SAPIENZA
UNIVERSITÀ DI ROMA

DEPARTMENT OF COMPUTER, CONTROL AND MANAGEMENT ENGINEERING

A Pepper Robot museum guide

ELECTIVE IN AI - HRI + RA

Professors:
Luca Iocchi
Fabio Patrizi

Students:
Enrico Fazzi
Luca Distefano
Gerardo Loffredo

All students contributed equally to the project

Contents

1 Introduction

In the last few years, the development of innovative solutions in the area of Robotics and Artificial Intelligence has given birth to new compelling projects that aim at realizing smart robots capable of interacting with humans and, at the same time, correctly interpreting the environment to find solutions to complex problems and tasks. The development of cooperative and reasoning agents is indeed an interesting and extremely useful branch of AI that relies on two different yet complementary concepts: the Human-Robot Interaction (HRI), which enables safe and fast interaction between users and machines, and Reasoning Agents (RA), which enables agents to collect or receive data, represent and interpret it and compute efficient solutions.
The two aforementioned concepts of HRI and RA are the basis of the project we developed: our objective, indeed, was to merge these two branches in order to realize an agent that interacts with users to receive information that will be leveraged to compute a plan that will solve the given task.
In our work, we intended to obtain a robotic guide in a fictitious art museum that is able to correctly interact with users and plan the shortest path to make our guests visit the indicated paintings in the shortest time possible.
More specifically, our robot is able to welcome our guests, ask them for some specific information that will be discussed in the following chapters, let the guests select the paintings they would like to visit, develop the shortest path to lead our visitors throughout the rooms of our museums that contain the requested works, and eventually bring them to the closest exits. Explanations of paintings and random events have also been developed to make our work as close as possible to a real-case scenario.
We have decided to leverage the robot's tablet to realize the interaction, developing a user-friendly interface that enables visitors to easily introduce the necessary values that will be used by the robot's planner to compute a path in our museum. Moreover, random events will also influence the tablet's activity, allowing us to have a dynamic and responsive interface.
Throughout our work, we leveraged the available libraries and software tools to work with the smart agent Pepper Robot developed by SoftBank Robotics. Additional information will be given further in the report. The remainder of this report is organized as follows: Section 2 provides an overview of the available papers that inspired our projects, Section 3 provides a meticulous description of the different elements that constitute our project, Section 4 describes formally the libraries and software tools we leveraged, while in section 5 we will discuss the results obtained, showing the behavior of our robot.
Section 6 concludes the report.

2 Related works

Several technologies have been developed over the past 30 years to improve the experience of visiting a museum. The importance of achieving a good level of interaction between the user and machines relies on various aspects: for instance, a robot that can interact with a human can, subsequently, correctly interpret the task and adapt easily to any specific requests the user may formulate, improving the results obtained. Moreover, employing machines in public spaces and therefore in constant contact with people requires reaching a high level of HRI. To this aim, different technologies have been developed, ranging in scope from apps and virtual realities to the latest social robots. In particular, the implementation of the latter has facilitated the provision of users with a visit experience that encompasses varying degrees of utility and usability. Germak et al.[1] distinguish according to their use of the robots into 3 categories: museum guides, telepresence, and installation.

2.1 Museum guide robots

The most commonly used robots as museum guides are mobile robots. Over the years humanoids have slowly replaced early mobile-based robots, due to their use in the interaction with humans. The first robot used as a museum guide was Rhino[2], a wheeled mobile base deployed at ”Deutsches Museum” in Bonn, Germany. This robot was followed by its successor Minerva [3], where the robot was equipped with a head able to produce facial expressions. Subsequent robot work focused on autonomous navigation within museums, trying to implement more efficient mapping in dynamic environments. The first implementation of these better techniques goes back to the ”TOURBOT” project [4]. In this project, it appears also the first type of teleoperation through web interfaces. Then the focus moved from autonomous navigation to a better interaction with humans, including speech recognition and emotional states. One of the first robots that introduced these aspects was ”CiceRobot” [5], used in Agrigento and that allowed humans to ask questions. One of the last works [6] focused, instead, on high-level planning and long-term operativity. Most recent works are now focusing on human-robot interaction and to a more humanoid appearance. In the Osaka Science Museum in 2006, four humanoids [7] were used. They were equipped with an infrared camera tracking system for Localization and active RFID tags for interactions with visitors. In the recent past, six Pepper robots were used in Washington and they were able to answer users with voice and gestures [8]. One of the most recent works is [9], where the authors developed an autonomous robot where intelligence is off-loaded to a remote machine via a 5G network.

2.2 Telepresence robots

Telepresence robots are restricted in their use and in most cases are useful to the user where there are inaccessible parts within the museum. Usually, they are employed with a camera and a screen. One of the most famous is the telepresence robot used at the Tate Museum in London [10].

2.3 Planning algorithm

The planning algorithm we used was first introduced in [11]. In the paper, it is presented a classical planning system based on heuristic search, characterized by a forward-directed search in the space of world states. The goal is finding a sequence of discrete actions that leads from an initial state to a goal state.
The algorithm is characterized by a translation phase, in which the PDDL input is converted into a finite domain representation. Specifically, the PDDL problem is first parsed into an internal representation; successively, it is converted into a normal form; eventually, the problem is converted into a finite domain representation by creating state variables with finite domains.
During the search phase, instead, heuristics techniques are employed to find a solution. A general scheme of the algorithm is the following:

1. Initialize the search with the initial state.
2. Calculate heuristic values for the initial state.
3. Expand the initial state to generate successor states.
4. Use the chosen search algorithm to explore the state space guided by the heuristic.
5. Continue the search until the goal state is reached.
6. Output a sequence of actions that constitute the plan.

This planner can also be used as a One-shot planner, meaning that the plan is directly generated from an initial state to a goal state without iteratively revising the plan itself.

3 Solution

In the following lines, we will describe the different components of our solution, analyzing their roles and the steps followed through their realization. These elements will be divided into related to the HRI branch (3.1) and RA branch (3.2).

