Universidad de Castilla La Mancha

Autonomous Robotics

# Project: Human-Robot Interaction (HRI)

*Authors:*
Jorge Daniel Laborda Sicilia
Alejandro Martín Simón Sánchez

May 29, 2018

# Contents

# List of Figures

# 1 Introduction

This project consists of the study and development of a speech-based robot control. The robot we use for this project is a Pioneer 3DX and we use ROS as the development framework.

The proposed project is divided in four smaller tasks:

1. Developing a ROS Node to interact with the Google Speech ASR API.
2. Integrate LU4R library to interpret speech commands like "Go to <sogoal>".
3. Implementation of "Go to <goal>" behaviour using rosarnl.
4. Semantic mapping (Optional).

[17]

We've considered this communication scheme (Figure 1) taking the subtasks into account:

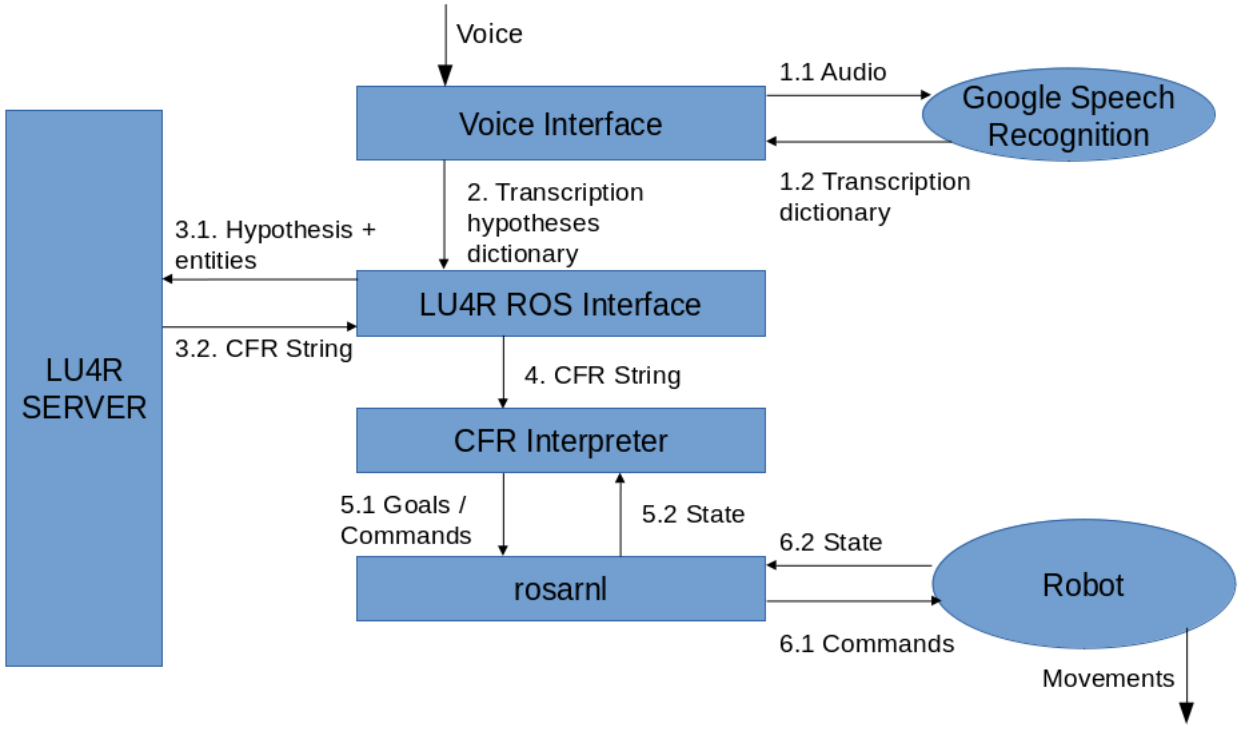


Figure 1: Communication Scheme

The steps the system will follow, as seen in Figure 1 are the following:

First the voice interface will listen to any audio processed by the microphone of the computer. Afterwards, this audio is posted to the Google Speech Recognition API, and it will give as a response a transcription of what the computer has heard. The transcription is a JSON file format, that contains a set of hypothesis transcripts with a confidence score.

Now this JSON file is passed to the LU4R ROS Interface so that it can post it to the LU4R server. Once the information has been processed by the LU4R server, this interface receives another JSON file with a command and details of this command such as the goal and the beneficiary.

The CFR interpreter receives the action and its relative data from the LU4R ROS Interface and interprets what the robot has to do. The CFR interpreter processes the goal information and adds the command to the command queue of rosarnl. When a command is finished, the robot picks up the next command until there are no more. Then it remains idle, waiting for a new command.

# 2 Theoretical Foundations

As we can see in Figure 1, the human-robot interaction project uses mainly four systems:

1. Google Speech Recognition
2. ROS
3. LU4R
4. ROSARNL

We also have to take into account that the robot we are using is a Pioneer 3-DX robot, that is equipped with a set of laser rangefinders and a gripper system that can pick up small objects.

