector_amplitude

    print("A 1D vector with amplitude: " +str(amplitude) + " and
```

```
    angle: " + str(angle) + " has been detected")
17
18      # -- y = 0.00188889x represent the linear transformation that
        map
19      # -- 0 - 450 --> 0 - 0.85
20      transform = 0.00188889
21
22      # -- Only two measures of angle are valid --#
23      if angle == 90:
24          # -- The goal is to augment the z coordinate -- #
25          translation[2] = translation[2] + amplitude * transform
26
27      elif angle == 270:
28          # -- The goal is to decrease the z coordinate -- #
29          translation[2] = translation[2] - amplitude * transform
30      else:
31          print("1D vector angle not valid!")
32
33      # -- Limit the components to max or min -- #
34
35      # -- Limit the x component --#
36      if translation[0] > teleoperate_arm.max_x:
37          translation[0] = teleoperate_arm.max_x
38
39      elif translation[0] < teleoperate_arm.min_x:
40          translation[0] = teleoperate_arm.min_x
41
42      # -- Limit the y component -- #
43      if translation[1] > teleoperate_arm.max_y:
44          translation[1] = teleoperate_arm.max_y
45
46      elif translation[1] < teleoperate_arm.min_y:
47          translation[1] = teleoperate_arm.min_y
48
49      # -- Limit the z component -- #
50      if translation[2] > teleoperate_arm.max_z:
51          translation[2] = teleoperate_arm.max_z
52
53      elif translation[2] < teleoperate_arm.min_z:
54          translation[2] = teleoperate_arm.min_z
55
56      return translation
```

Listing 5.4: Compute 1D motion

As you can see looking the function above, the idea is to linear mapping the amplitude vector range obtainable in the '1D Vector GUI' to the available arm workspace. Talking about the z component the amplitude vector range is 450 (the amplitude of the vector can vary between 0 and 450) instead the feasible

workspace for the z component is 0.85 meters since the minimum z component is 0.45 meters and the maximum z component is 1.3 meters (referred to the base_footprint frame). In this way we have obtained the corresponding quantity to raise or lower TIAGo's arm depending on the angle of the vector. In particular if the angle is equal to 90 degrees the arm will be raise, contrary if the angle is equal to 270 degrees the arm will be lower. It is possible to know the actual orientation and position of the arm_tool link frame with respect to the base_footprint with the following line of code:

```
(trans,rot) = teleoperate_arm.tf_listener.lookupTransform('/
    base_footprint', '/arm_tool_link', rospy.Time(0))
```

where teleoperate_arm.tf_listener is an instance of a TransformListener object:

```
teleoperate_arm.tf_listener = tf.TransformListener()
```

The return value of the function in Listing 5.4 is the vector that contains the values for the position of the arm_tool_link frame with respect to base_footprint frame, after the effect of the selected 1D vector. As regards the implementation of the 2D vector motion function things are the same of the 1D case with some precautions:

- Vector angle says how much of the vector amplitude has to be distributed along the x and the y component
- When the angle is equal to zero and 180 all the amplitude influences only the y component (right and left)
- When the angle is equal to 90 and 270 all the amplitudes only the x component (back and forth)

The following code can help you understand how the function works:

```
def compute_2D_motion(translation):
    """
    This function, starting from the actual position of the
    arm_tool_link with respect to base_footprint
    and knowing the amplitude and the angle of the vector,
    computes the new position of the arm_tool_link frame
    Assumption: the orientation of the frame is constant and
    equal to [pi/2 , 0 , 0] (gripper parallel to the ground)
    In particular this function is used for the "2D vector" arm
    state, where our purpose is to control the x and y components
    The angle indicates how to change the components
    Explaination:
        If the angle is 90/270 degree, it simply means that the
    only component that will be affected is the x component
        If the angle is 0/180 the only component that will be
    affected is the y component
```

```
        All the angles between have to be mapped in order to modify both x and y components
    Parameter:
        -translation: is a triple (x,y,z) that indicates the actual position of the arm_tool_link with respect to base_footprint
    Return:
        -translation: [x,y,z] with the z value modified
    """
    global teleoperate_arm

    print("Entered in the function the translation is: " + str(list(translation)))