3.1 HRI solution

3.1.1 Pepper robot

To develop our Pepper robot we decided to leverage the Android Studio software emulator and the Aldebaran libraries for Pepper robot. Further details will be given in the next chapters. Here we will limit to describe the functions of our robot and the actions it executes.
We have defined a class Pepper robot that presents different methods; each method is an action that the robot will execute leveraging the two modules *ALMotion* for movement and *ALSpeech* for speaking. In particular, the robot starts being in an idle state waiting for a human to approach him. When a human writes on a terminal the command *start*, the robot starts executing the first interaction, in which the robot welcomes the user by moving his arm waving his right hand, and giving the user instructions on how to proceed. The robot then proceeds to pass again to an idle state awaiting for the user to interact with his tablet. When the user clicks on the tablet, a start tour flag is set to True and the robot can pass to execute the following actions. At this point, our robot will have received from the client Pepper the actions planned by our planner and will be able to execute them one by one. Before the execution, though, the robot infers the order of the paintings to be visited by analyzing the list of actions and creating a new list in which only the paintings will be added. The Pepper client will then be leveraged to send the list as a message to the Server so that the message will be ultimately forwarded to the Tablet client.
We now pass on the execution of the actions received by the planner. Deeper insights on the two planners will be given in 3.2, as well as further information on the actions; now we limit to say that the action can be of three different types: **is_queue**, if there is a queue in the paint we intend to visit; **move**, when the robot moves from room A to room B; **visit** when we have arrived in a room with a selected paint and we can proceed to visit it.
If there is a queue in a paint we want to visit, the robot will first extract the room and the paint from the action string; it will then map the paint's indices (for example "p1") with its title (for example "The Birth of Venus"), inferring it from the *painting _info* dictionary. Eventually, it will say to the visitors that there is too much queue in the current paint, so no visit will be realized.

If the robot has to move from room A to room B, it will first extract from the action string the two rooms (where he currently is and where he wants to go). It will then store room B, since we may need it in the case in which the user decides to end the tour after visiting the following painting. It will, successively, instruct the visitors to move from A to B leveraging the *ALSpeech* module and will finally move in the simulated environment. To realize this motion, we had to consider not only the two rooms in which it currently is and where it wants to go but also the room where it was before moving to room A. In fact, this information is extremely important to determine the orientation the robot will assume when moving from A to B. For instance, if we refer to , imagining that we want to pass from l4 to l5, the robot will have to turn right if it previously was in l3, but it will have to go back if it previously was in l5 since it will arrive in l4 with two different orientations. To realize this, we defined a dictionary of dictionaries *museum_map*, in which each possible couple of rooms A and B represents the key of the first dictionary, with its values being another dictionary; in this last dictionary, each key is represented by the previous room in which the robot was before A, and its value is a string that represents the direction the robot will assume when passing from A to B. Leveraging, therefore, the information on room A, room B, and the previous room we can extract the direction and use it to make the robot turn left (rotating 90°), turn right (rotating -90°), go back (rotating 180°) or not rotating at all, and finally move forward for 2 seconds at a 0.5 m/s speed. We decided to leave a constant moving time between rooms since our map is simulated; a different time can be for example added by creating a dictionary similar to the museum_map one, in which the keys are all the possible combinations of rooms A and B and their values are the respective commuting times.
Eventually, if the robot has to visit a paint, it will first extract the index of the paint from the action string and associate it with its title leveraging the "paintings_info" dictionary as previously explained. It will then check if the work's title is contained in a list in which we have all the paintings that can not be photographed with a flash. If the check is positive, the robot will talk and warn the visitors to not take pictures with a flash. We then generated a random event that should simulate the possibility that a user takes a photo with flash when it is not allowed to. In this case, the robot moves his head from left to right and talks to the visitors reproaching the group. Similarly, we also generate a random event that simulates if someone is too close to a paint. The probability of this event is influenced by the number of people in the group of visitors, and information inserted by the user using the robot's tablet. More visitors means a higher probability for this event. If the event occurs, the robot reproaches those who are too close to the paint by talking and shaking his head. Consequently, the robot sends a message to the Tablet client stating that the visit is ready to start and starts describing the characteristics of the paint by talking. While talking, it moves his right arm simulating an explaining gesture. The robot successively passes to an idle state

that will be terminated when the user clicks on the next or the end button of the tablet. In the former case, the robot will proceed to execute the following action; in the latter, it will leverage the Pepper client to send the message *"Tour aborted"* plus the room in which he currently is (stored during the execution of the move action) to the Server. This information will be processed to activate a second planner that will compute the shortest path to reach an exit and execute its actions to exit the museum. Further details will be given in 3.2.

Once all the actions are completed, it will communicate it to the Server and it will interact with the visitors thanking them and asking for an evaluation. It will hence pass in an idle state until an evaluation is inserted in the tablet (further details in 4.2) and, according to the number of stars given, it will react in three different ways: happily, if the given stars are 4 or 5, sadly if the stars are 1 or 2, mildly if the stars are 3. The user will be then thanked again and the interaction stops.

3.1.2 Tablet

The Tablet solution and implementation will be fully described in Section 4.2.

3.2 RA solution

In the realization of our robotic guide, we aimed at obtaining a smart agent capable of not only interacting with the user but also formulating a smart plan in a reduced temporal span to drive in the fastest way possible the user throughout the rooms of our museums. In order to do so, we needed to furnish our agent with complete offline knowledge of our museum.

We developed two different planners: the first one receives in input the list of paintings the user wants to visit and outputs the set of actions the robot must execute to solve the task of visiting every painting and bringing the visitors to the closest exit; the second receives as input the room in which the group currently is and outputs the path to follow to reach the closest exit. The planner component has been developed in Python3 and PDDL and runs locally on our PCs.

To better understand the planners' roles it is important to introduce a map of the museum. Our idea was to generate a museum with 20 rooms, from l0 to l19. There are three accessible rooms (l6, l11, and l13) and 15 paintings. The disposition of the paintings as well as the connection between rooms can be inferred from Figure 1. The museum has one entrance e0 and three possible exits e0, e1, and e2.

Our first planner is characterized by different predicates that defined the rooms, the entrances and exits of our museum, the paintings, and their locations (we used a *"has_painting_pred(A, pi)"* for room A and the i-th paint), a robot predicate, a connection predicate for each couple of communicating room, an *"at_pred"* that defines in which room the robot currently is, a visited predicate that is True if a painting has

been visited, and a queue predicate that assumes a true value if there is a queue in a specific work.
We also defined a set of three possible actions: the first action is defined as *"is_queue"* and it represents the possibility of finding a queue in the i-th painting; the action has as preconditions that the robot must be in the room in which the i-th painting is located, the i-th painting must not have been already visited and the predicate "is_queue" must be True. In particular, this last predicate assumes a true value according to a random event generated with different probabilities, depending on the indexes of the painting itself: for the paintings with indexes [0,4,6,11,12,14] there is a low probability of finding a queue; for the indexes [3,5,13,10] there is a medium probability and for the paintings with indexes [1,2,7,8,9] there is a higher probability. When defining the initial conditions for our planner, we check for each painting if the considered event realizes and hence set to True the "is_queue" predicate if the event itself is True. The effect of the action is that we will consider the paint as visited. In a real case scenario, we may consider just waiting for the queue to disappear adding an idle time, for example, for the robot. This time could be used by the group to rest, but in a simulated case we deem as useless adding a waiting time and decided to work with a different robot interaction for this case.
The second possible action is defined as *"move"* and it represents the movement of the robot between room A and room B. The preconditions for this action are the necessity of the robot to be in A and that there is indeed a connection between A and B. The effect of the action will act on the *"at_pred"* predicate, stating that we will not be anymore in A and we will be in B instead.
The last action is the *"visit"* action and has as effect that the predicate "visited" assumes a true value for the i-th painting. Its preconditions are the necessity of being in the room in which the i-th paint is located, the i-th paint must not have been already visited, the robot must be in the room in which the i-th work is, and there must not be a queue in the considered painting.
As regards the initial conditions, we defined the predicates for the rooms, robot, room connections, paintings, and entrances as True; we established that initially, the robot is in e0, the entrance; we assigned each paint to the corresponding room; we set to True according to the random event the "is_queue" predicate as previously commented; we set as false the "visit" predicate for each painting.
The goal state has instead been defined as having visited all the paintings we received in input and being in one of the three exits of the museum.
To solve the planning problem we employed an *Oneshot* planner, a planner that attempts to find a feasible plan in a non-iterative fashion, with optimality guaranteed as a parameter, which guarantees us that the found solution is the optimal one; we used as the solving algorithm the fast-downward algorithm [11] first described in chapter 6. The obtained solution is a list of actions, in which each action is a string; we save this