## 2.1 Google Speech Recognition

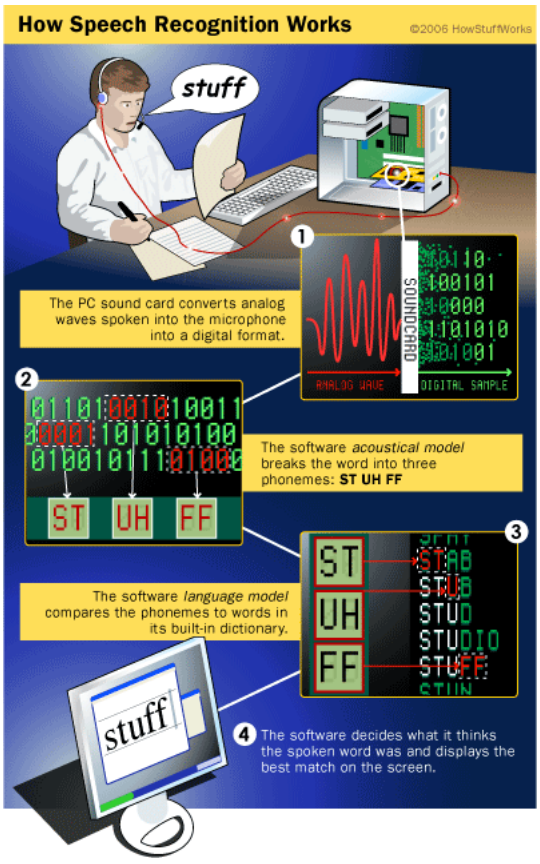


Figure 2: Speech Recognition Process

The basic functionality of a speech recognition system is described in Figure 2. Speech recognition systems have an input and an output. The input is a sentence said by a person, while the output is a transcript of the words that they have said.

When a person says a sentence, the computer uses its own sound card to pass the analog information (sound waves) to digital information (binary code) [9]. This information is sent to different computers around the world belonging to the Google Speech Recognition, and they will process it even more by passing the time domain into the frequency domain and splitting the voice spectrogram into parts

A speech recognition system needs two things to work. Algorithms to recognize phonemes and words, and a dictionary of words (database).

In terms of algorithms, Google [4] uses neuron networks that can adjust their behaviour to improve, learn and evolve. This means that it can give a different answer depending on the

person that is using the recognition system.

Once the processed audio enters the neuron network, the neural network will try to identify the individual components of the voice (vowels and consonants), later, it will identify phonemes, words, sentence structure... These recognized words will be contrasted with the dictionary of words and at the end, the most probable sentences will be sent back to the user.

## 2.2 ROS

ROS [8] stands for Robot Operating System and gives us a flexible framework for writing robot software. It has a collection of libraries, tools and conventions that help developers make their ideas come true.

ROS uses nodes to separate different functionalities of the system. For instance, you can have a node that controls the mobility of a robot, while having another node that controls the inputs of a camera mounted on the same robot. ROS uses a Publisher/Subscription methodology to make the nodes communicate between themselves. This is done by creating topics where some nodes will publish information, and other nodes will subscribe to the same topic.

ROS provides an easy to use network to communicate different parts of the same robotic system.

## 2.3 LU4R

LU4R [15] stands for *adaptative spoken Language Understanding chain For Robots*. It is the product of the collaboration between the SAG group at the University of Roma, Tor Vergata, and the Laboratory of Cognitive Cooperating Robots at Sapienza, University of Rome.

Communication between humans and robots in natural language is a complex subject that requires a deep understanding of cognitive abilities. For instance, a command such as *take the book on the table* would require that both the robot and the person know that both entities exist within the context. The robot must also know where these entities are located by using an inner representation of the objects and the environment surrounding it. Next, a mapping process from lexical references to real world entities must be made. To sum it all up, the robot needs to have a Spoken Language Understanding (SLU) in order to create any interactive dialogue system.

As we know from personal experience, when we use natural language in an interactive manner, both transmitter and receiver are aware of the context. This means that both of them make

constant references to the environment, and entities surrounding them. This problem domain must be made computational and available for the robot in order to create any dialogue with it. Robot interaction needs to be *grounded*, since the meaning of any command depends on the current state of the physical world, and the interpretation of said command, interacts with the perception the robot has about its surroundings.

Taking into account the above information, LU4R uses the proposed SLU by Bastianelli et al, 2016 [14] which integrates both perceptual and linguistic information. The process proposed allows commands to be produced that coherently expresses the constraints about the surroundings (including entities), the Robotic Platform and the pure linguistic level. In order to do this, a discriminative approach based on the Markovian formulation of Support Vector Machines (SVM-HMM) is used, in which grounded information is directly injected within the learning algorithm. Overall, this process improves the robustness of the overall interpretation process.

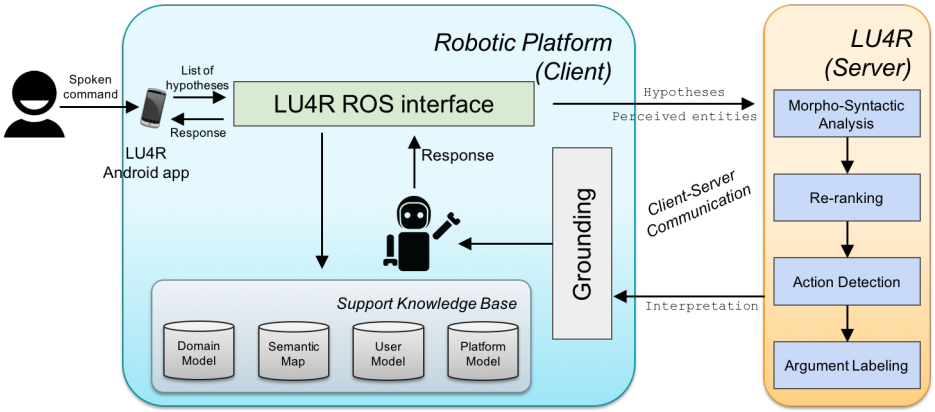


Figure 3: LU4R process

When we use LU4R, it uses the defined SLU process mentioned above, and it will produce an interpretation of a user's spoken words in terms of Frame Semantics [3] (dictionary of more than 13000 words with meanings). Using this methodology, a sentence can be divided into frames elements. For example, the sentence *take the book on the table* can be separated into the following:

$$[\text{take}]_{Taking}[\text{the book on the table}]_{THEME}$$

By using semantic frames, the robot is able to create a bridge between the actions expressed in natural language and the implementations of these actions in the robot's world.

The learning methods of the LU4R have been trained using the Human Robot Interaction Corpus (HuRIC) [5], which contains about 900 audio files for more than 600 sentences annotated

with respect these 18 frames:

*ARRIVING, ATTACHING, BEING_IN_CATEGORY, BEING_LOCATED, BRINGING, CHANGE_DIRECTION, CHANGE_OPERATIONAL_STATE, CLOSURE, COTHEME, GIVING, INSPECTING, LOCATING, MANIPULATION, MOTION, PERCEPTION_ACTIVE, PLACING, RELEASING, TAKING.*

We can see a simplified explanation of how lu4r works in Figure 3

## 2.4 ROSARNL

ROSARNL is responsible for the navigation, localization and movement of our Pioneer 3-DX robot. The navigation done by rosarnl uses an algorithm based on the Monte Carlo algorithm [1].

The Monte Carlo localization algorithm[6] [11] [12] uses a particle filter of the environment to get the best estimation of the location of a robot. A particle filter is the application of Bayes filters [13] to estimate a probability distribution given by a set of samples of the environment. Bayesian filters gives the robot a probabilistic approach of it's own location.

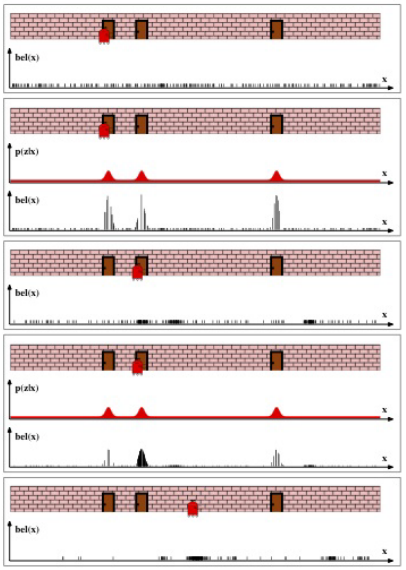


Figure 4: Monte Carlo localization using particle filters

Monte Carlo localization has two steps:

- Prediction or motion update

- Measurement or sensor update

In the prediction step the robot predicts its new location based on the command the robot has been given. This is done by applying the simulated motion to each particle it detects.

In the measurement step, the robot measures the particles again using its lasers. It then updates its particles to get a more accurate representation of the scenario. For each particle the robot needs to compute the probability that, had it been at the state of the particle was believed to be, it would have perceived the same information it is perceiving now.

This makes the localization go smoother as time goes on, since it will be able to have more information and therefore reduce its error.

ROSARNL uses this algorithm to locate the robot, it moves the robot by listening to a subscribed topic where the robot needs to go. Once it recognizes the location it needs to go within the map, the monte carlo algorithm is executed to make sure that the robot is going the right way.

## 2.5 Pioneer 3-DX robot

The Pioneer 3-DX [7] robot is a small lightweight robot with two wheels and two motor differential drives. It has a front SONAR, one battery, wheel encoders, a microcontroller with ARCOS firmware and the Pioneer SDK advanced mobile robotics software development package.

Figure 5: Pioneer 3-DX model

This robot is one of the world's most popular and used mobile robots used mainly for education and research purposes. We can see a photograph of this robot in Figure 5.

This robot will have integrated ROSARNL and ROSARIA to make the developers work with more ease. To control the robot, messages will have to be published into the rosarnl topics.

# 3 Prerequisites

## 3.1 Speech Recognition

At first, we have to install the speech_recognition python package using:

```
pip install SpeechRecognition #pip3 in case of using python3
```

One of the problems we've found is how to capture the audio and use it. In order to solve that problem we need **PyAudio**.

