HRI & RA Project Report

Andrea Bonaiuti mat. 1618825

Contents

1 Introduction

Nowadays, the presence of informatics and even Artificial Intelligence is usual in a lot of environments, from industry ones to study and research, even in homes for assistance goals.
In the last decades the attention to increase this presence and the aid that artificial intelligence can give to us has led to the development of robotic systems, able to do physical labour instead of humans or as just interactable interfaces that are able to reason and reduce the load of work and reasoning of humans.
As shown in Figure 1, these robotic systems have spread in like every field of study/work/commerce and are coming in different shapes and logics. As robotic arms [1] for industrial purposes, accelerating and automating assembly lines. As humanoid robots [23, 12] that could reach all human motion capabilities and will find utility in like every kind of task, at least all task feasible for humans, from military purposes to even some product used as waiter/waitress in restaurants.
And even four-legged and animaloid ones has been developed and research to study and learn locomotion challenges and pros and cons of different locomotion structures.
Robotic and autonomous systems has found also successful application in space exploration of plane-

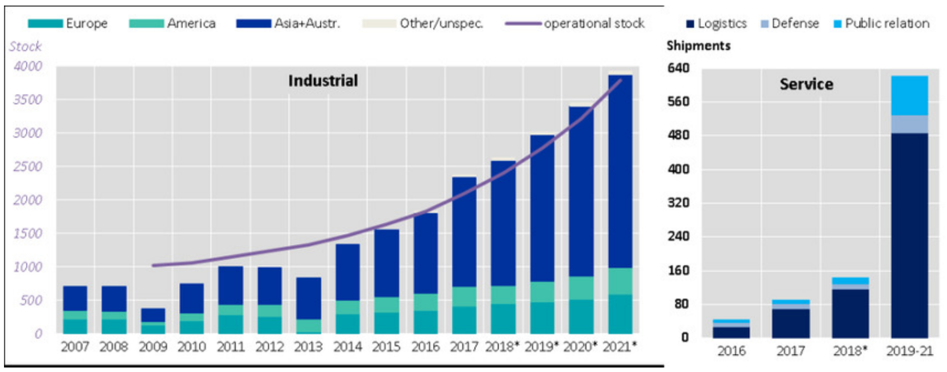


Figure 1: The growth of stock of industrial and service robotic systems in the latest years. Asia seems to be the leading region on this sector. On the right side, it's shown that logistic purpose is the most common.

tary surfaces. Systems like orbiters, landers and rovers have been used to substitute human exploration in those dangerous environments being able to retrieve important images, geographical and biological information of the moon and mars [32], making possible the research of sign of ancient microbial life [33].
Given that take off and flying are well studied task and can be planned and controlled by human teams, the first task for an autonomous system is the landing on a surface.
The landing should be as smooth as possible to not harm the system architecture, should be precise to land on a selected place, that should be selected for a safe landing (on a flat surface for example).
Autonomous landing has been developed mainly through deep learning approaches and reinforcement learning techniques, using simulation environments to train these systems in order to make them able to behave correctly in real life scenarios.
This report describes the structure, details and development process of the project for Human-Robot Interaction and Reasoning Agents module of the Elective in Artifical Intelligence project.
It consists in a framework for interacting and planning through a simulated agent/robot in the Lunar Lander environment offered by OpenAI's Gymnasium, an API standard for reinforcement learning with a diverse collection of reference environments.[8]
The user can interact with the agent, called LuLa, through voice or text in a terminal, asking information about what it can do, find what pure planning can achieve in the lunar landing task, plan new task defined by the user, learn the history of Apollo 11 while waiting for the plan to be finished and also play a minigame to learn the challenges and difficulties that an autonomous robot faces during landing.
At every interaction the robot state updates, but the user is able to come back and revert all of its progresses at any time and restart with different options.
Lula essentially uses available plans built through different discretization logics, runs a simulation in

the Gymnasium environment and collects data from it. Simulation information can be watched and retrieved by the user to know how well that plan performed.
The report continues with an overview of the code organization and its flow of information, then with some implementation details, used libraries and models. At the end some discussion about the development, with limitation and challenges faced and some solutions/workarounds.
This project could be an interesting basis to learn how planning can be used and how much it is able to adapt in continuous environments, through a system that can iteratively run simulations guided by policies made through planning.

Figure 2: Some example of different types of robotic systems, from humanoid ones (upper-left), drone (bottom-left) to robotic arms used for surgery (second from bottom-right).

2 Related Works

In this section some paper will be cited as relevant and related to argument covering the project.

2.1 Human-Robot Interaction

Human–robot interaction (HRI) is currently a very extensive and diverse research and design activity. The literature is expanding rapidly, with hundreds of publications each year and with activity by many different professional societies and ad hoc meetings, mostly in the technical disciplines of mechanical and electrical engineering, computer and control science, and artificial intelligence.[44]

2.1.1 Teleoperators and space

The robotics research started with the need of substituting human performances: for example, in environments that are dangerous to human utilizing a machine that removes the exposure of an operator to radioactive areas or spatial ones.
In the late 40s, Raymond Goertz at Argonne National Laboratory built master–slave remote handling devices by means of which one could grasp and move objects in all six degrees of freedom [14, 49]. In early works, human operator's master control and slave arm-hand framework was built on top of mechanical tapes, but later this technology was substituted by electromechanical servomechanisms with force feedback.
A first example of that was given by Goertz [26] and used by Erst [18], which is recognized as the first computer-controlled robot. Other early works on human-robot supervisory control are the ones by Brooks and Ferrell [10, 19].
Lots of progresses were made also in remote control of spacecrafts [42], undersea vehicles [45] and unmanned aircrafts [39]. Research and progress continued with the focus on improving and simplifying control interfaces, dealing with the need of complete observations of the remote environment and compensation for intermittent communication delays and dropouts.
The first robotic system that successfully operated on space terrain was a scoop on the Surveyor 3 lander launched to the Moon in 1967. Several subsequent missions were launched with different robotic systems implementing different levels of autonomy, from teleoperation fully controlled by human to fully autonomous operation by robot.

2.1.2 Dialogue Management

Spoken communication is a natural way of interacting between human, and the dream of a human-like robot needs a similar way of communication between the human operator and the robotic system. Social robots are specifically designed to interact with their users by using spoken dialogue.
This robotic skill is a crucial role for elderly care [9] and education [5].
Dialogue Management (DM) is the part of the dialogue system that does four main functions [48]: 1) maintains dialogue context, 2) includes context for interpretation of input, 3) selects timing and context of next utterance and 4) coordinates with other non-dialogue modules.
Dialogue management research started in the 60s dividing approaches in handcrafted, probabilistic and hybrid. In Handcrafted approaches, rules and states are defined by developers, finite-state machines (FSM) refer to systems where states directly correspond to the dialogue states like [43] who focuses on learning tasks through dialogue interaction, or [28] that implemented different FMSs systems for different offered interactions.
Other examples are frame-based systems that use empty slots, typically information units that can be filled at any point of the conversation [7], a similar work is done by [52] where slots are filled to determine user's food preferences in an interview conversation.
Frames with slots can be more complex including some form of model, for example about user, environment, situation or context. Examples of so called model-based can include a model of conversation history for long-term dialogue like Cobot [41] to avoid repetitive dialogue.

Similarly, [30, 31] use a stack to keep track of unfinished conversation sequences for tracking the context in the conversation. Other techniques born as model-based treat the conversation as a planning problem trying to reach a predefined goal in a selected conversation. [3, 20, 40]

2.2 Reasoning Agent

Automated Planning (AP) can solve complex problems in highly structured environments by exploiting models of the agents and their environment. [25]. A plan is an algorithm-like solution valid for a given planning domain/model, an example is the blocksworld domain [46] where the goal is to stack blocks on top of each other in a given sequence.
The task of composing general solutions for complex problems has been studied since the start of AI. In recent years new formalisms and new algorithms for planning were proposed, revealing the potential of techniques from generalized planning and suggests the application of these techniques to multiple fields of research.
The Planning Domain Definition Language (PDDL) [2] is the standard input language for classical planning instances and used for the International Planning Competition [16]. This language has been modified and updated in several versions to be more expressive and allowing introduction of different domain properties like temporal planning [22] or path constraints [24].
In addition with the classical definition of actions, that includes *preconditions* and *actions*, conditional effects [35] can be introduced and their support are required since the International Planning Competition of 2014, adding the keyword *when* after an action, specifying a condition (it can either be a negation, conjunction or disjunction) and the relative desired effect.