list as a string itself and pass it to the Server, which will successively forward it to the client Pepper.
The second planner is similar to the first one. The main differences are the absence of the actions "visit" and "is_queue" since it aims to bring our visitors to the closest exits and the definition of the initial and goal state.
The goal is now defined as just being in one of the three exits; the initial condition is slightly modified since we do not consider anymore the paintings and their positions, and we set as True the "at_pred" for the room in which the robot is when the second planner is called. To do so, the Server receives from the Pepper client the room in which it currently is and passes it to our second planner. The solution will be a list of "move" actions that will bring the robot to the closest exit.

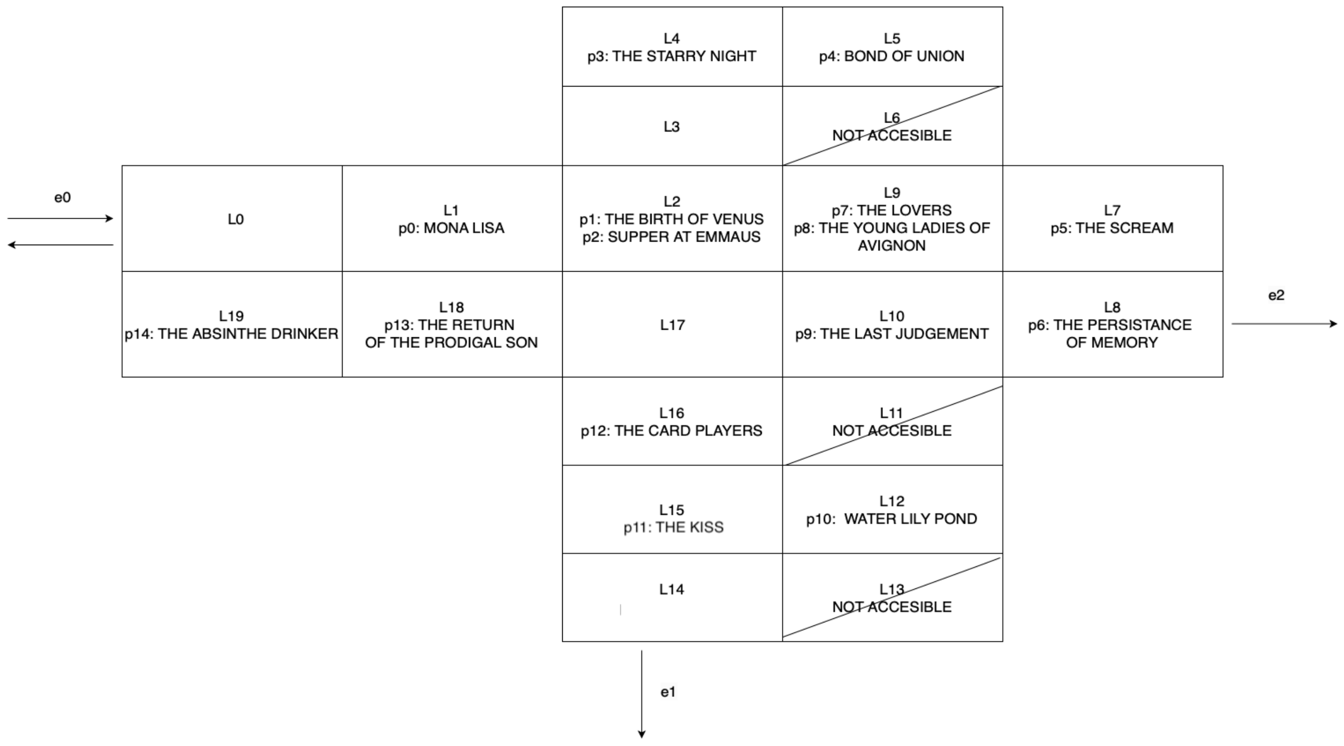

Figure 1: Museum's map

4 Implementation

In this Section, we intend to discuss the different tools and software platforms used to develop our project. We will explain our implementation choices, showing the interactions between its parts and the corresponding snippets of code. We split our discussion into three components *Pepper*, *Tablet* and *Planner*, in which we respectively discuss tools and methodologies used to create motion and speech, to create a tablet interface and to create the plan used by the robot to navigate in the museum.

4.1 Pepper

In this Paragraph, we will briefly discuss the tools we used to realize the movements of the robot, to allow its speech, and to visualize all these, and then we will provide technical details of the implementation.

4.1.1 Tools

To begin, we present the tools and software platforms that enabled us to develop our work. The core tool for the development of the project is undoubtedly *Docker*, which allows the creation of one or more independent containers in which it is possible to work separately without applying modification to the entire system. In this case, our Docker container includes the main tools of our work: the *pepper_tools*. These last ones are used to create Pepper's motion and to simulate the interactions Pepper would have in a real environment with humans. It is important to mention that unfortunately not all tools were available and, as we will see in the following sections, some aspects (e.g. the presence of humans near the robot) are simulated or even replaced as in the case of tablet functionalities by other tools. The following lines of code are useful to run the Docker image

```
./run.bash
```

and to enter in the robot's architecture

```
docker exec -it pepperhri tmux a
```

Concerning the movement and speech of the robot, we decided to develop all the code in Python, thanks also to the availability of the libraries provided by *NAOqi*. In order to facilitate the development of the code, we utilized one of the most prevalent text editors, namely *Visual Studio Code*. To use the Pepper tools within the Docker container, we were required to rely on the common directory, designated as *Playground*. Finally, to visualize all the actions performed by Pepper and executed on the tablet, we relied on *Android Studio*.