    # -- Store locally angle (in degree) and amplitude of the vector -- #
    angle = teleoperate_arm.vector_angle
    angle_radians = math.radians(angle)
    amplitude = teleoperate_arm.vector_amplitude

    # -- Amplitude range for 2D vector is [0 - 1006] --#
    # -- Transform is a factor that map linearly:
    # -- [0-450] --> [0 - 0.25] for x component
    # -- [0-900] --> [0 - 1] for y component -- #

    transform_x = 0.000555556 #(0.25 / 450)
    transform_y = 0.001111111 #(1/900)

    print("Amplitude: " + str(amplitude) + " Angle: " + str(angle))

    # -- Four cases depending on the angle 's value -- #
    # -- Angle it is important to understand how components change -- #
    # -- scaling_factor belongs to [0,1] -- #

    # -- x increase y decrease -- #
    if angle <= 90 and angle >= 0:
        print("Case 1")

        # --Decompose Amplitude along x and y component -- #
        x_amplitude = amplitude * math.sin(angle_radians)
        # print("Sin Result: " + str(math.sin(angle_radians)))
        y_amplitude = amplitude  * math.cos(angle_radians)
        # print("Cos Result: " + str(math.cos(angle_radians)))
        print("X amp: " + str(x_amplitude) + " Y amp: " + str(y_amplitude))

        # -- Found the scaling factor for the components -- #
```

```
52          translation[0] = translation[0] + (transform_x *
    x_amplitude)
53          translation[1] = translation[1] - (transform_y *
    y_amplitude)
54          print("Product for x component : " + str(transform_x *
    x_amplitude))
55          print("Product for y component " + str(transform_y *
    y_amplitude))
56
57      # -- x increase y increase -- #
58      elif angle <= 180 and angle > 90:
59          print("Case 2")
60
61          # --Decompose Amplitude along x and y component -- #
62          triangle_angle = 180 - angle
63          triangle_angle_radians = math.radians(triangle_angle)
64          x_amplitude = amplitude * math.sin(triangle_angle_radians
    )
65          # print("Sin Result:   " + str(math.sin(
    triangle_angle_radians)))
66          # print("Cos Result: " + str(math.cos(
    triangle_angle_radians)))
67          y_amplitude = amplitude  * math.cos(
    triangle_angle_radians)
68          print("X amp: " + str(x_amplitude) + " Y amp: " + str(
    y_amplitude))
69
70          # -- Found the scaling factor for the components -- #
71          translation[0] = translation[0] + (transform_x *
    x_amplitude)
72          translation[1] = translation[1] + (transform_y *
    y_amplitude)
73          print("Product for x component : " + str(transform_x *
    x_amplitude))
74          print("Product for y component " + str(transform_y *
    y_amplitude))
75
76      # -- x decrease y increase -- #
77      elif angle > 180 and angle <= 270:
78          print("Case 3")
79
80          # --Decompose Amplitude along x and y component -- #
81          triangle_angle = angle - 180
82          triangle_angle_radians = math.radians(triangle_angle)
83          x_amplitude = amplitude * math.sin(triangle_angle_radians
    )
84          y_amplitude = amplitude  * math.cos(
    triangle_angle_radians)
85          # print("Sin Result: " + str(math.sin(
```