FOND planning stands for Fully Observable Non-Deterministic planning and aims to handle the uncertainty of the effects of actions. [6] In FOND planning, states are fully observable and actions may have non deterministic effects. Concretely, it adds multiple possible outcomes/effects of actions in the domain, i.e. an action may generate a set of possible successor states.
Solutions for this scenario can be *strong policies* that guarantee to reach the goal in a finite sequence of steps, and *strong cyclic policies* that guarantee to reach only to states from which a goal condition is satisfiable in a finite number of steps [13].
FOND planning can be used as a planning approach for solving trajectories and paths from continuous domains through discretization. Discretizing is what actually makes the need of non determinism. Given that a value in the PDDL domain represents an interval of the real world, the trigger of an action may or may not change the interval in which it is triggered.
However, the use of FOND planning is limited in the literature and sometimes PDDL+ is used [21], an extension of PDDL2.1 that supports the modelling of continuous time-dependent effects.
For example, work from Cardellini and Maratea [11] use it to solve hybrid planning problems to encapsulate different levels of discretisation for agents with very different speeds and trajectories. Similarly, Thomas and Amatya [47] develops a framework for integrating belief space reasoning within a hybrid task planner for autonomous robots operating in real world complex scenarios.
McCluskey and Vallati [34] use automated planning with PDDL+ in the control of traffic flow in large city centre to deal with unexpected events sush as roadworks, closures to generate light signal control strategies in real time.

For what regards FOND planning application, Della Penna and Mercorio [15] discretise and propose a solving algorithm for cost-optimal strong planning on the inverted pendulum on a cart, where the goal is to bring the pendulum to equilibrium using a minimum amount of force applied to the cart. As stated before, the literature over the use of FOND planning is very limited, researchers prefer the use of the aforementioned PDDL+ settings, and sometimes upgrading FOND planning with probabilistic planning (PPDDL) [51].
An example of the latter is the work by Eleuterio and De souza [17] that integrates Prob-PRP (an extension of a FOND planner) on ROS systems for visualization, execution and monitoring of unmanned autonomous aerial vehicles plans, discretizing routes between locations of interest with equidistant waypoints, and also fuel resource in fuel levels.
Work is by Konidaris and Kaelbling [27] that enables an agent to autonomously learn its own symbolic representation of a continuous environment and from this constructs a PPDDL representation .
Another worth mentioning work is by Ames and Thackston [4] that parametrize motor skills and sim-

ply discretize abstract representations for planning in PPDDL, applying their method on the famous Angry Birds game.

3 Solution

The solution is mainly composed by an input handler as well as an output one, an inner logic of the workflow and the status of the system, a subcomponent dedicated to the environment simulation part, one that mimics the remote manipulator modality and another for planning the trajectories and parsing the relative policies.
A more detailed description follows in the next subsections and a sketch of the overall architecture is depicted in Figure 3.

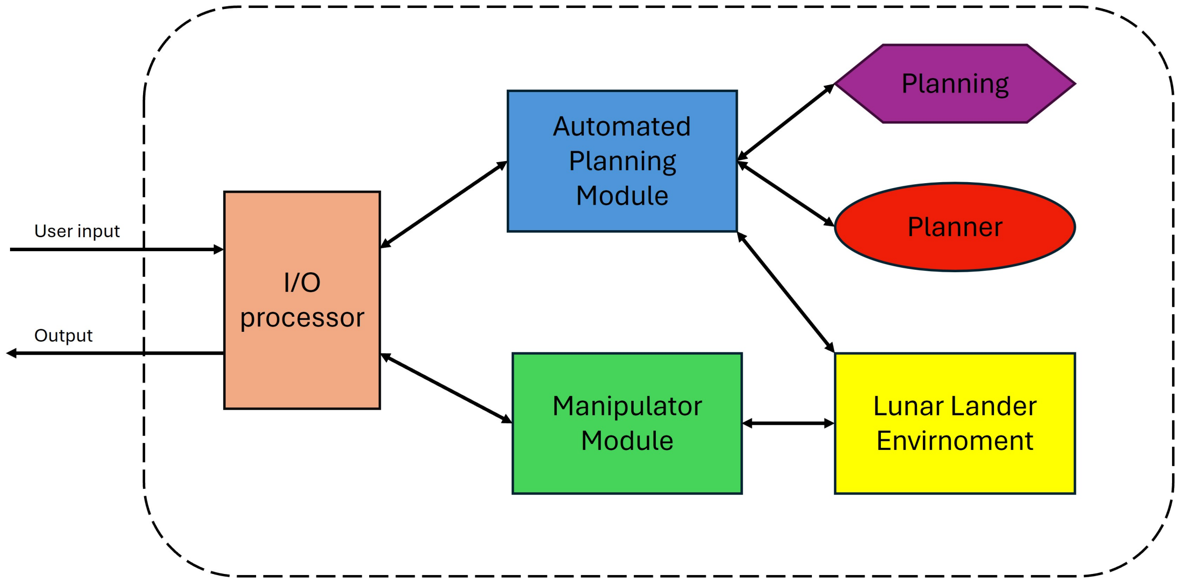


Figure 3: Overall architecture of the solution. The I/O processor is the fundamental block in input and output flows, then the two branches refer to the two modalities implemented for the Lula robot.

3.1 Human Robot Interaction

3.1.1 Input and Output Management

The system can be seen as a Robotic system interacting with the user/human through speech, text and visualization of its environment. As for interacting with it, the user can do that by inputting through keyboard in a terminal (the same terminal used for text output), or by voice.
The input and output flows are basically alternated in a classical sequence made by: waiting for input, computing consequences of input, outputting the result/advances and then waiting again for input.
The same text output written in the terminal is synthesized in an audio and then played.
This behaviour is maintained and controlled by the I/O Processor block of the Figure 3.

3.1.2 Input Interpretation

At every step, the input is interpreted contextually.
This means that, with the use of a pretrained language model for text classification with variable labels, the input meaning is predicted through different labels depending on the current system's status.
The unique always present label is the close/quit one, to enable the user to quit the current task or even the whole system whenever he/she wants.

3.1.3 Logic Modules

The two logic modules are built in the same way, with a sort of handcrafted dialogue management system with a dynamic status that is able to distinguish what to do and the resulting output to communicate.
The evolution of interaction follows a predefined sequence of information requested and/or returned

to the user, but with the possibility of reverting some steps to start again from previous point.
The status of the modules is always recorded to allow the waiting for the next input and to allow also confirmation at every step when the user asks to close the current task or module, being capable of restart from the last thing done and remembering the user at what point he/she was.
For every state, the user can ask for more information on the current task or on the current information requested, from what actions he can order to the manipulator, to the available predefined domains for planning a trajectory.

The **Manipulator** module (green block) in Figure 3 is responsible for implementing the environment in a manipulator modality, the system simulates its environment (Lunar Lander from Gymnasium OpenAI) and waits for the user instructions, like a teleoperator.
The environment is gamificated, not in real time, to teach the user the difficulties and the logic needed for a smooth landing on a surface.
The user is able to select an action (or close the game/task) from the available ones, which are the engines of the lander. A main one and one for each side.
After each action the lander state is updated and visualized. In addition the system outputs via text and speech the current status of the lander and also warns the user for some coordinates not aligned with the safe landing goal. The system is able to warn also for critical velocity values that are to be corrected as soon as possible in order to maintain the goal affordable.

The **Automated Planning** Module (blue block) in Figure 3 is a little more complex module and is responsible for handling the whole planning process.
It can use predefined policies for the landing goal to draw trajectories, simulating them in the environment and stores information about the run. The user can retrieve some information about it, like the maximum velocities or the average ones, how many times the trajectory is corrected (i.e. the side engines are fired) and so on.
In addition the user can ask to plan for some other trajectories that may lead or may not lead to the landing, like flying from the left side to the right side.
In this case the user is asked to select desired starting and ending states, and also to select one of the available domains in which the planning would run.
After the system has all it needs, it starts to plan in the background and while the planning subprocess is running the system can entertain the user in two modalities: 1) by running the Manipulator module to make the user play a game while waiting, in the same modality as the clean run of that, or 2) by telling a story about the Apollo 11 lunar landing.
In both cases, when the planning run is finished, the system alerts the user and runs a final simulation of the newly planned trajectory, storing informations and making them available for user requests as the case of predefined plans.