4.1.2 Methods

Let's describe in detail the methodologies used to develop motion and communication. We start our discussion from the initial step of the interaction where the robot simulates the approach of a human. As we mentioned in the 3, the robot is waiting for a *Start* command from a human. Once it receives the start from the command line, Pepper starts the interaction. In particular, the robot welcomes the user to speak and makes a classical gesture of greeting. As aforementioned, the libraries used to move the robot and to make it say sentences are *ALMotion* and *ALTextToSpeech*. The sentences that the robot says are defined in an *util* file, while the movements that the robot will do are defined in another file, namely *motion.py*. The following snippet of code is related to the initial greeting:

```
def move_greeting(robot):
  session = robot.service("ALMotion")
  isAbsolute = True
  jointNames = ["RShoulderPitch", "RShoulderRoll", "RElbowRoll",
    "RWristYaw", "RHand", "HipRoll", "HeadPitch"]
  jointValues = [-0.141, -0.46, 0.892, -0.8, 0.98, -0.07, -0.07]
  t = 1.0
  times  = [t, t, t, t, t, t, t]
  session.angleInterpolation(jointNames, jointValues, times, isAbsolute)
  for i in range(2):
    jointNames = ["RElbowYaw", "HipRoll", "HeadPitch"]
    jointValues = [1.7, -0.07, -0.07]
    times  = [0.6, 0.6, 0.6]
    session.angleInterpolation(jointNames, jointValues, times, isAbsolute)
    jointNames = ["RElbowYaw", "HipRoll", "HeadPitch"]
    jointValues = [1.3, -0.07, -0.07]
    times  = [0.6, 0.6, 0.6]
    session.angleInterpolation(jointNames, jointValues, times, isAbsolute)
  return
```

As we explained before we used the method *ALMotion*. We considered the joints to raise Pepper's right arm and to make it undulate. It is possible to notice that Pepper repeats the undulation twice. In the following section, we will show the results of all motions step-by-step. After the greetings, the user will start his interaction with the tablet. Following interaction with the tablet, the robot will then proceed to provide guidance to users through the museum. To do it the robot has to know the entire map of the museum and the direction of the motion. As we mentioned in 3, we defined the links between the different rooms of the museum, adding them in the *utils.py* file. We show an example of the connection between the two rooms and the code that makes the motion between the two rooms:

```
"l4,l5": {"l3": "right", "l5": "back"}

def motion_rooms(robot, l1, l2, prev):
    # Extracts the room from where the robot moves from and where it wants to go
    key = "{},{}".format(l1,l2)
    direction = ""
    # Defines which is the direction the robot has to follow to get to the next
    if key in museum_map:
        if prev in museum_map[key]:
            direction = museum_map[key][prev]
    print(direction)
    # Moves the robot according to the infer direction
    session = robot.service("ALMotion")
    if direction == 'left':
        # Rotate 90 degrees to the left
        session.moveTo(0, 0, 1.5708)  # 90 degrees in radians
    elif direction == 'right':
        # Rotate 90 degrees to the right
        session.moveTo(0, 0, -1.5708)  # -90 degrees in radians
    elif direction == 'front':
        # Move forward
        session.moveTo(0.5, 0, 0)
    elif direction == 'back':
        # Rotate 180 degrees
        session.moveTo(0, 0, 3.14159)  # 180 degrees in radians

    # Moves forward at 0.1 m/s
    session.move(1, 0, 0)
    # Moves forward for 2 seconds
    time.sleep(2)
    session.stopMove()
    # Repasses to speech part
    session = robot.service("ALTextToSpeech")

    return
```

The function *motion_rooms* extracts the direction of the motion considering a third room, which represents the room where the robot was previously, as explained in

3.1.1. As we can notice the angle rotations are expressed in radians. At the end of the execution, it returns to the method *ALTextToSpeech*. To complete our discussion about speech and motion, we briefly show the remaining motions that are executed respectively when someone is too close to an artwork and when someone is taking a picture where it is not allowed to. These are the functions that execute those motions:

```
def look(robot, direction):
    session = robot.service("ALMotion")
    jointNames = ["HeadYaw", "HeadPitch"]
    isAbsolute = True
    if direction == "left":
        initAngles = [0.5, -0.2]
        timeLists  = [1.0, 1.0]
        session.angleInterpolation(jointNames, initAngles, timeLists, isAbsolute)
    if direction == "right":
        finalAngles = [-0.5, -0.2]
        timeLists  = [1.0, 1.0]
        session.angleInterpolation(jointNames, finalAngles, timeLists, isAbsolute)
    return

def new_talk(robot, n_hands = 2):
    session = robot.service("ALMotion")
    isAbsolute = True
    jointNames = ["RShoulderPitch", "RShoulderRoll", "RElbowRoll",
        "RWristYaw", "RHand"]
    jointValues = [0.5, -0.46, 0.892, 1.5, 0.98]
    times = [1.0, 1.0, 1.0, 1.0, 1.0]
    times = [time * 0.7 for time in times]
    session.angleInterpolation(jointNames, jointValues, times, isAbsolute)

    jointNames = ["RElbowYaw", "HipRoll", "HeadPitch"]
    jointValues = [2.7, -0.07, -0.07]
    times = [0.6, 0.6, 0.6]
    for i in range(2):
        session.angleInterpolation(jointNames, jointValues, times, isAbsolute)
    if n_hands == 2:
        jointNames.append("LElbowYaw")
        jointValues.append(-2.7)
        times.append(0.6)
```

4.2 Tablet

In this section, we will explore in detail the Android application developed for the tablet, analyzing every fundamental aspect of it. The description is structured in several sections, each highlighting a specific aspect of the development process. Below is an overview of how the chapter will be organized:

- Macroscopic Structure: We will begin with an overview of the macroscopic structure of the application. This section will provide a general description of the main components of the app, highlighting the main activities and the flow of navigation between them.
- Libraries Used: Next, we will examine the external libraries used to enhance and facilitate the development of the app. We will discuss the functionality these libraries offer and how they have been integrated into the project.
- Aesthetics and Design: The aesthetics section will focus on the visual design of the application. We will analyze the design choices, including the layouts used, the graphic elements defined in the drawable files, and the styles applied. We will see how these choices contribute to a consistent and pleasant user experience.
- Technical realization: In this section, we will go into the technical details of the app's realization. We will show how the application logic, implemented in Java, integrates with the layouts defined in XML. We will analyze the main Java files that manage user activities and interactions, explaining their role and operation. We will also see how the graphical components are managed and updated during the app execution.
- Future Upgrades and Improvements: Finally, we will explore possible upgrades and improvements in future versions of the application. We will discuss ideas for new features, performance optimizations, and user interface improvements.

This structure will allow us to analyze the application in a systematic and in-depth manner, providing a clear and detailed understanding of each of its components.

4.2.1 Application Folder Structure

Describing the file architecture in an Android application is crucial to understanding how the various components, that enable the app to work properly, are organized. Mainly will be described the manifest file that defines essential app information, the *"gradle"*, which is about the build process performed by Android, the Java elements for the basic logic of the app, the layout elements realizing the GUI via XML files, and lastly the graphic resources in the *"drawable"* directory, consisting of images, objects and graphic customization of the active elements in the GUI.