This has to be downloaded from `https://pypi.python.org/pypi/PyAudio#downloads` and be installed with the following commands, so all the dependencies are installed too:

```
sudo apt-get install python-pyaudio python3-pyaudio
sudo apt-get install portaudio19-dev python-all-dev python3-all-dev
&& sudo pip install pyaudio && sudo pip3 install pyaudio
cd PyAudio-0.2.11 #where it is downloaded
sudo python setup.py install #python3 depending on the version
```

[16]

We've uploaded it to this repository: `https://github.com/Crantor/lu4r_ros_interface.git`

## 3.2 LU4R - Language Understanding 4 Robots

### 3.2.1 ROS Interface

We need a ROS Interface for LU4R, we have to clone this repository in the src folder of the workspace with this commands:

```
cd $HOME/catkin_ws/src #where the workspace is located
git clone https://github.com/andreavanzo/lu4r_ros_interface.git
```

We've made a fork of that repository, so we have to clone this one:

```
cd $HOME/catkin_ws/src #where the workspace is located
git clone https://github.com/Crantor/lu4r_ros_interface.git
```

We also need to install this python package:

```
sudo pip install requests
```

### 3.2.2 LU4R Server

We also need the LU4R package itself, it is downloaded from `http://sag.art.uniroma2.it/download/lu4r/`

A password is needed to download it, it can be requested to the creators, their emails are vanzo@dis.uniroma1.it and croce@info.uniroma2.it.[15]

We need Java 1.8 to run it, these commands can be used to install it:

```
sudo add-apt-repository ppa:webupd8team/java
sudo apt-get update
sudo apt-get install oracle-java8-installer
sudo update-alternatives --config java #choose a java installation to be used
```

In order to define the environment variable "JAVA_HOME" we follow these steps. At first, we can see where java is installed by using this command:

```
sudo update-alternatives --config java
```

Secondly, we have to edit /etc/environments

```
sudo gedit /etc/environment
```

We have to add or modify this line in which we have to put where the installation is located, in our case it is like this:

```
JAVA_HOME="/usr/lib/jvm/java-8-oracle/jre/bin/java"
```

Finally we have to reload the file and check the variable:

```
source /etc/environment
echo $JAVA_HOME
```

As an alternative, we can deploy the server in a docker container [2]. For this, docker needs to be installed in our computers, you can find the download file in it's home page. This way we can guarantee that the necessary changes have already been done inside the container, and there is no need for further configurations of the environment.

To create a docker container, we have to create a folder that will contain the folder containing the LU4R server as well as a Dockerfile [10]. The folder containing the server needs to be called

lu4r-0.2.1_. The Dockerfile will have the following content:

```
# Pulling Base image
FROM ubuntu:16.04

# Installing requisites
RUN apt-get update \
  && apt-get -y install software-properties-common \
  && add-apt-repository ppa:webupd8team/java \
  && apt-get update \
  && echo oracle-java8-installer shared/accepted-oracle-license-v1-1 select true | /usr/
  && apt-get -y install oracle-java8-installer

# Adding JAVA HOME Variable
ENV JAVA_HOME="/usr/lib/jvm/java-8-oracle/jre/bin/java"

# COPYING
COPY lu4r-0.2.1_ /root

# Workdirectory
WORKDIR /root

# Default command
CMD ["java","-jar","-Xmx1G","lu4r.jar","basic","cfr", "en", "9090"]
```

Now all we have to do is build the image and run it. This is done by using the following commands in a terminal:

```
1. cd [PATH_TO_LU4R_DOCKER_FOLDER]
2. docker build --tag=lu4r-server .
3. docker run -it --rm -p 9090:9090 lu4r-server
```

Once the server is running, we can check that everything is working by going to the url *https://localhost:9090/service/nlu*. Here we will get a 405 error saying that the method is not approved.

## 3.3 ROSARNL

For this assignment we make use of **rosarnl**, here we add a list of relevant topics/services for this task. We are using the following packages:

- ARIA. MobileRobots Advanced Robot Interface for Applications (ARIA) is a C++ library (software development toolkit or SDK) for all MobileRobots/ActivMedia platforms.

  It uses ARIA 2.9.1.

- ARNL. It is a set of software packages built on top of ARIA for intelligent navigation and localization. [17]

  It uses Base ARNL Library 1.9.2 and ARNL Laser Localization Library 1.9.2. [17]

- MobileSim 0.7.3. It is a software for simulating MobileRobots/ActivMedia platforms and their environments, for debugging and experimentation with ARIA and ARNL. [17]

- ros-arnl. The ros-arnl package contains a ROS node called rosarnl node which provides a ROS interface to basic ARNL features.

  This can be got from `https://github.com/MobileRobots/ros-arnl` We have renamed the ros package to "rosarnl" (notice that we removed the '-' character).

  In our case, we have replaced it for a rosarnl node which was modified by Cristina Romero González, this version is able to use the grips of the robot. This version will be referenced as "rosarnl_node_with_gripper".

### 3.3.1 ROSARNL Topics

The following topics [18] [19] are used:

- **Go to goal:** To send the robot to a goal using the goal name, the rosarnl node offers the /rosarnl_node/goalname topic, which receives a string (std_msgs/String) with the name of the target goal.

  **rostopic pub -1 /rosarnl_node/goalname std_msgs/String "GoalName"**

- **Goal reached:** To know that the robot has reached the target position, we will need to subscribe to one of the following topics:

  - The topic **/rosarnl_node/arnl_server_status** publishes a string message **(std_msgs/String)** with different information of the current status of the robot. One of the messages

published in this topic is "Arrived at X", where X is the name of the goal.

- The topic **/rosarnl_node/arnl_path_state** publishes a string message **(std_msgs/String)** with different information of the path planning status. One of the messages published in this topic is **"REACHED_GOAL"**.

- **Grip open**: An empty message can be published in the /rosarnl_node/gripper/open topic to open the claws. This topic is used in "rosarnl_node_with_gripper" version.

  **rostopic pub -1 /rosarnl_node/gripper/open std_msgs/Empty**

- **Grip close**: An empty message can be published in the /rosarnl_node/gripper/close topic to close the claws. This topic is used in "rosarnl_node_with_gripper" version.