3.2 Reasoning Agents

The autonomous behaviour of the lander is guided by policies created by planning. The actual environment of Lunar Lander of Gymnasium OpenAI is continuous and this has introduced the need of a discretization over the domain.
The state space of the environment is then divided into values that represent intervals, called bins, for each fundamental dimension (i.e. x and y coordinate, angle w.r.t. ground and the respective velocites).
The action space is a low-level one, so the actions available are the same as the Gymnasium environment. This is because it is not meant to be an abstraction, but only a discretization.
The definition of planning domains has been done in PDDL, making use of FOND planning technique to handle the non-determinism added with the discretization. A main engine fired should increase the vertical component of velocity, but does not guarantee that the resulting new velocity will be included in the next bin of vertical component of velocity.

Below there is an example of what is the main addition of the FOND planning, the keyword *oneof* that enables different possible outcomes. In this case there is also the presence of a conditional effect to determine the environment dynamics beneath the highlighted action.
As we can see, in the *oneof* statement the *(and)* effect means concretely no effects and represent the case in which the action outcome does not provide a change of the discretization bin.

```
:effect (and
        (oneof (and (not (current_vy ?vy)) (current_vy ?vy_prev)) (and))
        (when
            (positive ?vy) (oneof (and (not (current_y ?y)) (current_y ?y_next)) (and))
        )
        (when
            (negative ?vy) (oneof (and (not (current_y ?y)) (current_y ?y_prev)) (and))
        )
        (when (current_y y_0) (ground))
    )
```

The system comes with three different predefined domains to study the outcomes of different levels of discretization on the simulation and also the different times needed to plan in these domains.
The first and simplest domain is the *novelocity* one, an abstraction in which the velocities are not modeled and the only values are x and y coordinate and angle w.r.t. ground.
In *simplified* domain velocities are modeled but considered as effect only in actions that actually modify them, for example the dynamics of lateral movement and rotation are not considered in the main engine action and for the same reason in the side engine actions the vertical component of velocity is absent.
The *mini* domain is the most complete one, the whole environment dynamics are considered in all actions.
The number of bins is different for all the domains, this is due to their increasing complexity that need to restrict the number of bins (i.e. increase the intervals' range) to make the planning feasible. The novelocity domain has the biggest number of bins due to its simplicity and for the same reason the mini domain has the smallest one.
The planner used for this FOND planning task is PRP Planner for Relevant Policies [37, 36, 38]

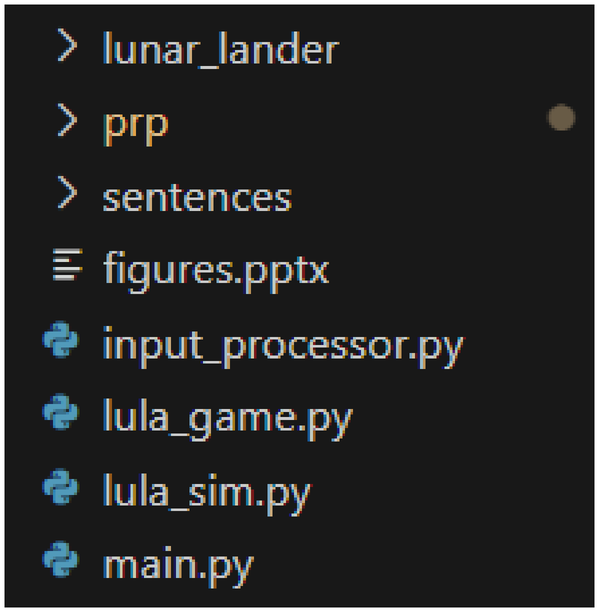
When a policy is available, the Lunar Lander environment is started. At each step the system is able to collect the environment state, search in the policy the corresponding state and fire the corresponding action.
In order to do so the environment state needs to be discretized again to match the bins in the selected domain. For the same reasons as above, different domains need different discretization reasonings.

4 Implementation

The project can be mainly divided in two different sections, the files at the root folder, responsible for implementing the main interface and behaviour of the system, and the lunar lander folder, that contains all the planning domains, the environment wrapper of lunar lander, some useful methods for planning and parsing the policies used for simulations of the environment and some for discretizing the running environment observations to make them match with the bins of the discretized domains and to make the observation vector readable as rules for the policy.
A more detailed description of each component follows.

4.1 Human Robot Interaction

This section will describe in depth the root folder files structure and behaviour that consists for the most part of the Human Robot Interaction component.



```
> lunar_lander
> prp
> sentences
  figures.pptx
  input_processor.py
  lula_game.py
  lula_sim.py
  main.py
```

Figure 4: Screenshot of the root folder of the system, the HRI components are in the 4 python files, a little description will be given for the "sentences" folder.

4.1.1 Main.py

The main file is responsible for starting the system and mantains the main loop for receiving input, passing it to the component that elaborates it and then waiting again for the next input.
It also start with a greetings phrase to welcome the user at its start.

4.1.2 Sentences folder

As previously stated, the dialogue system is an handcrafted one, this means that, if the system is able to predict the intent of the user input in whatever he/she says, the system replies selecting from a set of predefined sentences the most accurate one.
This doesn't happen everytime, there are certain circumstances where the sentence needs to be dynamic and composed in real time. When that's the case, this folder is not considered. On later sections this case will be described.
Additionally, the folder includes a text file with a story of the APollo 11 mission that will be presented in a later section.

4.1.3 Input_processor.py

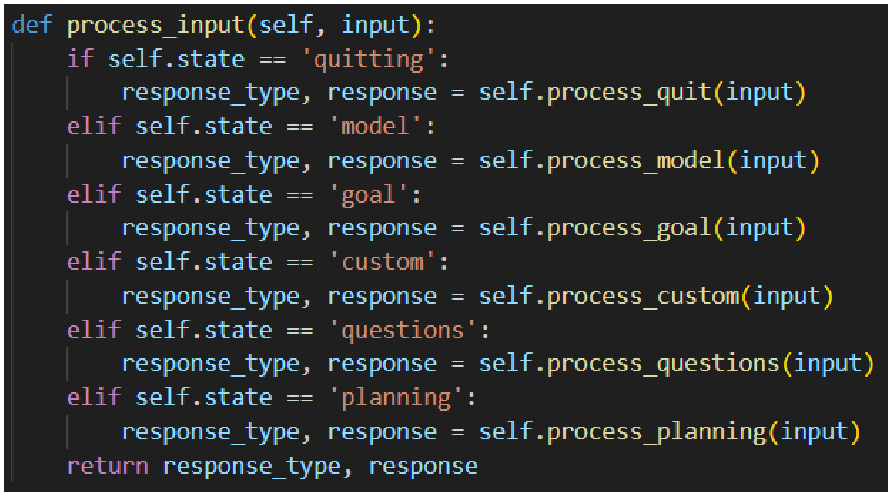
This module is the main file for interaction with the system. The input received in the main.py file is always passed to the IO_Processor class defined here.
It is also responsible for maintain and switch from the two behaviour modalities of the system, the manipulator modality and the autonomous one. The IO_Processor class creates them and starts one of them when the user requests it, and closes the same when asked to do it.
It maintains record of the active modality, called task, and passes to the active one the next inputs.
This class is also responsible for output management. This comes in two modalities with a textual one that is simply written in the main terminal. Then the text, either it be a sentence from the "sentences" folder or a personalized one, is synthetized in an .mp3 file with the *gTTS* library (Google Text-To-Speech) that uses the Google Translate's text to speech API, a simple and light way to produce an acceptable speech from text.
Then the newly .mp3 file is played thanks to the *playsound* library and then immediately removed in order to maintain the code clean and light.
Input interpretation is done firstly in this class. When no task is started, it have to predict what task the user wants to start, if he/she wants more information about them or just quit the system.
This is simply accomplished with the use of BART model [29], a denoising autoencoder for pretraining sequence-to-sequence models, more specifycally the *facebook/bart-large-mnli*, a ready-made zero-shot sequence classifiers [50] that it's a checkpoint from the large version of BART trained on the MultiNLI (MNLI) dataset.
The method works by posing the sequence to be classified as the NLI (Natural Language Inference) premise and to construct a hypothesis from each candidate label. For example, if we want to evaluate whether a sequence belongs to the class "politics", we could construct a hypothesis of This text is about politics.. The probabilities for entailment and contradiction are then converted to label probabilities.
This pretrained model is loaded through the *transformers* library.