```
    triangle_angle_radians)))
86          # print("Cos Result: " + str(math.cos(
    triangle_angle_radians)))
87          print("X amp: " + str(x_amplitude) + " Y amp: " + str(
    y_amplitude))
88
89          # -- Found the scaling factor for the components -- #
90          translation[0] = translation[0] - (transform_x *
    x_amplitude)
91          translation[1] = translation[1] + (transform_y *
    y_amplitude)
92          print("Product for x component : " + str(transform_x *
    x_amplitude))
93          print("Product for y component " + str(transform_y *
    y_amplitude))
94
95      # -- x decrease y decrease -- #
96      elif angle > 270 and angle <= 360:
97          print("Case 4")
98
99          # --Decompose Amplitude along x and y component -- #
100         triangle_angle = 360 - angle
101         triangle_angle_radians = math.radians(triangle_angle)
102         x_amplitude = amplitude * math.sin(triangle_angle_radians
    )
103         y_amplitude = amplitude  * math.cos(
    triangle_angle_radians)
104         # print("Sin Result: " + str(math.sin(
    triangle_angle_radians)))
105         # print("Cos Result: " + str(math.cos(
    triangle_angle_radians)))
106         print("X amp: " + str(x_amplitude) + " Y amp: " + str(
    y_amplitude))
107
108         # -- Found the scaling factor for the components -- #
109         translation[0] = translation[0] - (transform_x *
    x_amplitude)
110         translation[1] = translation[1] - (transform_y *
    y_amplitude)
111         print("Product for x component : " + str(transform_x *
    x_amplitude))
112         print("Product for y component " + str(transform_y *
    y_amplitude))
113
114     # -- Limit the components to max or min -- #
115
116     # -- Limit the x component --#
117     if translation[0] > teleoperate_arm.max_x:
118         translation[0] = teleoperate_arm.max_x
```

```
119
120     elif translation[0] < teleoperate_arm.min_x:
121         translation[0] = teleoperate_arm.min_x
122
123     # -- Limit the y component -- #
124     if translation[1] > teleoperate_arm.max_y:
125         translation[1] = teleoperate_arm.max_y
126
127     elif translation[1] < teleoperate_arm.min_y:
128         translation[1] = teleoperate_arm.min_y
129
130     # -- Limit the z component -- #
131     if translation[2] > teleoperate_arm.max_z:
132         translation[2] = teleoperate_arm.max_z
133
134     elif translation[2] < teleoperate_arm.min_z:
135         translation[2] = teleoperate_arm.min_z
136
137     # print("Before Exit Translation is " + str(list(translation)
    ) )
138
139     return translation
```

Listing 5.5: Compute 2D motion

Once the new position is computed it is now time to assign to MoveIt the new position to be reached. See the next portion of code in order to undestand how to assign the goal to MoveIt and perform the motion.