Manifest

The Manifest file is an essential component of every Android project. It provides crucial information to the Android system about how to run the application and which functionalities it can use. It essentially represents an "identity card" for the application. More specifically, a Manifest deals with definitions such as the declaration of the app's package name, description of the app's components (activities, services, broadcast receiver, content provider), specification of the required permissions, declaration of the required minimum Android API level, list of external libraries used as well as aesthetic definitions regarding icons and themes applied. For the development of the application, it was therefore essential to precisely define the structural elements, like the specific *"namespace"*[1], the permissions, in particular for the use of the hardware characteristic of Pepper (a robot developed by Softbank Robotics), define the general properties of the application, such as the request for a larger memory heap, and the specification of the customized theme. Fundamental is the declaration of the app's main activity, which is launched when the app is started, and all other activities. Each activity represents a single screen view the user can interact with. Activities are declared without additional intent filters, as opposed to the main activity, so they are not automatically started when the app is started.

Gradle

The provided *build.gradle* files are used to configure and manage the dependencies of an Android project. Basically is defined by two files: one for the global configurations for the project, like the gradle versions and the plugin applied, and the other is specific for the app module, including dependencies, SDK versions, and the specific configurations of the build (like debug, release, and so on). The Android plugin for applications has been applied, allowing the Android project to be built and compiled, in addition to specifying the SDK version with which the project will be compiled and the default configurations. In the dependencies section, AndroidX libraries for compatibility and UI have been included, as well as testing libraries (JUnit, Espresso), SDKs for the Pepper robot, and additional libraries for JSON and basic components (androidx.core).

Exploited Libraries

All the activity classes extend *AppCompatActivity*, a *FragmentActivity* subclass, and part of the *AndroidX* support library. The reason for using this extension is for the following reasons:

- Inherits the methods to handle the life-cycles of Activity Android.
- Allows the use of modern *UI* functionalities on less recent versions of Android.

[1]An attribute used to organize and generate classes, ensuring consistency and uniqueness of names within the project.

- Natively supports elements from the Material Design interface.
- Provides a consistent implementation of the Action bar over different versions of Android. In particular, allows us to remove it and visualize the full frame of the view of the application, a problem that could not be solved if it was extended to a more suitable RobotActivity[2] class.

In order to realize the communication with the server, the implemented *WebSocketClient* class exploits *OkHttp* library. *OkHttp* is an open-source HTTP client library for Android and Java applications, developed by Square[3]. It simplifies the process of making HTTP requests and handling responses, providing robust features such as connection pooling, transparent GZIP compression, and response caching to improve efficiency and reduce latency. OkHttp supports both synchronous and asynchronous requests and offers a simple, flexible API that makes it easier to handle network operations in a reliable and efficient manner.

GUI features
Minimalism and a soft grey-blue palette characterize the visualization of the application's pages, conforming as far as possible to Google's material design rules. The aesthetic and functional features of the graphic interface follow the basic rules of a proper user experience, thus favoring ease of use and visual cleanliness, considering that its main use is on a tablet placed on the chest of a robot. The buttons must be clearly visible, and the approach must be intuitive and clear. Since these are interactions on the tablet, it was thought that they should be as essential and quick as possible, thus without any manual text input. Therefore, it was decided to reduce user interactions to the exclusive pressing of buttons of different types.

4.2.2 From Logic to Interface: Java Files and Layouts

The application manages the communication between the server and the View, where the View only represents the interface reflecting the data managed by the Java classes. The structure of the app follows an architecture close to the Model-View-Presenter (MVP), with an emphasis on separation to manage the communication with the server via a dedicated class, the *WebSocketClient* class.
Next, it will be described what can be seen sequentially while using the developed application.

[2]Typically, this kind of application that works with the Pepper robot exploits the *RobotActivity* class from the *QiSDK*, the library provided by Softbank Robotics

[3]Square, Inc. is an American financial services and mobile payment company, known for its innovative solutions that help small and medium-sized businesses handle financial transactions more efficiently.

WebSocketClient
Before describing what will be displayed, it is necessary to portray the fundamental class that realizes the communication with the robot, because without it, there would be no point in the whole application.
WebSocketClient is an essential class for managing communication via socket in the app. It exploits the *OkHttp* library, implements the Singleton pattern, handles app-specific tasks, and processes messages in real-time to update the status of the application, making a smooth interaction with the server. The Singleton approach ensures that there is only one instance of *WebSocketClient* throughout the application, avoiding multiple creation problems and allowing centralized access from anywhere in the app. It ensures proper management of activities (like *MainActivity* class, *FormActivity*, and all others) by passing and using appropriate references within the respective socket callbacks and other methods.

Home screen: *MainActivity* class
When the application is launched, this is the first activity that is executed, as it is defined in the manifest file, executing the related graphic layout. The layout has an essential appearance, showing a greeting test and a single button that starts the robot's greeting speech and animation. Pressing the button on the main view will move to the next one, showing the first page where the users can fill in the tour preferences.

This activity is designed to be part of an Android application that interacts with a *WebSocket* server and uses advanced features for user interface management. Briefly, the main functionalities of the implemented *MainActivity* class are the following:

- User Interface Setup: Configure the user interface of the *MainActivity*, hiding the status bar and bottom bar for a full-screen experience.
- Server connection: Initialises and manages a *WebSocket* connection to a remote server in a separate thread. Use this connection to send and receive messages from the server.
- Navigation between activities: Configure a button that, when pressed, starts a new activity (*FormActivity*). In addition, it sends a message to the *WebSocket* server, thus to the robot, when the button is pressed, to initiate the motion and speech procedure with natural body movement.
- Lifecycle management: contains methods to handle the resources when the activity is destroyed. These methods are essential when the activity is no longer needed, as the application risks memory overflow and may crash.

Form Filling: *FormActivity* and *TourMode* classes
The *FormActivity* class allows users to configure tour settings (number of participants

and speed), displays formatted descriptions of tour modes, and sends these configurations to the *WebSocket* server. When the start button is pressed from the main activity view, the control is transferred to the *FormActivity* class, which takes care of acquiring information about the number of participants and the mode of the tour: slow, normal, or fast. This information is forwarded to the server, which takes care of scheduling the tour according to the user's choice. According to the choice made, a different view will be shown: if the user chooses a fast tour modality, will be able to select every visitable work; if he chooses a normal tour modality, he will be able to choose only the works defined as "secondary" or "additional", with all the main works pre-selected by default; if the user chooses a slow tour modality, will see all the works in the museum without being able to make any customization.

In terms of GUI, it is presented with two essential *CardViews*[4] horizontally positioned side by side and a button.

On the first *CardView*, the number of participants is selected. Using a convenient interaction, it is possible to increase and decrease the number of participants by employing an increase and decrease button on either side of the real-time updated amount of participants.