4.1.4 Lula_sim.py

This file and its associated class *LulaSim* cover the Automated Planning module represented in the blue block in Figure 3.
After being initialized by the IO_Processor, this class takes the control over the input interpretation, through the same model that is passed to it as the aforementioned class., until it is told by the user to stop.
The evolution of the dialogue and interaction with this class is controlled by a state variable that can have different values for different states of the planning process, other variables can help to record some properties of states like the last state that occurred, used for resuming the task when the user ask to leave but doesn't confirm.

The main method for doing that is *process_input()*, that distributes the input to be interpreted and its consequences to other specific methods as illustrated in Figure 5 Figure 5 is also useful to understand the evolution of the states and the behaviour of the class, even if the *if* statements ae not in perfect order.
The quitting state represent the confirmation state for quitting the current task, it is able to resume from one of the other states when the user changes is mind and does not confirm to quit the task.
The other states represent the steps that the user need to so and the information needed for the LuLa system in order to launch a simulation with a policy to make trajectory informations available for the user.
The user has to specify which domain it want to test and then the goal. The user can select a default goal, an attempt to land smoothly between the given flags, that comes with precomputed policies and does not need a planning phase.
Additionally the user can try to plan for different paths, for example he/she can propose the lander to fly at the same height from the left side to the right side. These specifications are collected in the *custom* state.

If the user requests a custom plan, the system launches a subprocess to run the PRP planner. This



```python
def process_input(self, input):
    if self.state == 'quitting':
        response_type, response = self.process_quit(input)
    elif self.state == 'model':
        response_type, response = self.process_model(input)
    elif self.state == 'goal':
        response_type, response = self.process_goal(input)
    elif self.state == 'custom':
        response_type, response = self.process_custom(input)
    elif self.state == 'questions':
        response_type, response = self.process_questions(input)
    elif self.state == 'planning':
        response_type, response = self.process_planning(input)
    return response_type, response
```

Figure 5: Screenshot of the principal method of LulaSim class. This method makes clear what are the main states of the class during its execution and interaction with the user.

is the *planning* state, where the system waits for the termination of the planning subprocess and in the meantime offers the user two activities for waiting.
The user can "play a game", i.e. run the manipulator module passing the time trying to land manually the lander. A more detailed process will come on the next section.
Alternatively, the user can select to hear/read a story of the first human Lunar Landing mission, the Apollo 11. The story is written on a text file in the folder of sentences cited before.
The story is divided in long periods written in one line each, the system reads and outputs (by both text and voice) one line at a time and after that checks if the planning subprocess is completed.

When the planning phase finishes, the system alerts the user, makes a final simulation on the Lunar Lander Environment and collects data from the run. At this time, information are available for the user and the system enters in the *question* state.
In this state, the user can ask the system some information about the expected performances of the given policy. Examples are how many steps it would take, the average velocities and so on.

This class contains also some dicts useful for translation from user natural language specification of the starting and goal states for the planning phase and for the translation from actions selected by the user to ones understandable for the Lunar Lander environment.

4.1.5 Lula_game.py

This file and its associated class *LulaGame* cover the Manipulator module represented in the green block in Figure 3.
As the previous section, this module takes control over the flow of inputs when it is initialized by the IO_Processor class after the user asks to do it.
The structure is the same as the LulaSim class, with a state variable that controls the evolution of the conversation, aided with some other variable for specifications. It is more lightweight than LulaSim, because there is a less need of user intention in order to launch the environment and apply one by one instructions by the user, that is the main focus of this class.

The user can inform itself about the actions available for him/her and after start of the "game" the system will freeze the environment waiting for the user's next action selection.
Once an action is selected, the system applies it and the rendering of the new actions is updated. In order to make a compromise between game speed and precision of movement, an action selected by the "human operator" is actuated three times in the actual environment.
This is because a run in the lunar lander environment could take hundreds of steps and it would make

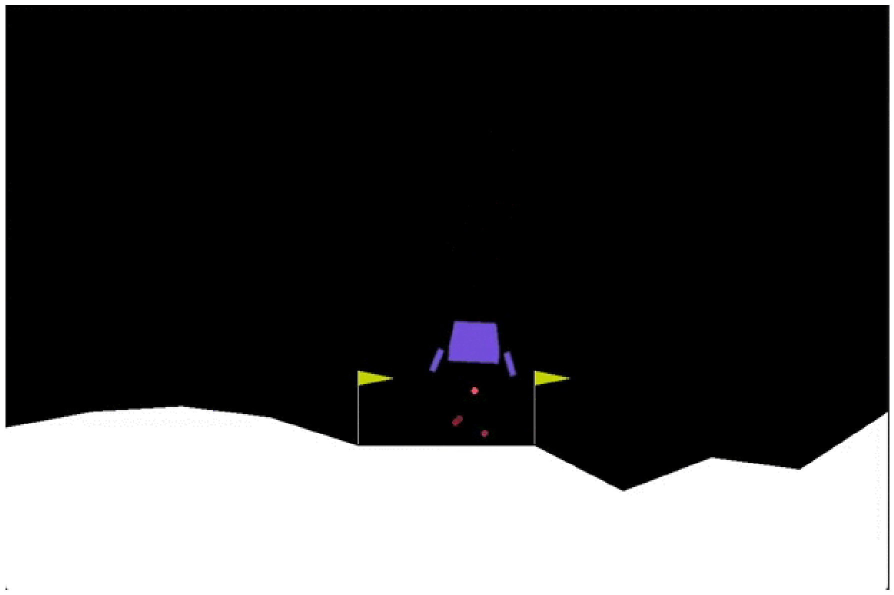
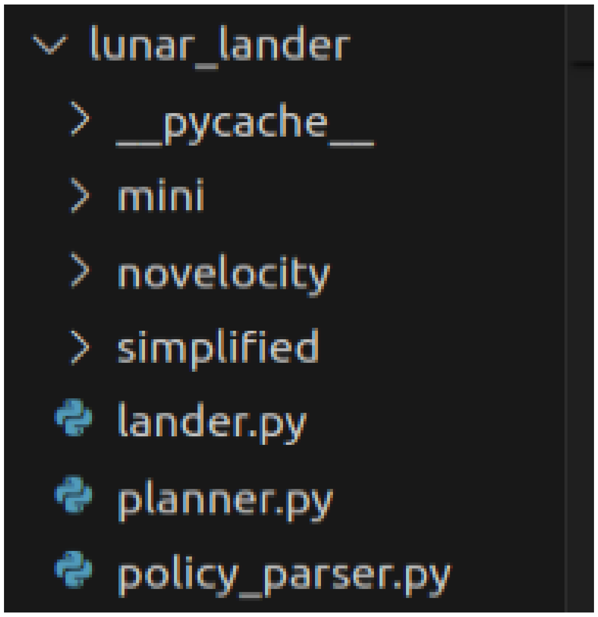


Figure 6: Image of the Lunar Lander environment by OpenAI's Gymnasium. This snapshot can be one of the state reached using the manipulator modality. The red points represent "flames" from an engine fired, in this case the main engine.

it unfeasible and boring with the speed of interaction of the system.
After each action application, the environment returns the newly updated observation vector containing respectively x-coordinate, y-coordinate, x-velocity, y-velocity, angle w.r.t. ground and angular velocity. It also returns a reward used in Reinforcement Learning training and some boolean values used for termination checking etc.
The system, before waiting for the next user action, tells the user the new observation and compares the returned observation with acceptable ranges of values and warns him/her when some of them are not aligned with what should be the final goal, the landing. It also warns the user when velocities are too high and when their value could make the landing impossible.
The actions can be selected by natural language by the user, interpreted by the system's NLI model and then translated in the environment action through the same dictionary used in the LulaSim class.