```
goal = [translation, gripper_orientation]
#goal has dimension equal to six, 3 positions and 3 orientations
 # -- Set the target and Move the arm -- #
group.set_pose_target(goal)
group.set_start_state_to_current_state()
group.set_max_velocity_scaling_factor(1.0)
plan = group.go(wait=True)
```

Listing 5.6: Assign goal to MoveIt

The parameter wait=True specify that the call is blocking. If the variable plan is true the plan and the motion were successful.

5.7 Arm Teleoperation and Free Mode

Should now clear for you how the teleoperation of the arm has been implemented. We have seen in the previous sections how the interfaces allow to select intuitively the wanted vector and how the vector information is dealt ROS side. At this point should be clear the overall architecture for teleoperate TIAGo's arm, in figure 5.6.

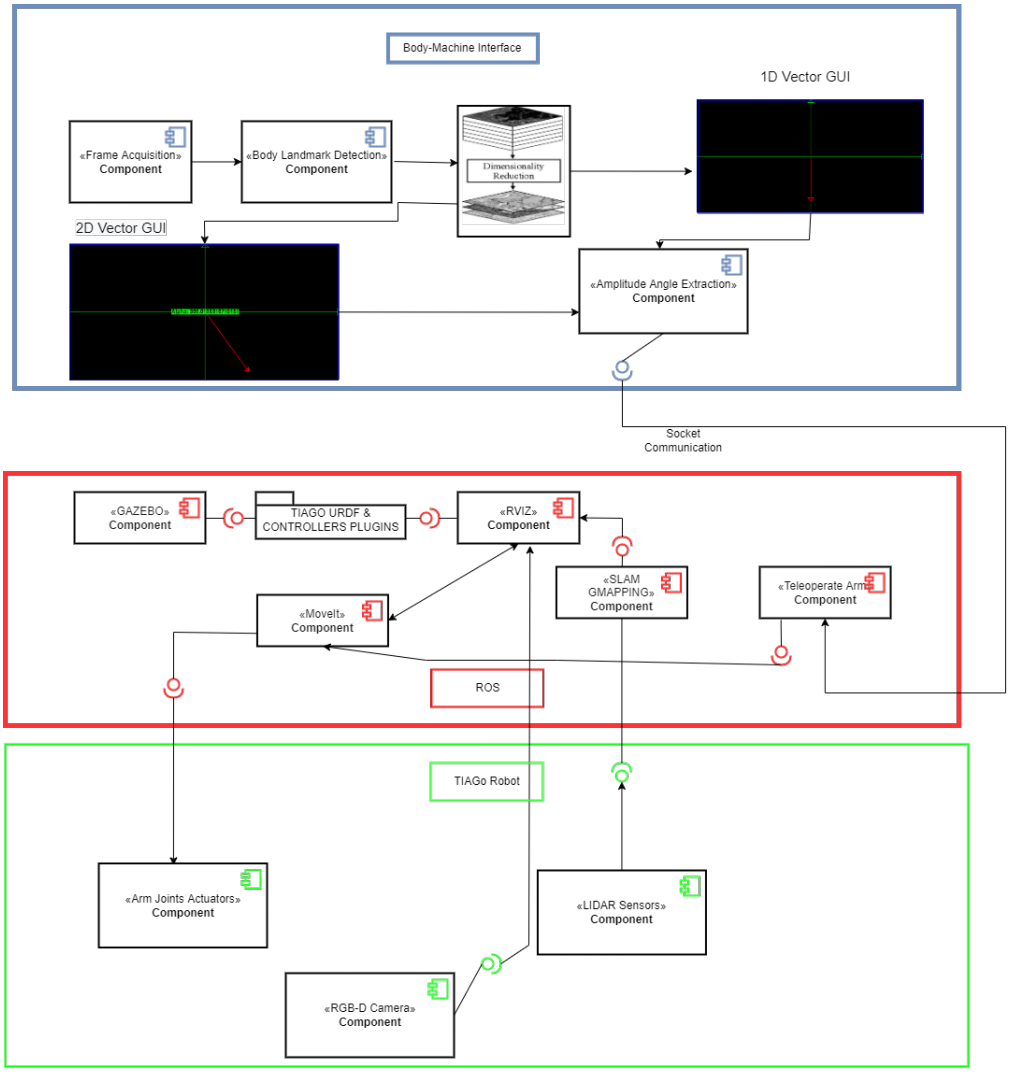


Figure 5.6: Arm Teleoperation Overall Architecture

Let's see now the behaviour of the arm teleoperation mode. As done for the mobile base, the final version of the 'Main Program' contains a dedicated mode to teleoperate only the arm. This mode has not great sense, but is thought to allow the user to make practice with the arm teleoperation. The arm teleoperation mode could be represented by the following state diagram:

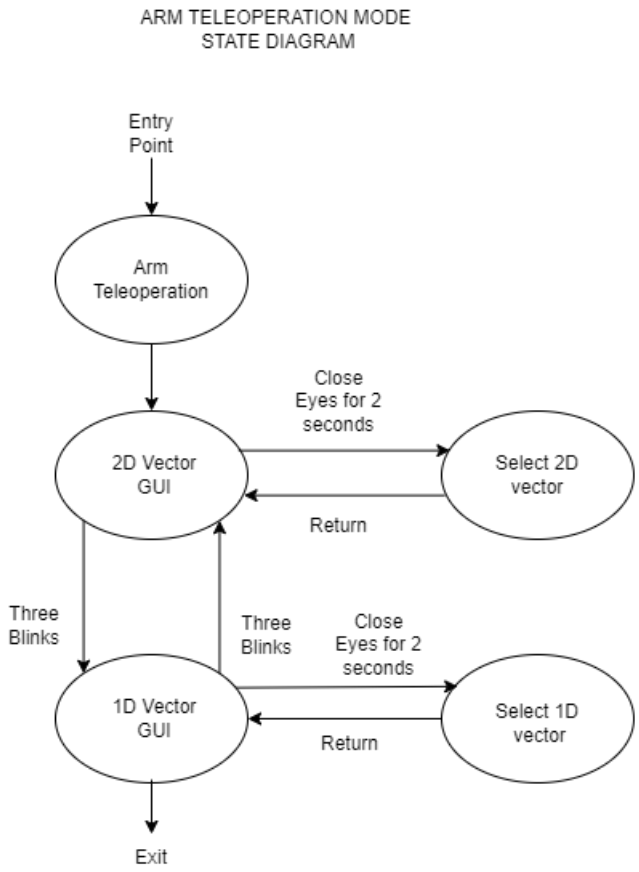


Figure 5.7: Arm Teleoperation Mode

Once started the arm teleoperation mode the FSM enter in the 2D vector GUI state, see figure: 5.4. Closing eyes for 2 seconds the user has the possibility to select a 2D vector in order to control TIAGo's arm in a 2D plane. Blinking eyes for three times the user has the possibility to switch the interface and to control the altitude of the arm, see figure: 5.5, once chosen 1D vector the user can select it maintaining body position fixed and closing eyes for 2 seconds. Blinking again three times eyes the user return to control the arm on a 2D plane and so on.

In order to control both TIAGo's mobile base and the arm, a new mode has

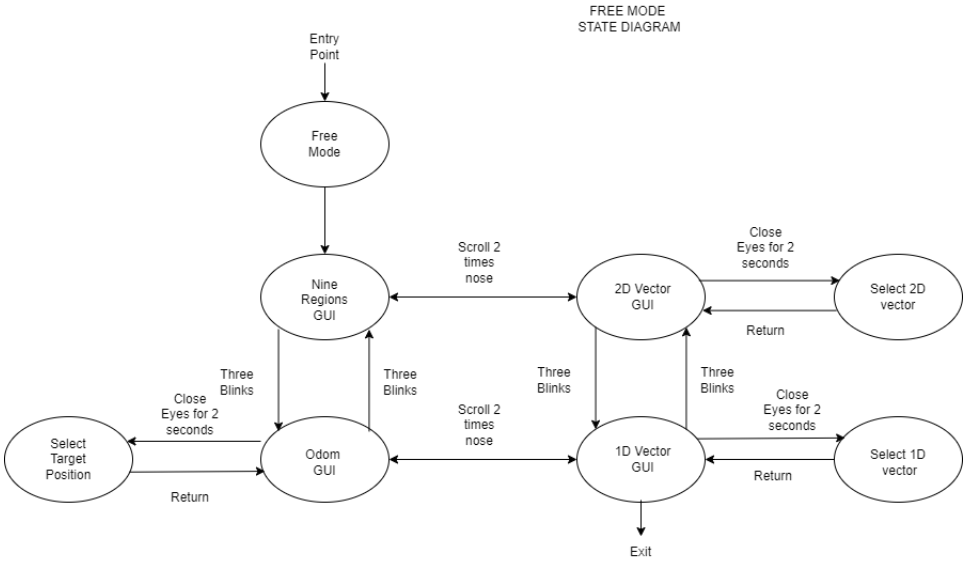


Figure 5.8: Free Mode State Diagram