The second *CardView* presents the tour modality choice: Fast, Regular, or Slow. It has been customized so that the radio button appearance looks essential, modern, and also coherent with all the button appearances in the app view. This type of control is called "segmented control" or "toggle button group". The control appears as a single semi-transparent rectangle with rounded angles and is made of three adjacent elements, placed horizontally, representing the "Fast", "Regular", and "Slow" options. The selected button also has a rectangular shape with rounded corners but a darker background while the unselected buttons have a lighter background and are uniformly colored with the entire rectangle which groups all toggle buttons. This toggle button group works like a traditional radio button group, allowing a mutually exclusive choice, but with a more visually engaging and touch-friendly interface, perfect for touchscreen device applications. Also, according to the choice made by the user, it will pop out a brief description of the chosen modality under the toggle button group: for the "Fast" modality, it is possible to select each work individually to be added to the desired tour; for the "Normal" modality, all the main artworks are selected by default, thus it is possible to select only the so-called "secondary" works to be added to the tour; for the "Slow" modality, is not allowed to choose any artwork since it is intended a complete tour, thus encompassing of all available artworks of the museum.

Finally, centrally at the bottom, there is a button that leads to the next page,

[4]A CardView is a visual container that presents information in a card-like layout, with rounded corners and a customizable shadow. These are widely used to present information in a visually attractive and organized manner, often in lists or grids of elements.

controlled by the *TourMode* class, which represents the second and most important user choice: the selection of works to visit.

As said, in the following view, the users are allowed to select the artworks for the customized guided visit. According to the chosen modality, it will be handled in the proper layout, thus allowing or not the selectability of the artworks. The multiple selection of the works is realized by small and intuitive elements, the so-called Chips, widely used in the modern GUI. They will be more detailed defined later in the description of the related layouts. According to the chosen modality, some Chips will be or not be selectable.

The selected works, with the eventual pre-selected ones, are sent to the server to compute the proper plan. The order of the artworks received at the end of planning represents the exact order of artworks that the users will visit through the tour. While waiting and executing the periodic check on the reception of the artworks list from the server, a loading view is shown, displaying a standard dynamic circular bar, at the end of which the robot will proceed with guiding users to the desired visit, starting with the first work of the plan.

As on the previous page, even here there are two *CardViews*, one containing a *ChipGroup* for the main works and the other for the secondary ones. A *ChipGroup* is a container, a component of the Android user interface, used to organize and manage a collection of Chips, which is a compact element that represents an input, an attribute or an action. Typically, a *Chip* has an oval or rectangular shape with rounded corners, thus a button appearance, and it can contain text, an icon or both: it is a highly customizable element in terms of style and behaviour. Like a button, it can be clicked, selected or removable. It is used to represent tags, contacts, filters, rapid actions, and in the case in point each represents an artwork that can be chosen for the visit. Hence, the *ChipGroup* automatically organize the *Chips* in horizontal or vertical rows and handles the flow of them in it, like the spacing or spanning between them. The *ChipGroup* can handle a single or multiple selection of the *Chips* in it, and in this case collect the multiple selections of all the works made by the user, captured by the related listener in the associated class. *Chip* and *ChipGroup* offer an effective way to present options in a compact and visually appealing way, thus managing multiple selections intuitively through a dynamic and modern interface.

As already mentioned, there are three different layouts, shown according to the chosen mode. The layouts associated with normal mode and slow mode represent small variations from fast mode, as they impose constraints on the possibility of selecting or not selecting certain works: in normal mode, the works selectable by the user are only the secondary ones, as the main ones are selected by default; in slow mode, the *ChipGroups* are presented as a non-clickable list and they represent the complete list of works in the museum and which will be visited, without therefore the possibility of deselecting them.

Artworks
To follow, all the class activities that handles the artworks visualization and the interactive elements. There are three types of interactions: through a button that brings the users to the next artwork; through a button that allows to interrupt the tour and leave the museum at any time; through the in-depth analysis buttons. With regard to the latter, the user can select one or more elements for which is interested in learning more about the description just provided by the robot; hence, by clicking on one or more ImageButtons, on which a cropped detail of the work is depicted, they will be described by the robot vocally.

Once the robot finds itself in front of an artwork, it will display a framed[5] representation of it on the tablet screen, surrounded by *ImageButtons* depicting certain details of the work. An *ImageButton* is equivalent to the classic button, on which, however, an image can be loaded. The application offers a unique interactive experience for exploring the artworks: when the user touches an *ImageButton* depicting a specific detail of the work visited, the robot immediately activates a targeted explanation, providing detailed and relevant information on the particular element selected. In this way, after listening to the robot's full explanation, users can delve into only those aspects of the work that capture their interest, transforming passive viewing into active and personalised exploration. Also, each *ImageButton* clicked will highlight a small dot on the complete and framed representation of the work, in order to better locate the detail on the work. All details are located by a semi-transparent dot, but the one selected will be highlighted. The combination of visual interactivity and audio explanations creates an immersive learning experience, enabling a deeper and more focused understanding of the various elements that make up the artwork.

The layouts of eight works are implemented, i.e. the entirety of the main works and two secondary works. Each layout was developed to be replicated and easily adapted to new works, but for demonstration purposes, they were reduced to those presented. The layouts realised are therefore of the following works:

- The Birth of Venus - Sandro Botticelli
- The Kiss - Gustav Klimt
- The Last Judgment - Michelangelo

[5]The customized frame is implemented via code and is designed to make the work stand out, automatically adapting to its shape and framing it with shapes, colours and shadows, typical of wooden frames.

- Mona Lisa - Leonardo da Vinci
- The Persistence of Memory - Salvador Dalí
- The Scream - Edvard Munch
- The Starry Night - Vincent van Gogh
- The Supper at Emmaus - Caravaggio

As previously mentioned, once satisfied with the experience of a work, it is possible to interact with two buttons, one to proceed to the next work and one to end the tour early, taking users to the exit and showing the final screen. It should be noted that if the last work is encountered, both buttons will lead users to the exit.

End view
At the end of the tour, a text will be displayed on the screen indicating the end of the visit and an invitation to leave an evaluation of the experience in the form of stars, where one star represents the lowest rating and five stars represent the highest rating. The rating score will be forwarded to the server and the robot will react according with the rating received, in particular when the highest rating is left, an animation will be performed on the screen and the robot will react enthusiastically.