  **rostopic pub -1 /rosarnl_node/gripper/close std_msgs/Empty**

- **Grip stop**: An empty message can be published in the /rosarnl_node/gripper/stop topic to stop the claws movement. This topic is used in "rosarnl_node_with_gripper" version.

  **rostopic pub -1 /rosarnl_node/gripper/stop std_msgs/Empty**

# 4 Development

## 4.1 Speech Recognition

We need:

- Python 2.6, 2.7, or 3.3+.
- PyAudio 0.2.11+ (for microphone input).

  The instructions to install it are available in section 3.1.

- Google Speech Recognition.

  We have chosen this speech recognition API by one main reason, there's no need of any API key and it's a powerful tool. We just have to use the API directly in the ROS node.

  The implementation is the following:

  As we can see in the Figure 6, the main method just keeps the node running until the user writes "quit"

```
if __name__ == '__main__':
    init()
    while True:
        print "Press Enter to start listening, type \"quit\" to exit..."
        line = raw_input()
        if "quit" in line:
            break
        print "..say something!"
        listen()
```

Figure 6: Voice ROS Interface - Main method

The init method (Figure 7) initializes the ros node, the voice recognizer and the publisher.

```
def init():
    global r
    global m
    global pub
    rospy.init_node('audiolistener', anonymous=True)
    r = sr.Recognizer()
    m = sr.Microphone()
    with m as source:
        r.adjust_for_ambient_noise(source)
    pub = rospy.Publisher('/audio',String, queue_size = 1000)
```

Figure 7: Voice ROS Interface - Init method

The listen method (Figure 8) gets the audio and sends it to Google Speech Recognition service and receives the transcription as a dictionary.

Then, we have to add the "rank" key as it's necessary but Google doesn't add it, we can add it in order as the transcriptions are already ordered. A "confidence" key also has to be added when it's not present. The dictionary is transformed into json format and the "transcript" is replaced by "transcription", which is the one that LU4R uses. All that is stored using "hypotheses" key. Finally the string is published in the "/audio" topic.

```
def listen():
    with m as source:
        audio = r.listen(source)
    try:
        voice_command = r.recognize_google(audio, None, "en", True)
    except sr.UnknownValueError:
        print("Google Speech Recognition could not understand audio")
    except sr.RequestError as e:
        print("Could not request results from Google Speech Recognition service; {0}".format(e))

    if isinstance(voice_command,dict):
        print voice_command
        j = 1
        for i in voice_command['alternative']:
            i['rank'] = j # the first one will always take rank 1, the most realiable transcription.
            j = j + 1
            if not 'confidence' in i.keys():
                i['confidence'] = 0 # As google just returns one value with confidence it has to be a
        voice_command = json.dumps(voice_command['alternative']) # converted into json format
        voice_command = '{"hypotheses":' + voice_command.replace("transcript","transcription")+'}' #
        print "Voice command string: " + voice_command
        pub.publish(voice_command)
    else:
        print voice_command
```

Figure 8: Voice ROS Interface - Listen method

Finally, we have to give it execution permission:

```
chmod +x voice_interface.py
```

## 4.2 ROS Interface to communicate with LU4R server

The listener method (Figure 9) is called in the main method, it initializes the node, the publisher the subscriber and keeps the node running. It gets the LU4R server ip (127.0.0.1 by default) and its port (9090 by default) from the parameters, so the url of the service provided by the LU4R server is set. It also loads the entities from the semantic map (semantic_map1.txt by default).

```
def listener():
    global semantic_map
    global sub
    global lu4r_ip
    global lu4r_port
    global lu4r_url
    global entities
    global pub

    rospy.init_node('simlab_interface', anonymous=True)
    rospy.Subscriber('/audio', String, inputaudiocallback)
    pub = rospy.Publisher('/interpretation',String, queue_size = 1000)

    lu4r_ip = rospy.get_param("~lu4r_ip", '127.0.0.1')
    lu4r_port = rospy.get_param("~lu4r_port", '9090')
    lu4r_url = 'http://' + lu4r_ip + ':' + str(lu4r_port) + '/service/nlu'

    #loading entities
    sem_map = rospy.get_param('~semantic_map', 'semantic_map1.txt')
    entities = open(directory + "/semantic_maps/" + sem_map).read()
    json_string = json.loads(entities)
    print 'Entities into the semantic map:'
    for entity in json_string['entities']:
        semantic_map[entity['type']] = Pose2D()
        semantic_map[entity['type']].x = entity["coordinate"]["x"]
        semantic_map[entity['type']].y = entity["coordinate"]["y"]
        semantic_map[entity['type']].theta = entity["coordinate"]["angle"]
        print '\t' + entity['type']
        print str(semantic_map[entity['type']])
        print

    rospy.spin()
```

Figure 9: ROS Interface for LU4R Server - Listener method

The inputaudiocallback method is called when data is published into the "/audio" topic, it receives the transcriptions from that topic and stores the hypotheses and the entities into a dictionary which is sent to the LU4R server using the HTTP protocol. It receives the interpretation as a response which is published into the /interpretation topic.

```
def inputaudiocallback(data):
    print 'Received: ' + data.data
    to_send = {'hypo': data.data, 'entities': entities}
    response = requests.post(lu4r_url, to_send, headers=HEADERS)
    print response.text
    if(response.text is not 'NO FRAME(S) FOUND'):
        pub.publish(response.text)
    else:
        print('Lu4r server could not understand the command.')
```