to be created, which mixes the two teleoperation mode. This new mode is called Free Mode and it is developed in order to allow user to make practice both with TIAGo's mobile base and its arm. Free Mode's behaviour can be summarized looking the state diagram 5.8.

The user has the possiblity to switch between base teleoperation and arm thanks to nose detector script explained in section 4.8. In particular right crossing the precalculated threshold two times in two seconds the user can change teleoperation mode.

CASE STUDY: Reach a bottle placed on a table 2 meters in front of TIAGo

In this way the steps to reach a bottle placed on a table two meters from TIAGo are:

1. Run the Main Program and select the wanted joints, at least shoulders.
2. Make all the calibrations (body dance, nose and eyes calibration)
3. Run server socket on the second machine.
4. Blink three times eyes in order to control TIAGo with Odom GUI and select a point 2 meters far from TIAGo.

5. Once TIAGo has reached the target position, the user can blinks again three times eyes and control manually TIAGo position and orientation in order to make possible to reach the bottle.

6. Switch teleoperation mode using nose.

7. Reach the bottle using ad hoc GUIs reserved to teleoperate the arm.

Of course no

Chapter 6

Conclusions

Write the conclusions here...

References

[1] C.Pierella; M.Casadio; E.Galofaro; A. De Luca; L.Losio; S.Gamba; A.Massone; A.Mussa Ivaldi (2021), *Recovery of Distal Arm Movements in Spinal Cord Injured Patients with a Body-Machine Interaface* 2

[2] A. Farshchiansadegh; F. Abdollahi; D. Chen; Mei-Hua Lee; J. Pedersen; C. Pierella; E. J. Roth; I. Seanez Gonzalez; E. B. Thorp; F. A. Mussa-Ivaldi, *A Body-Machine Interface based on Inertial Sensors*

[3] M.Casadio; A. Pressman; S. Acosta; Z. Danzinger, A.Fishbach; F. A. Mussa-Ivaldi; K. Muir; H. Tseng; D. Chen (2011), *Body machine interface: remapping motor skills after spinal cord injury* 2

[4] E. Thorp; F. Abdollahi; M. Ivaldi; C.Pierella (2016), *Upper Body-Based Power Wheelchair Control Interface for Individuals With Tetraplegia* 2

[5] A. Carfi; F. Mastrogiovanni, *Gesture-Based Human–Machine Interaction: Taxonomy, Problem Definition, and Analysis* 1

[6] J. Pages; L. Marchionni, *TIAGo: the modular robot that adapts to different research needs*