4.3 Planner

The planner, as specified in the previous chapter, has been developed in Python3 and run on our local machine. The communication between the planner and all the other components will be better described in the following section.
To implement our planner, we relied on 4 main libraries, each of which allowed us to execute our Python code as a PDDL planning problem and find a feasible solution.
The first one is Pyperplan, a lightweight STRIPS planner written in Python that implements several classical planning algorithms and various heuristic functions that can be used with its search algorithms to improve planning efficiency. It also supports PDDL.
The second is the PDDL framework itself, which is a family of languages that allows us to define a planning problem. Each PDDL file is characterized by two main components: a domain file, in which the general properties and actions of a planning problem are defined; and a problem file, which specifies the initial state and goal state for a particular instance of the planning problem.
The key components of this language are: types, which define the types of objects in the domain; predicates, which describe the properties and relationships between objects; actions, which are the actions that can be performed, including their parameters, preconditions, and effects; objects, specified according to the particular problem instance;

initial and goal state that represents the initial and desired condition respectively at the beginning and the end of the execution of the plan.
Eventually, we used a unified-planning[fast-downward] library, which is the integration of the Unified Planning framework with the Fast Downward planner. The fast downward planner was described in detail in Section 2. Unified Planning, instead, is a Python package that provides a simple API for defining planning problems and running planners. It offers a common interface for various planning algorithms and tools. Its main objective is to allow, therefore, a general description of a planning problem that is independent of the used type of planner.

4.4 Server - Client block

The pivotal element that allowed us to make the robot, its tablet, and the planner communicate is a Server-Client communication block. We developed a server that we ran locally on our computer that mainly served as a hub and two clients that ran on the pepper docker: Pepper and Tablet.
We based our clients and server on the tornado Python library, which is an asynchronous networking library. It is designed to handle simultaneous connections efficiently by employing a non-blocking network I/O model.
In our code, we defined a WebSocketClient class, which is an essential class for managing WebSocket communication in the app. It uses the OkHttp library, implements the Singleton pattern, handles specific app activities, and processes real-time messages to update the application's state, enabling smooth interaction with the server.
The Server was developed in Python3 and it is at the heart of the Human-Robot Interaction we developed. It is characterized by a block of functions that is responsible for receiving messages from one client and forwarding them to the other client, allowing an exchange of important information.
Specifically, the messages it receives are a *"Start tour"* message, which indicates that the user has started to use the tablet. A *"Participants"* message, that passes to the Pepper client the number of people that form the group of visitors and that will be used when generating a probability of having someone that is too close to a painting as previously described; both the previous message are sent by the client Tablet and passed forwarded to the client Pepper. A *"Selected works"* message, that indicates the painting the user has chosen to visit, is sent from the Tablet client and not forwarded to Pepper; specifically, this message is used by the server to obtain a list of strings containing the paintings the user selected; in addition, the server creates a separate thread in which the planner will run; once the actions to execute will be computed, the server will forward the message *"The actions to execute are"* as a string to the Pepper client. A *"The order of the painting is"* message, sent from the Pepper to the Tablet client, that contains the order of the painting the robot will visit, inferred

from the sequence of action the planner computed. A *"Ready to start visit"* message, sent from the Pepper to the Tablet client, when the robot has arrived at the painting and the action to execute is to visit the paint. A *"Next"* message, is sent from the Tablet to the Pepper when the user decides to pass to visit the following paint and therefore clicks on the "Next" button of the tablet. A set of possible messages will be sent by the Tablet to the Pepper if the user has clicked on a specific detail of the paint in order to receive a detailed description of it. A *"Ready to start visit"* message, sent from the Pepper to the Tablet client, when the action to execute is to visit the paint. A *"Tour finished"* message, is sent by the pepper robot to the tablet when all the actions planned have been executed. A *"Stars"* message sent by the tablet to the robot containing the number of stars received as an evaluation given by the user to the robot. A *"Exit"* message received from the Tablet client if the user clicks on the End button of the tablet, deciding to end the tour before every painting has been visited; the server will receive from Pepper client the room in which the user is when he clicks on the end button and create a new thread in which a second planner will receive as input the current room and compute the fastest path to reach an exit of the museum. A *"Someone took a photo with flash"* if it realizes the simulated event in which someone took a photo with a flash when it is not allowed; the message is received from the Pepper robot and forwarded to the tablet.
The client Pepper is the client that we used to receive information from the Server as a string message and use it to infer important data that will be used by the Pepper robot while executing the code. In particular, the Pepper client presents a set of flags defined as a False at the beginning of execution of the code that will turn True when a specific message is received. The specific function of each flag has been previously described in 3.1.1.
The Pepper client is also responsible for sending messages to the Server, for instance when a specific random event occurs as commented before.
Eventually, it has the pivotal role of extrapolating the actions to execute from the relative Server message, transforming them from a string into a list of strings, and passing them to the Pepper robot by altering the value of the attribute "actions" of the PepperRobot class.
The last client we implemented is the Tablet's client, fully described in 4.2.2.

5 Results

In the following lines, we will describe briefly the four scenarios presented in the realized video, analyzing the different behaviors of the robot and highlighting how different interactions lead to different behaviors.
In particular, the four parts that constitute our video differ mainly in four different aspects: the type of selected tour, the rising of different random events, the selection of the artworks, and the final evaluation given by the user. Moreover, in the second and third scenarios presented, we decided to terminate the tour earlier than its natural end.
The initial interaction is, instead, common to all four analyzed cases. Please note that the video was accelerated to reduce its duration.
In the first scenario presented, we decided to select a fast tour mode, choosing two paintings to visit, namely *The Last Judgment* and *Supper at Emmaus*. The robot proceeds to the room where the first artwork is and, after a first explanation, asks if there is any further information that the user wants. We decided to show a single detail description for each paint analyzed in this scenario to show how this interaction works generally; the user clicks on the detail itself in the robot's tablet starting, therefore, the explanation of the related detail. The same operation is repeated with the second artwork; eventually, the robot leads the visitors to the nearest exit and asks for an evaluation. We gave the robot five stars, the maximum possible value for it, thus starting the correspondent robot's reaction.
We can see that while passing from one room to another, a loading screen appears on the tablet. No random events are generated in this scenario.
In this case, we can see that the planner used is just one; it computes the order of the paints to visit and outputs a set of actions for the robot to execute in order to solve the task of visiting all the selected paintings and reaching the closest exit.
In the second case, we decided to select the normal tour mode, which means that only the main paintings were the ones we wanted to visit.
We asked for two further details on the first visited paintings, the *Mona Lisa*. The robot then proceeds to visit the second artwork, but encounters too much queue and decides, therefore, to proceed to the following one.
After visiting three paintings, we decided to stop earlier the tour: we clicked on the *END* button of the robot's tablet to terminate the tour.
In this second experience, a first planner is used to compute all the actions the robot must execute to visit all the paintings and reach the closest exit, as in the first scenario. However, after the *END* button is clicked, a second planner is activated; it receives the room where the visiting group currently is and outputs the actions to execute to reach the closest exit. This new set of actions overwrites the previous one.
To conclude this second scenario, we gave a three-star evaluation to the robot, activating

a second type of final interaction.
In the latter scenario, we decided to select a slow tour mode, including therefore all the most important paintings plus just one of the secondary ones. In this case, we decided once again to conclude the tour by clicking on the robot's tablet *END* button after visiting two artworks.
Similarly to the previous case, both the 2 planners are used.
The final evaluation given here is only 1 star so that the last possible final interaction can be shown.
In the last case, we intended to show only how the robot reacts when random events are activated. For this reason, we decided to select one painting, namely *The Kiss*. The robot tells the visitors not to take any photos, reproaches the group when a photo with a flash is taken and asks to move away from the painting when a visitor is too close to it.