Figure 10: ROS Interface for LU4R Server - Audio Callback

Finally, we have to give it execution permission:

```
chmod +x simlab_interface.py
```

## 4.3 ROS Interpreter

The listener method initializes the node and all the publishers and subscribers. See Figure 11.

```
def listener():
    global sub_interpreter
    global sub_arnl
    global pub
    global pub_gripOpen
    global pub_gripStop
    global pub_gripClose

    # Initialize node, subscribers and publishers
    rospy.init_node('simlab_interpreter', anonymous=True)
    sub_interpreter = rospy.Subscriber('/interpretation', String, interpretercallback)
    sub_arnl = rospy.Subscriber('/rosarnl_node/arnl_path_state', String, stateCallback)
    pub = rospy.Publisher('/rosarnl_node/goalname',String, queue_size = 1000)
    pub_gripOpen = rospy.Publisher('/rosarnl_node/gripper/open',Empty, queue_size = 1000)
    pub_gripStop = rospy.Publisher('/rosarnl_node/gripper/stop',Empty, queue_size = 1000)
    pub_gripClose = rospy.Publisher('/rosarnl_node/gripper/close',Empty, queue_size = 1000)

    # Close the claws at the beginning
    rospy.sleep(2)
    pub_gripClose.publish(Empty())

    rospy.spin()

if __name__ == '__main__':
    listener()
```

Figure 11: ROS CFR Interpreter - Init - Listener method

The checkList method (Figure 12) checks if an element is in a list and returns its index, otherwise it returns -1.

```
def checkList(x,l):
    for elem in l:
        if x.lower() in elem:
            return l.index(elem)
    return -1
```

Figure 12: ROS CFR Interpreter - Checklist method

The statecallback method receives information from rosarnl about the state of the robot. When it reaches a goal receives a string which is "REACHED_GOAL". Firstly, the claws method (Figure 13) is defined, which opens the claws and then closes them back again.

```
def stateCallback(data):
    # Manages the claws of the robot.
    def claws():
        # Open claws
        pub_gripOpen.publish(Empty())
        # Wait a few seconds
        rospy.sleep(2)
        # Stop
        pub_gripStop.publish(Empty())
        # Wait a few seconds
        rospy.sleep(2)
        # Close claws
        pub_gripClose.publish(Empty())
        # Wait a few seconds
        rospy.sleep(2)


    global isWorking
    global current_target
    global last_target
    global command_list
    global objects_to_be_taken
```

Figure 13: ROS CFR Interpreter - State callback 1

When the robot reaches a goal (Figure 14) the last target is updated. In case the current action is the first part of "Bringing" the requests the object and open and closes the claws, in case it's the second part, as it is already taking the object it open the claws to leave the object and closes them.

If there aren't any commands left the current target is set to a tuple of a void string and isWorking is set to False. If there are more actions to be executed the next command is assigned to current target and the goal of the current target is published.

```
if(data.data) == "REACHED_GOAL":

    # Establishes the last visited target
    last_target = current_target

    # BRINGING1
    # In case it is a BRINGING1 action, it has to return and request the object
    if (current_target[0] == possible_actions[18]):
        print("Give me the " + objects_to_be_taken.pop(0))
        # Get object in the claws
        claws()
    # BRINGING2
    elif (current_target[0] == possible_actions[19]):
        print(" Giving object to master")
        # Release the object
        claws()


    # In case there are no commands it stops working
    if len(command_list) == 0:
        isWorking = False
        current_target = ("","")
    # In case it has to do more actions, it does the next one
    else:
        # Establishes the new target and publishes it
        current_target = command_list.pop(0)
        pub.publish(current_target[1])
```

Figure 14: ROS CFR Interpreter - State callback 2

In Figure 15 we can see a list with the possible actions that can be received from the LU4R server, as well as the defined goals and objects which are in each goal. The command list, the objects list that the robot has to take, the working state and the targets are initialized.

```
possible_actions = ['ARRIVING',                 # 0
                    'ATTACHING',                # 1
                    'BEING_IN_CATEGORY',        # 2
                    'BEING_LOCATED',            # 3
                    'BRINGING',                 # 4
                    'CHANGE_DIRECTION',         # 5
                    'CHANGE_OPERATIONAL_STATE', # 6
                    'CLOSURE',                  # 7
                    'COTHEME',                  # 8
                    'GIVING',                   # 9
                    'INSPECTING',               # 10
                    'LOCATING',                 # 11
                    'MANIPULATING',             # 12
                    'MOTION',                   # 13
                    'PERCEPTION_ACTIVE',        # 14
                    'PLACING',                  # 15
                    'RELEASING',                # 16
                    'TAKING',                   # 17
                    'BRINGING1',                # 18 (CUSTOM MADE)
                    'BRINGING2'                 # 19 (CUSTOM MADE)

                    ]

command_list = []
goals = [
    ["Goal0","initial","start","beginning"],
    ["Goal1","kitchen"],
    ["Goal2","bathroom","toilet","bath","sink"],
    ["Goal3","exit","door","escape"]]

objects = [
    ["Goal0","bottle"],
    ["Goal1","knife","oven","spoon","fork","food"],
    ["Goal2","toothbrush","soap"],
    ["Goal3","mobile"]]

# Checks if the robot is working in a task
isWorking = False
# Tuple containing the order and the target the robot needs to go
current_target = ("","")
# Tuple containing the last order executed with the target the robot was.
last_target = ("","")
# Objects the robot needs to take from the environment.
objects_to_be_taken = []
```