[7] M. E. Cabrera; K. Dey; K. Krishnaswamy; T. Bhattacharjee; M. Cakmak (2021), *Cursor-based Robot Tele-manipulation through 2D-to-SE2 Interfaces* 1

[8] M. Ciocarlie; K. Hsiao; A. Leeper; D. Gossow (2012), *Mobile Manipulation Through An Assistive Home Robot*

[9] R.Murphy (2004), *Human-robot interaction in rescue robotics* 1

[10] T. Fong; R. Berka; M. Bualat; M. Diftler; M. Micire; D. Mittman;V. Sun-Spiral; C. Provencher (2012), *The human exploration telerobotics project* 1

[11] M. J. Lum; D. C. Friedman; G. Sankaranarayanan; H. King; K. Fodero; R. Leuschke; B. Hannaford; J. Rosen; ; M. N. Sinanan (2009), *The raven:Design and validation of a telesurgery system* 1

[12] A. Munawar; G. Fischer (2016), *A Surgical Robot Teleoperation Framework for Providing Haptic Feedback Incorporating Virtual Environment-Based Guidance* 1

[13] M. Moro; F. Rizzoglio; M. Casadio, *A Video-Based MarkerLess Body Machine Interface: A Pilot Study* 6

[14] M. Moro @ https://github.com/MoroMatteo/markerlessBoMI_FaMa *Installation Tutorial and Explanation of Markerless BoMI* 7

[15] By Google (2019), *Github Link: https://github.com/google/mediapipe* 7

[16] S. L. Colyer; M. Evans; D. P. Cosker; A. I. T. Salo, (2018), *A Review of the Evolution of Vision-Based Motion Analysis and the Integration of Advanced Computer Vision Methods Towards Developing a Markerless System* 6

[17] C. Pierella; A. Sciacchitano; A. Farshchiansadegh; M. Casadio; Member, IEEE and S.A. Mussa-Ivaldi, Member, IEEE, (2018) *Linear vs non-Linear mapping in a body machine interface based on electromyographic signals* 5

[18] Georgia Insitute of Technology (March 2009) *EL-E: An Assistive Robot* 1

[19] F&P Personal Robotics *LIO Robot as new type of assistive Robot* 2

[20] MATIAS Robotics 2020. *Anatomical Concepts – The TEK RMD from Matias Robotics* 2

[21] Valeria Seidita, Francesco Lanza, Arianna Pipitone, and Antonio Chella (December 2020) *Robots as intelligent assistants to face COVID-19 pandemic* 10

[22] Antonio Andriella,Carme Torras,Carla Abdelnour and Guillem Alenyà (March 2021) *Introducing CARESSER: A framework for in situ learning robot social assistance from expert knowledge and demonstrations* 10