4.2 Reasoning Agents

In this section will be described the Reasoning Agents component, most of which is in the *lunar_lander* folder, but with some other insights from the class/files described before.



```
lunar_lander
  __pycache__
  mini
  novelocity
  simplified
  lander.py
  planner.py
  policy_parser.py
```

Figure 7: Overview of the lunar_lander folder, the three subfolder contain three different domains for the lunar lander environment, while lander.py and policy_parse.py are utilities for planning and simulating, while the planner.py file were used to plan the default problems.

4.2.1 PDDL Domains and their folders

The three subfolders in the lunar_lander folder contain the three domains written for the discretized lunar lander environment.
These domains are not meant to be an abstraction of the environment, writing actions of high level in order to just represent the semantics of the trajectory, but they should just represent a discretization of lunar lander, a continuous scenario, where values can change by a very little amount from a step to the next one.
The values determining the status of the lander have been transposed to the PDDL domain, converting them in objects, divided by type of different represented value. The entire x axis, for example, that can vary from -1.5 to 1.5 has been divided in different objects of a unique type, in order to not interfere with other observation values' types. So each object/value/bin represents an interval of values on that axis.
The number of object/bins for each value is different in the three domains, given that the complexity of the domains is different and a more complex one needs a reduced number of objects in order to complete the planning process in an acceptable amount of time.
To build a connected and contiguous environment, in order to make the domain able to change from a current value to the next one, fluents that represent contiguity have been added for each direction (i.e. *(next)* and *(prev)* fluents), in addition to fluents that represent the current status for each value (i.e. *(current_x)*).

```
(:constants t_-1 t_0 t_1 - t_value
    vt_-1 vt_0 vt_1 - vt_value
    vx_-1 vx_0 vx_1 - vx_value
    vy_-2 vy_-1 vy_0 vy_1 - vy_value
    x_-1 x_0 x_1 - x_value
```

```
        y_0 y_1 y_2 - y_value
    )

    (:predicates
        (current_t ?t - t_value)
        (current_vt ?vt - vt_value)
        (current_vx ?vx - vx_value)
        (current_vy ?vy - vy_value)
        (current_x ?x - x_value)
        (current_y ?y - y_value)
        (negative ?v - value)
        (next ?n ?v - value)
        (positive ?v - value)
        (prev ?p ?v - value)
    )
```

The above lines of code show the setting of objects and fluents needed to build a domain in PDDL, the one shown above is the *mini* one.

Actions were left the same as of the original environment, that are the actuation of the three available engines of the lander, one for each side and another for the main one, at the bottom of the lander, and the action *idle*, that means to not fire any engine for a step.
For the same need of contiguity specification, parameters of actions are a little verbose, as it's needed to specify, for each current value, the previous and next value/bin.

These domains have been built from the simpliest to the more complex one, to study gradually how much computation was needed for each detail added and for different amounts of bins selected.
The first domain, the simpliest one, is the *novelocity* domain. As its title says, it is a discretization in which linear and angular velocity have not been considered. In this domain, an action could just modify the current coordinate or angle w.r.t. the ground on not. Just simple as it is. Because of this simplicity, the number of bins/object used is large and this doesn't affect the planning time. But at the same time, as we will see in the Results section, plans from this domain have the worst results of all.
The second domain is called *simplified*, this is because is a simplified version of the actual one, but with all the properties it should have (i.e. velocities are included). The simplification from which its name derives its the fact that not the whole environment dynamics are considered at the same time. The main engine action, as well as the idle one, produces effects only in the vertical component of velocity and the y/vertical coordinate itself.
The structure of the action, that will be the same for the side engine actions, is composed by a first effect on the velocity itself (it may or may not change the current bin of vertical velocity) and then, there is a check on the current velocity bin if it is positive or negative (through another fluent that states which objects are negative and which are positive) and if it's the former case, the y coordinate may increase or not, and if it is the latter case, the y coordinate may decrease or not.
The possibility of no effects in the actions is the non determinism addition, a consequence of the discretization. An action in the real environment affects the velocity it is meant to change, but by a little amount, and this amount could result in a new value of velocity that, in the PDDL domain, is in the same bin as the velocity before the action was fired. The change of coordinates follows the same logic.

In the same domain folders, other than the domain, there are two PDDL problem files. One problem file corresponds to the default problem of a safe landing, so an initial state at the center of the environment and a goal state that correspond to the 0 y-coordinate, the 0 x-coordinate (ground), a 0 as angle w.r.t. ground and a 0 as vertical velocity.
The other file is a template problem file that is used for planning custom problems. When the user requests to do it, this file is modified withe the values selected from the user and then fed to the PRP planner.
The computed policy from the planning phase is also written in the same domain folder, in order to distinguish from one domain policy to another domain's one.

4.2.2 Lander.py

this file consists in a Wrapper class for the Lunar Lander environment. It is a class offered by Gymnasium API and it is useful to apply modifications of the environment.
In this case, it was used to change the starting position of the lander, needed when the user requests to plan for a custom problem in which he can select a different initial position.
In order to do so, the *reset()* method of the environment was overridden with modifications on the starting coordinates, depending on the added parameters.

4.2.3 Policy_parser.py

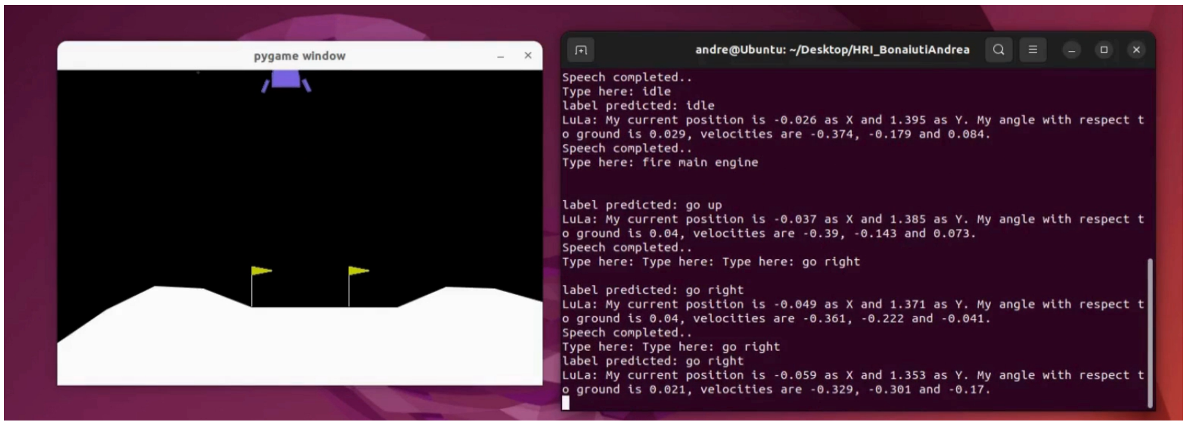
This file builds the logic for writing the policy in a readable way for the system, defining a Rule class that encapsulates states for which an action has been planned.
It is also responsible to discretize the observations from the Lunar Lander environment into the selected domain's bins, to make the observations comparable with the Rules of the policy.
To make the policy more readable from user/developer and also the system, it is saved in pickle and also json format.

4.2.4 Planner.py

This file describes the methodology of planning used for the default goal of safe planning.
It is an attempt to plan for an extended state space. The reason for doing it is that the Lunar Lander environment starts with random forces applied to the lander, that makes each iteration different from others and that can lead to state/coordinates that can be far from the expected trajectory that a simple planning explores.
To build a policy that covers a wider state space, different combinations of starting state have been used for planning the same goal with the PRP planner. Each time the template PDDL problem file has been rewritten and fed into PRP. After the termination of each iteration, the new policy rules were appended on a final policy file that during the execution of the LuLa system is parsed and saved as pickle and/or json file.
This methodology has been adopted for all the three domains in a single loop as can be seen in the code.

5 Results

The project has been developed first in windows/WSL, but due to the audio compatibility not supported by the WSL, it has been moved into a virtual machine with Ubuntu on it.1
This has caused a little loss of computational power, as seen in the video during the background activities. But it was the easier way to have an environment compatible with all the tools and methodologies used.
The system results in a simple one to use and interact with, maybe just a little slow to interact with, this is due to the limited properties of the text-to-speech system: an increased velocity of the speech makes it unnatural and difficult to use as the voice output has been considered the main one and it would be hard to understand. Text output, however, reports any speech spoken and can be visualized for a later read, especially for numeric outputs.
The dealing with the lunar lander environment is a bit complicated and has not been touched, so the window for the lunar lander is just opened when it's needed and it results with the terminal as the main interface to deal with. Pygame is a bit tricky to deal with and due to time limits it has been left as a future work
But the system does what it has been built for. It has a good performance for predicting human intentions, returns all the information needed for the user/operator, plans in the background without noise and interference and interacts well in the manipulator environment in which it suggests in a correct way what to do in order to reach the goal and when the user is near the failure of the task.
Images follow as demonstration of precise cases of the behaviour described before.