Figure 15: ROS CFR Interpreter - Variables

The motion command method checks if one of the words is a goal, then the goal is assigned and the tuple (action, goal) is appended to the command list. See Figure 16.

```
def motion_command(action,content):
    global command_list

    content = content.split(":")[1]
    content = content.replace("\"","")
    # Splitting content into words
    content = content.split(" ")

    # Checking if any of the words is a goal
    for word in content:
        index = checkList(word, goals)
        if index != -1:
            break

    if index != -1:
        goal = goals[index][0]
        # Giving order to go to the indicated goal
        print("MOTION: Going to ",goal)
        command_list.append((action, goal))

    else:
        print("I did not understand you, please repeat.")
```

Figure 16: ROS CFR Interpreter - Motion command method

The motion command method also checks where the object is located and assigns it as a goal. Then if the robot has just started and there's no last target assigned it is set to "Goal0" by default. Then the goal where the object has to be deposited is assigned to the last visited goal. See Figure 17.

```
# Bringing function
def bringing_command(action,content):
    global current_target
    global last_target
    global command_list
    global objects_to_be_taken

    # Deparsing the content
    content = content.split("theme:")[1]
    content = content.replace("\"","")
    # Splitting content into words
    content = content.split(" ")

    # Checking if any of the words is an object
    for word in content:
        index = checkList(word, objects)
        if index != -1:
            objects_to_be_taken.append(word)
            break
    # In case the object is recognized
    if index != -1:
        goal = goals[index][0]
        bringing_object = objects_to_be_taken[len(objects_to_be_taken)-1]
        # Giving order to go to the indicated goal
        print("BRINGING: Bring the " + bringing_object)
        # No order given yet. First time executing robot
        if last_target == ("",""):
            # Adding "Bringing1 to last_target"
            last_target = (possible_actions[18],"Goal0")

        #  There are orders in the command list. Gets the last place where the ro
        if len(command_list) > 0:
            return_object_to = command_list[len(command_list)-1][1]
        # In case the robot has no target go to the last visited target
        elif current_target == ("",""):
            return_object_to = last_target[1]
        # In case the robot is going to a target has to return the object to it
        else:
            return_object_to = current_target[1]
```

Figure 17: ROS CFR Interpreter - Bringing command method 1

Afterwards the object is removed from where it was. In case that the last target is a kraken type the goal where the object has to be returned is set to "Goal0" by default. After that it's calculated where the object has to be deposited, so it's assigned to the list using the right index.

Finally the two commands are appended to the command list, one for each action (take and deposit the object). See Figure 18.

```
    # Changing the location of the object
    # Removing object from object list
    objects[index].remove(bringing_object)

    # If the return is to the kraken goals it returns to the start
    if return_object_to.lower() in ["kraken1","kraken2","kraken3","kraken4"]:
        return_object_to = "Goal0"

    # Getting index of the goal
    g_aux = [goals[i][0] for i,_ in enumerate(goals)]
    indexAdd = g_aux.index(return_object_to)
    # Adding the object to the new location
    objects[indexAdd].append(bringing_object)

    # Adding the target and the return target to the command_list.
    command_list.append((possible_actions[18], goal))              # BRINGING1
    command_list.append((possible_actions[19], return_object_to))   # BRINGING2


# In case the object is not recognized
else:
    print("I did not understand you, please repeat.")
```

Figure 18: ROS CFR Interpreter - Bringing command method 2

For the release command we've implemented a funny movement, if we say "Release the Kraken" it makes a square in the center of the room. See Figure 19.

```
def release_command(action,content):
    global command_list

    content = content.split(":")[1]
    content = content.replace("\"","")
    # Splitting content into words
    content = content.split(" ")

    # variable to check if a word is found
    found = False
    # Checking if any of the words is kraken
    for word in content:
        if word.lower() == "kraken":
            command_list.append((action, "kraken1"))
            command_list.append((action, "kraken2"))
            command_list.append((action, "kraken3"))
            command_list.append((action, "kraken4"))
            found = True
            break

    # in case it is not found
    if not found:
        print("I did not understand you, please repeat.")
    # reinitialize found
    found = False
```

Figure 19: ROS CFR Interpreter - Release command method

When the robot receives the stop command it stops moving by calling to the stop service of rosarnl and reinitializing the variables. See Figure 20.

```
def stop_command():
    global isWorking
    global current_target
    global last_target
    global command_list
    global objects_to_be_taken

    # reinitialize variables
    command_list = []
    isWorking = False
    last_target = current_target # last visited target
    current_target = ("","")
    objects_to_be_taken = []

    # stops the robot
    os.system('rosservice call /rosarnl_node/stop')
```