6 Conclusions

In this project, we developed a highly interactive smart agent based on the Pepper robot software tools and using a PDDL-based planner. Our work has aimed to show how our robot reacts differently depending on the interaction it has with a human user.
We simultaneously worked on the interaction and planning part, trying to generate a complex yet user-friendly behavior. The development of the tablet interface has given our work an important boost to the interactive capacity of our model, enabling users to easily follow the instructions of the robot and insert their choices without any complication.
The planner is instead characterized by a guaranteed optimality, meaning that the path it will find will always be optimal. We added random events and a fictitious museum map to complicate the problem and get as close as possible to a real-case scenario. A possible drawback of our planner is the necessity of acquiring all the information offline; this choice has been dictated by the impossibility of using the simulated robot's sensors. We added random events to try to obviate this issue, even though we deem that further work, possibly with an online planner, is desirable.
The speaking and moving actions the robot performs have been generated in order to favor a human-like reaction of the robot to human inputs as well as give him a sort of self-awareness regarding the environment it is in; for instance, when the action it has to execute is the "move" action, the robot knows which orientation it has to assume.
All in all, we retain that the project gave us a deep knowledge of the Pepper software tools as well as improved sensibly our PDDL programming capacities. Using and developing the robot's tablet interface boosted our abilities in Java programming language, giving us the possibility of concretely seeing and testing the results of our work. Observing different reactions of our agent depending on the input it was given has been a compelling task. We retain the adoption of a Server-Client communication fashion an interesting option to fully leverage the tools furnished by the Pepper robot, allowing a high level of interaction between the users and the agent.
Throughout the development of our project, we identified various possibilities and details that could be added to our work: further studies may include using a physical robot to leverage its sensors and overcome the simulation of random interaction; different random inputs that are coherent with a museum environment can be added, bringing our work as close as possible to a real-case. The plant of the museum itself can be modified, making it closer to the one of a real museum. We also may modify how the robot manages the "is_queue" event, for example considering the paint in which there is too much queue as visited for a certain period and then again not visited; or, we can add "queue sensors" in each painting so that the robot knows when it is the right time to visit each artwork. In addition, possible improvements may include

the development of a tablet page for all the artworks, completing, therefore, the list of paintings available and visitable. Or, we can improve the robot's reaction to the alerts, by modifying the motion and speech of the robot and opening a specific tablet page that varies according to the type of alert that activates. We also may add a loading page for the tablet that would activate when we pass from one room to another; the page would show the current position of the robot in the museum. Eventually, we may also include a pause functionality that would activate when the visitors are too tired and decide to take a pause; in this case, we may modify the museum map by adding some resting zones or even benches where users can sit and rest; the Pepper guide would add to its planner these components and, on the user's request, interrupt the visit and conduct the visitors to one of these elements. After the user clicks on resuming the visit, the robot will restart its previous plan.
In any case, we think that our project can be a solid and complete base to further develop both the planning and interactive capabilities of the robot, making it ready to be used in a real-case scenario.
To conclude, we can state that this project has given us a solid basis for programming a human-robot interaction and a robot planner; our work has proved extremely interesting, as the various details we inserted in our work testify. We look forward to working on stimulating and more complex projects in the area of HRI and RA.

References

[1] Claudio Germak, Maria Luce Lupetti, Luca Giuliano, and Miguel E. Kaouk Ng. Robots and cultural heritage: new museum experiences. *Journal of Science and Technology of the Arts*, 7(2):47–57, Jul. 2015.

[2] Wolfram Burgard, Armin Cremers, Dieter Fox, Dirk Hähnel, Gerhard Lakemeyer, Dirk Schulz, Walter Steiner, and Sebastian Thrun. The interactive museum tour-guide robot. pages 11–18, 01 1998.

[3] Sebastian Thrun, Maren Bennewitz, Wolfram Burgard, Armin Cremers, Frank Dellaert, Dieter Fox, Dirk Hahnel, Charles Rosenberg, Nicholas Roy, Jamieson Schulte, and Dirk Schulz. Minerva: A second-generation museum tour-guide robot. volume 3, pages 1999 – 2005 vol.3, 02 1999.

[4] P. Trahanias, W. Burgard, A. Argyros, D. Hahnel, H. Baltzakis, P. Pfaff, and C. Stachniss. Tourbot and webfair: Web-operated mobile robots for tele-presence in populated exhibitions. *IEEE Robotics Automation Magazine*, 12(2):77–89, 2005.

[5] I. Macaluso, E. Ardizzone, A. Chella, M. Cossentino, A. Gentile, R. Gradino, I. Infantino, M. Liotta, R. Rizzo, and G. Scardino. Experiences with cicerobot, a museum guide cognitive robot. In Stefania Bandini and Sara Manzoni, editors, *AI*IA 2005: Advances in Artificial Intelligence*, pages 474–482, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[6] Francesco Del Duchetto, Paul Baxter, and Marc Hanheide. Lindsey the tour guide robot - usage patterns in a museum long-term deployment. In *2019 28th IEEE International Conference on Robot and Human Interactive Communication (RO-MAN)*, pages 1–8, 2019.

[7] N. Hagita, M. Shiomi, T. Kanda, and H. Ishiguro. Interactive humanoid robots for a science museum. *IEEE Intelligent Systems*, 22(02):25–32, mar 2007.

[8] SOAR. Pilot pepper robot program evaluation, 2019. Accessed: 2024-07-12.

[9] Stefano Rosa, Marco Randazzo, Ettore Landini, Stefano Bernagozzi, Giancarlo Sacco, Mara Piccinino, and Lorenzo Natale. Tour guide robot: a 5g-enabled robot museum guide. *Frontiers in Robotics and AI*, 10, 2024.

[10] Tate. After dark, 2014. London, UK: Tate Britain. Accessed: 2024-07-12.

[11] M. Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, July 2006.

[12] L. Caracciolo, A. de Luca, and S. Iannitti. Trajectory tracking control of a four-wheel differentially driven mobile robot. In *Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No.99CH36288C)*, volume 4, pages 2632–2638 vol.4, 1999.

[13] Sertac Karaman, Matthew Walter, Alejandro Perez, Emilio Frazzoli, and Seth Teller. Anytime motion planning using the rrt*. pages 1478–1483, 06 2011.

[14] Daniel Alves Barbosa de Oliveira Vaz, Roberto S. Inoue, and Valdir Grassi. Kinodynamic motion planning of a skid-steering mobile robot using rrts. In *2010 Latin American Robotics Symposium and Intelligent Robotics Meeting*, pages 73–78, 2010.