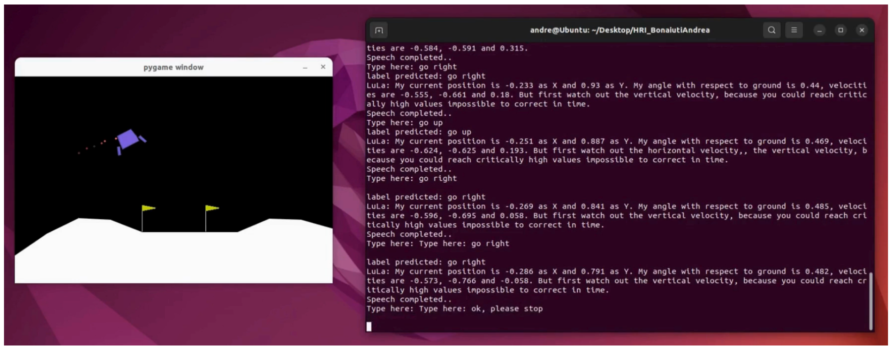
```
Speech completed..
Type here: idle
label predicted: idle
LuLa: My current position is -0.026 as X and 1.395 as Y. My angle with respect t
o ground is 0.029, velocities are -0.374, -0.179 and 0.084.
Speech completed..
Type here: fire main engine


label predicted: go up
LuLa: My current position is -0.037 as X and 1.385 as Y. My angle with respect t
o ground is 0.04, velocities are -0.39, -0.143 and 0.073.
Speech completed..
Type here: Type here: Type here: go right

label predicted: go right
LuLa: My current position is -0.049 as X and 1.371 as Y. My angle with respect t
o ground is 0.04, velocities are -0.361, -0.222 and -0.041.
Speech completed..
Type here: Type here: go right
label predicted: go right
LuLa: My current position is -0.059 as X and 1.353 as Y. My angle with respect t
o ground is 0.021, velocities are -0.329, -0.301 and -0.17.
```

Figure 8: This is an example of the lunar lander classic environment along with the terminal for interaction with the system and for log review.

In the image above, the classic example of lunar lander environment is shown, with the terminal at the side with which the user interact and can read previous logs for a deeper insight.



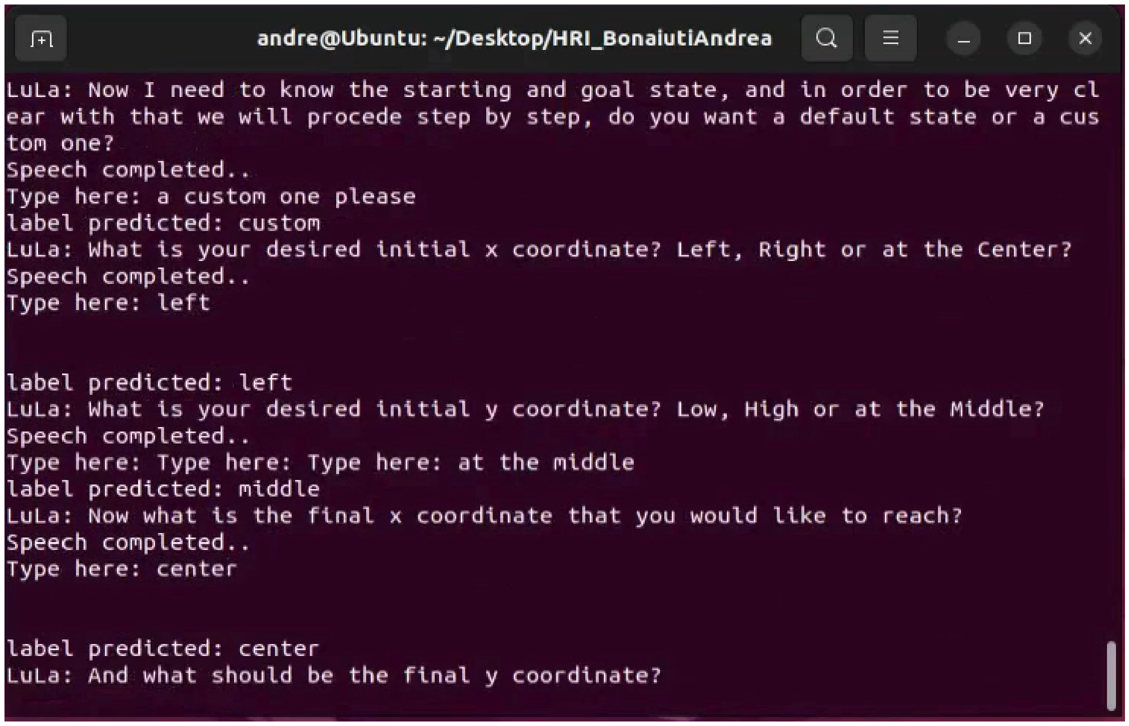
```
ties are -0.584, -0.591 and 0.315.
Speech completed..
Type here: go right
label predicted: go right
LuLa: My current position is -0.233 as X and 0.93 as Y. My angle with respect to ground is 0.44, velociti
es are -0.555, -0.661 and 0.18. But first watch out the vertical velocity, because you could reach critic
ally high values impossible to correct in time.
Speech completed..
Type here: go up
label predicted: go up
LuLa: My current position is -0.251 as X and 0.887 as Y. My angle with respect to ground is 0.469, veloci
ties are -0.624, -0.625 and 0.193. But first watch out the horizontal velocity,, the vertical velocity, b
ecause you could reach critically high values impossible to correct in time.
Speech completed..
Type here: go right

label predicted: go right
LuLa: My current position is -0.269 as X and 0.841 as Y. My angle with respect to ground is 0.485, veloci
ties are -0.596, -0.695 and 0.058. But first watch out the vertical velocity, because you could reach cri
tically high values impossible to correct in time.
Speech completed..
Type here: Type here: go right

label predicted: go right
LuLa: My current position is -0.286 as X and 0.791 as Y. My angle with respect to ground is 0.482, veloci
ties are -0.573, -0.766 and -0.058. But first watch out the vertical velocity, because you could reach cr
itically high values impossible to correct in time.
Speech completed..
Type here: Type here: ok, please stop
```

Figure 9:

The system is able to detect critical values in the coordinates and velocities that, if not adjusted in time, could lead to an inevitable failure (a crash on the ground or going outside the current viewport) .



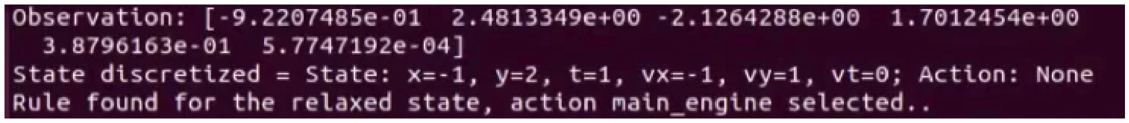
```
andre@Ubuntu: ~/Desktop/HRI_BonaiutiAndrea
LuLa: Now I need to know the starting and goal state, and in order to be very cl
ear with that we will procede step by step, do you want a default state or a cus
tom one?
Speech completed..
Type here: a custom one please
label predicted: custom
LuLa: What is your desired initial x coordinate? Left, Right or at the Center?
Speech completed..
Type here: left


label predicted: left
LuLa: What is your desired initial y coordinate? Low, High or at the Middle?
Speech completed..
Type here: Type here: Type here: at the middle
label predicted: middle
LuLa: Now what is the final x coordinate that you would like to reach?
Speech completed..
Type here: center


label predicted: center
LuLa: And what should be the final y coordinate?
```

Figure 10:

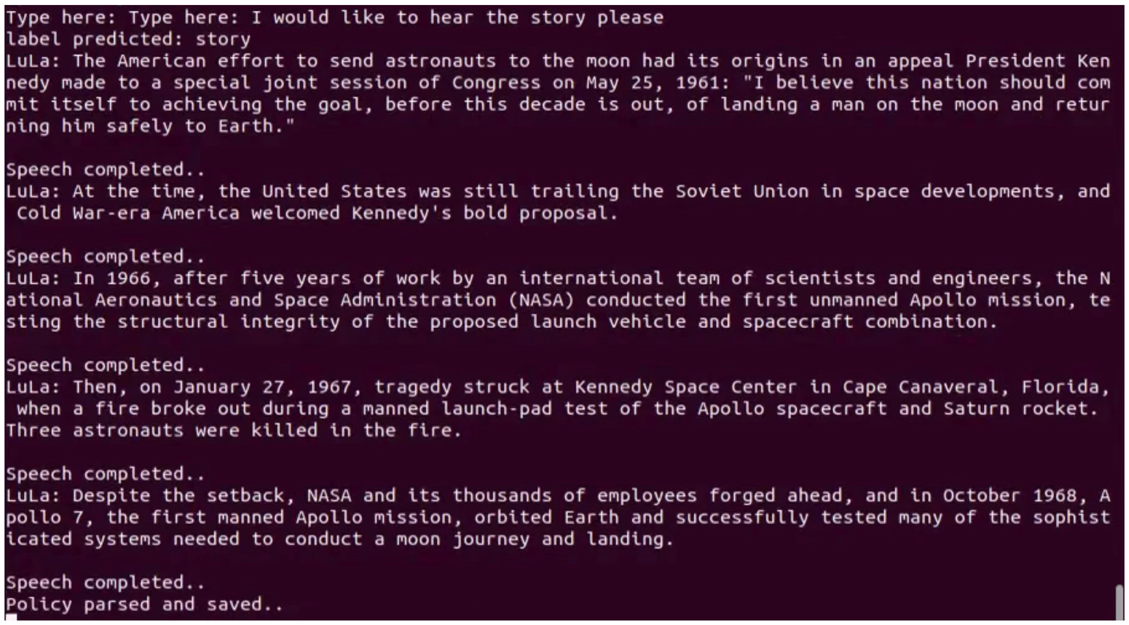
Here, it is demonstrated how the user is able to suggest in natural language the approximate position of the initial state and final one.
For simplicity, three regions of the environment have been selected for each axis, in order to make it easy to reason about. Left, right and at the center for the x axis, and low, high or a middle level for the y coordinate.



```
Observation: [-9.2207485e-01  2.4813349e+00 -2.1264288e+00  1.7012454e+00
  3.8796163e-01  5.7747192e-04]
State discretized = State: x=-1, y=2, t=1, vx=-1, vy=1, vt=0; Action: None
Rule found for the relaxed state, action main_engine selected..
```

Figure 11:

This log is outputted during the final simulation when a policy is parsed and stored. It reports the actual discretization, how it is discretized in the actual used domain, and the rule found in th policy along with the relative action to do.
It is possible, and actually frequent, to obtain a discretized observation that is not included in the policy (i.e. there is not a rule, and therefore an action, for that state). In these cases the log tells it instead of "Rule found.." and the Lander does a random action between *idle* and *main engine*.
Unfortunately, this situation comes frequently, because of the random initialization of the environment, that has been untouched in order to prove the limitations of using planning techniques alone, and because of the limited state spaced explored by that.



```
Type here: Type here: I would like to hear the story please
label predicted: story
LuLa: The American effort to send astronauts to the moon had its origins in an appeal President Ken
nedy made to a special joint session of Congress on May 25, 1961: "I believe this nation should com
mit itself to achieving the goal, before this decade is out, of landing a man on the moon and retur
ning him safely to Earth."

Speech completed..
LuLa: At the time, the United States was still trailing the Soviet Union in space developments, and
 Cold War-era America welcomed Kennedy's bold proposal.

Speech completed..
LuLa: In 1966, after five years of work by an international team of scientists and engineers, the N
ational Aeronautics and Space Administration (NASA) conducted the first unmanned Apollo mission, te
sting the structural integrity of the proposed launch vehicle and spacecraft combination.

Speech completed..
LuLa: Then, on January 27, 1967, tragedy struck at Kennedy Space Center in Cape Canaveral, Florida,
 when a fire broke out during a manned launch-pad test of the Apollo spacecraft and Saturn rocket.
Three astronauts were killed in the fire.

Speech completed..
LuLa: Despite the setback, NASA and its thousands of employees forged ahead, and in October 1968, A
pollo 7, the first manned Apollo mission, orbited Earth and successfully tested many of the sophist
icated systems needed to conduct a moon journey and landing.

Speech completed..
Policy parsed and saved..
```

Figure 12:

This is a screenshot of the story told during the waiting for the subprocess of a custom problem's planning.
The subprocess output have been muted and removed in order to keep the listening and the reading possible and relaxing. The same behaviour is done if the user chooses to use the simulator modality when waiting for the termination of the planning in background.

6 Conclusions

This project has showed how to use planning techniques on real time environments and how they perform on a continuous environment through discretization and non deterministic setting.
As showed, it is not enough to reach optimal performances, with the most reason in the fact that planning in such a big and complex environment result in a long time required for planning the most precise domains and the difficulty to explore the whole state space needed to deal with unexpected situations that non deterministic environments create, with also the property of casual initial states that cause different trajectories that can lead to unexplored states from the planning phase, even if the planning is done by iterating different starting states as how it has been done in this project for the default problem of safe landing. But this could be seen as a preliminary work for the introduction of a phase of Reinforcement Learning after a first planning on the environment.

An attempt to make the interaction more natural with non handcrafted dialogue system has been attempted with the introduction of a NL model that aims to generate paraphrases of the sentences used in the actual project, but with poor results. The sentences generated was very dissimilar and resulted in unnatural way of communication that would have make the interaction a bit strange. The addition of a more recent and performing dialogue system as ChatGPT and familiars has been considered, but it would have made the system heaviear and maybe a bit out of the scope of the project.
The developing has been straightforward for the most part, with some delays caused by difficulties in choosing the bins of the discretized domains. They have been decided by numerous experiments to reach acceptable values for each domain.
Also the interaction part was pretty straightforward, the only problem was the omnipresent one that is the choosing of the right developing environment to use, from the right python version in which all the libraries worked together, the right way to use the subprocess for planning in the background when the system is running. The most part of the developing phase has been done by moving on a virtual machine with Ubuntu OS, due to the easier way to deal with subprocesses and libraries as the text-to-speech tools.

This project could be an example of to build and interact with a remote interface to deal with manipulators or even to autonomous robots, and what should be the minimum interaction from a robotic system to communicate with a user, as for communicating what to do, but also to inform the user/operator what will be done and how.

References

[1] Evan Ackerman. “A Robot for the Worst Job in the Warehouse: Boston Dynamics’ Stretch can move 800 heavy boxes per hour”. In: *Ieee Spectrum* 59.1 (2022), pp. 50–51.

[2] Constructions Aeronautiques et al. “Pddl— the planning domain definition language”. In: *Technical Report, Tech. Rep.* (1998).

[3] Fernando Alonso-Martín et al. “Augmented robotics dialog system for enhancing human–robot interaction”. In: *Sensors* 15.7 (2015), pp. 15799–15829.

[4] Barrett Ames, Allison Thackston, and George Konidaris. “Learning symbolic representations for planning with parameterized skills”. In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2018, pp. 526–533.

[5] Tony Belpaeme et al. “Social robots for education: A review”. In: *Science robotics* 3.21 (2018), eaat5954.

[6] Piergiorgio Bertoli et al. “Extending PDDL to nondeterminism, limited sensing and iterative conditional plans”. In: *Extending PDDL to nondeterminism, limited sensing and iterative conditional plans 15* (), p. 15.

[7] Daniel G Bobrow et al. “GUS, a frame-driven dialog system”. In: *Artificial intelligence* 8.2 (1977), pp. 155–173.

[8] Greg Brockman et al. “Openai gym”. In: *arXiv preprint arXiv:1606.01540* (2016).

[9] Joost Broekens, Marcel Heerink, Henk Rosendal, et al. “Assistive social robots in elderly care: a review”. In: *Gerontechnology* 8.2 (2009), pp. 94–103.

[10] Thurston L Brooks and Thomas B Sheridan. “Superman: A system for supervisory manipulation and the study of human/computer interactions”. In: (1979).

[11] Matteo Cardellini et al. “Taming Discretised PDDL+ through Multiple Discretisations”. In: *34th International Conference on Automated Planning and Scheduling*. AAAI press. 2024.

[12] Matthew Chignoli et al. “The MIT humanoid robot: Design, motion planning, and control for acrobatic behaviors”. In: *2020 IEEE-RAS 20th International Conference on Humanoid Robots (Humanoids)*. IEEE. 2021, pp. 1–8.

[13] Alessandro Cimatti et al. “Weak, strong, and strong cyclic planning via symbolic model checking”. In: *Artificial Intelligence* 147.1-2 (2003), pp. 35–84.

[14] William R Corliss and Edwin G Johnsen. *TELEOPERATOR CONTROLS. An AEC-NASA Technology Survey.* Tech. rep. National Aeronautics and Space Administration, Washington, DC, 1968.

[15] Giuseppe Della Penna et al. “Cost-optimal strong planning in non-deterministic domains”. In: *International Conference on Informatics in Control, Automation and Robotics*. Vol. 2. SciTePress. 2011, pp. 56–66.

[16] Stefan Edelkamp and Jörg Hoffmann. *PDDL2. 2: The language for the classical part of the 4th international planning competition.* Tech. rep. Technical Report 195, University of Freiburg, 2004.

[17] VINICIUS VELOSO ELEUTERIO and Luiz Edival de Souza. “UAV mission planning and execution via non-deterministic AI planning on ROS”. In: *Congresso Brasileiro de Automática-CBA*. Vol. 1. 1. 2019.

[18] Heinrich A Ernst. “MH-1, a computer-operated mechanical hand”. In: *Proceedings of the May 1-3, 1962, spring joint computer conference*. 1962, pp. 39–51.

[19] William R Ferrell and Thomas B Sheridan. “Supervisory control of remote manipulation”. In: *IEEE spectrum* 4.10 (1967), pp. 81–88.

[20] Mary Ellen Foster et al. “Two people walk into a bar: Dynamic multi-party social interaction with a robot agent”. In: *Proceedings of the 14th ACM international conference on Multimodal interaction*. 2012, pp. 3–10.

[21] Maria Fox and Derek Long. “PDDL+: Modeling continuous time dependent effects”. In: *Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space*. Vol. 4. 2002, p. 34.

[22] Maria Fox and Derek Long. “PDDL2. 1: An extension to PDDL for expressing temporal planning domains”. In: *Journal of artificial intelligence research* 20 (2003), pp. 61–124.

[23] Jean Garcia. “Boston Dynamics’ Atlas robot now performs parkour”. In: *UWIRE Text* (2018), pp. 1–1.

[24] A Gerevini and D Long. *Plan constraints and preferences in PDDL3: The language of the fifth international planning competition. University of Brescia*. Tech. rep. Italy, Tech. Rep, 2005.

[25] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: theory and practice*. Elsevier, 2004.

[26] RC Goertz, JR Burnett, and F Bevilacqua. *Servos for remote manipulation*. Tech. rep. Argonne National Lab. Argonne, IL (US), 1953.

[27] George Konidaris, Leslie P Kaelbling, and Tomas Lozano-Perez. “Symbol acquisition for probabilistic high-level planning”. In: AAAI Press/International Joint Conferences on Artificial Intelligence. 2015.

[28] Changyoon Lee, You-Sung Cha, and Tae-Yong Kuc. “Implementation of dialogue system for intelligent service robots”. In: *2008 International Conference on Control, Automation and Systems*. IEEE. 2008, pp. 2038–2042.

[29] Mike Lewis et al. “Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension”. In: *arXiv preprint arXiv:1910.13461* (2019).

[30] Shuyin Li, Britta Wrede, and Gerhard Sagerer. “A computational model of multi-modal grounding for human robot interaction”. In: *Proceedings of the 7th SIGdial Workshop on Discourse and Dialogue*. 2006, pp. 153–160.

[31] Shuyin Li, Britta Wrede, and Gerhard Sagerer. “A dialog system for comparative user studies on robot verbal behavior”. In: *ROMAN 2006-The 15th IEEE International Symposium on Robot and Human Interactive Communication*. IEEE. 2006, pp. 129–134.

[32] Mark Maimone et al. “Surface navigation and mobility intelligence on the Mars Exploration Rovers”. In: *Intelligence for space robotics* (2006), pp. 45–69.

[33] Nicolas Mangold et al. “Perseverance rover reveals an ancient delta-lake system and flood deposits at Jezero crater, Mars”. In: *Science* 374.6568 (2021), pp. 711–717.

[34] Thomas McCluskey and Mauro Vallati. “Embedding automated planning within urban traffic management operations”. In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 27. 2017, pp. 391–399.

[35] Argaman Mordoch et al. “Safe learning of pddl domains with conditional effects”. In: ICAPS. 2023.

[36] Christian Muise, Vaishak Belle, and Sheila A. McIlraith. “Computing Contingent Plans via Fully Observable Non-Deterministic Planning”. In: *The 28th AAAI Conference on Artificial Intelligence*. 2014. URL: http://www.haz.ca/papers/muise-aaai-14.pdf.

[37] Christian Muise, Sheila A McIlraith, and J Christopher Beck. “Improved Non-deterministic Planning by Exploiting State Relevance”. In: *The 22nd International Conference on Automated Planning and Scheduling (ICAPS)*. 2012.

[38] Christian Muise, Sheila A. McIlraith, and Vaishak Belle. “Non-Deterministic Planning With Conditional Effects”. In: *The 24th International Conference on Automated Planning and Scheduling*. 2014. URL: http://www.haz.ca/papers/muise-icaps-14.pdf.

[39] RR Murphy et al. *The safe humanrobot ratio (pp. 31-52)*. 2010.

[40] Ronald PA Petrick, Mary Ellen Foster, and Amy Isard. “Social state recognition and knowledge-level planning for human-robot interaction in a bartender domain”. In: *Workshops at the Twenty-Sixth AAAI Conference on Artificial Intelligence*. 2012.

[41] Stephanie Rosenthal and Manuela Veloso. “Mixed-initiative long-term interactions with an all-day-companion robot”. In: *2010 AAAI Fall Symposium Series*. 2010.

[42] Carl F Ruoff. *Teleoperation and robotics in space*. Vol. 161. Aiaa, 1994.

[43] Paul E Rybski et al. "Interactive robot task training through dialog and demonstration". In: *Proceedings of the ACM/IEEE international conference on Human-robot interaction*. 2007, pp. 49–56.

[44] Thomas B Sheridan. "Human–robot interaction: status and challenges". In: *Human factors* 58.4 (2016), pp. 525–532.

[45] Thomas B Sheridan, William L Verplank, and TL Brooks. "Human/computer control of undersea teleoperators". In: *NASA. Ames Res. Center The 14th Ann. Conf. on Manual Control*. 1978.

[46] John Slaney and Sylvie Thiébaux. "Blocks world revisited". In: *Artificial Intelligence* 125.1-2 (2001), pp. 119–153.

[47] Antony Thomas et al. "Towards perception-aware task-motion planning". In: *Proceedings of the AAAI 2018 Fall Symposium on Reasoning and Learning in Real-World Systems for Long-Term Autonomy. Arlington, VA, USA*. 2018.

[48] David R Traum and Staffan Larsson. "The information state approach to dialogue management". In: *Current and new directions in discourse and dialogue* (2003), pp. 325–353.

[49] Jean Vertut and Philippe Coiffet. *Teleoperations and robotics: evolution and development*. Prentice-Hall, Inc., 1986.

[50] Wenpeng Yin, Jamaal Hay, and Dan Roth. "Benchmarking zero-shot text classification: Datasets, evaluation and entailment approach". In: *arXiv preprint arXiv:1909.00161* (2019).

[51] Håkan LS Younes and Michael L Littman. "PPDDL1. 0: The language for the probabilistic part of IPC-4". In: *Proc. International Planning Competition*. 2004.

[52] Jie Zeng et al. "Eliciting user food preferences in terms of taste and texture in spoken dialogue systems". In: *Proceedings of the 3rd International Workshop on Multisensory Approaches to Human-Food Interaction*. 2018, pp. 1–5.