Figure 20: ROS CFR Interpreter - Stop command method

The interpretercallback method checks what kind of action corresponds to the received string data, so it calls the right method from the ones mentioned above. This can be seen in Figures 21 and 22.

```
def interpretercallback(data):
    global isWorking
    global current_target
    global last_target
    global command_list
    global objects_to_be_taken

    def getContent():
        return data.data.split('(')[1].split(')')[0]

    print 'Received: ' + data.data
    action = data.data.split('(')[0]
    # content = data.data.split('(')[1].split(')')[0]

    # BRINGING
    # Example of transcription received: BRINGING(beneficiary:"me",theme:"the book")
    if action == possible_actions[4]:
        # Getting content
        content = getContent()
        # Executing bringing command
        bringing_command(action, content)

    # CHANGE_OPERATIONAL_STATE (STOP)
    if action == possible_actions[6]:
        stop_command()
```

Figure 21: ROS CFR Interpreter - Interpreter callback method 1

Finally, if the robot is stopped (when it has just started working, or it has received a stop command or it has finished all the actions) the working state is set to true and the current target is set to the first command in the command list, the last target is set to the current target and the current target goal is published. See Figure 22.

```
# MOTION
# Example of transcription received: MOTION(goal:"to the bathroom")
if action == possible_actions[13]:
    # Getting whole content
    content = getContent()
    # Executing Motion Command
    motion_command(action, content)

# RELEASING
# Example of transcription received: RELEASING(theme:"the Kraken")
if action == possible_actions[16]:
    # Getting whole content
    content = getContent()
    release_command(action, content)

# Start moving when it has stopped, or it's moving for the first time.
if not isWorking and len(command_list) is not 0:
    isWorking = True
    # Sending work to the arnl node
    current_target = command_list.pop(0)
    last_target = current_target
    pub.publish(current_target[1]) # Getting First command.
```

Figure 22: ROS CFR Interpreter - Interpreter callback method 2

## 4.4 Launch file

As there are a lot of nodes it is useful to use a launch file. For this project we use four nodes inside a roslaunch file:

- audiolistener: This node transmits the audio to Google Speech Report and gets a transcription. (Figure 23).

```
<!-- Launch voice interpreter -->
<node name="audiolistener" pkg="lu4r_ros_interface" type="voice_interface.py" output ="screen">
</node>
```

Figure 23: Launch File - Node AudioListener

- simlab_interface: Provides a ROS interface between the audiolistener and simlab_interpreter nodes and the LU4R server. (Figures 24 and 25).

```
<launch>

  <arg name="lu4r_ip" default=""/>
  <arg name="lu4r_port" default=""/>
  <arg name="semantic_map" default=""/>
```

Figure 24: Launch File - Node Simlab_Interface Arguments

```
<!-- Launch simlab_interface -->
<node name="simlab_interface" pkg="lu4r_ros_interface" type="simlab_interface.py"
  args="_lu4r_ip:=$(arg lu4r_ip) _lu4r_port:=$(arg lu4r_port) _semantic_map:=$(arg semantic_map)" output="screen">
</node>
```

Figure 25: Launch File - Node Simlab_Interface

- simlab_interpreter: This node interprets the response of LU4R server and interacts with rosarnl. (Figure 26).

```
<!-- Launch simlab_interpreter -->
<node name="simlab_interpreter" pkg="lu4r_ros_interface" type="simlab_interpreter.py" output="screen">
</node>
```

Figure 26: Launch File - Node Simlab_Interpreter

- rosarnl_node: Provides a ROS interface to basic ARNL features. (Figure 27).

```
  <!-- Launch rosarnl -->
  <node name="rosarnl_node" pkg="rosarnl" type="rosarnl_node" output="screen">
  </node>

</launch>
```

Figure 27: Launch File - Node rosarnl

We've made two launch files for this project, one includes the node rosarnl_node, but the other one doesn't, so we use one for the simulation and the other launch file when we want to use a real robot.

# 5 Testing

We also have to add the path of the map to the file arnl.p in the computer in which rosarnl is going to be running (Figure 28):

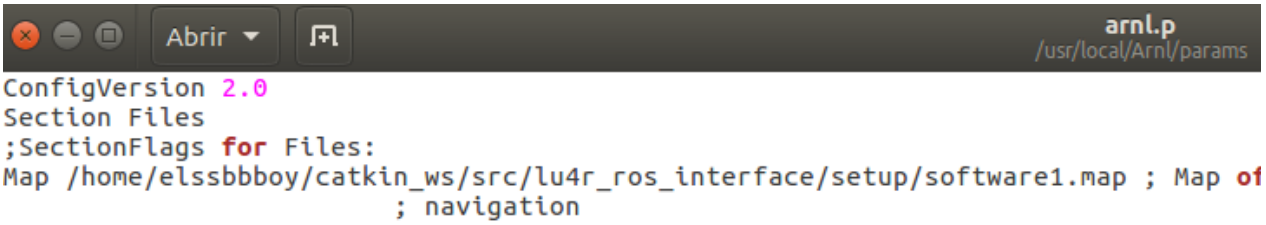


Figure 28: Map path - arnl.p

There are two main ways to execute it:

- We can use the launch file we made using roslaunch command.
- Or we can use rosrun, so we use one terminal for each node.

## 5.1 Using the roslaunch file

To execute everything we need to use these commands:

**Terminal 1 - LU4R Server**

```
java -jar -Xmx1G lu4r.jar [type] [output] [language] [port] #in lu4r server path
```

where:

[type] is the LU4R operating mode, basic or simple. The basic setting does not contemplate perceptual knowledge during the interpretation process. Conversely, the simple configuration relies on perceptual information, enabling a context-sensitive interpretation of the command at the predicate level.

[output] is the preferred output format of the interpretation. The available formats are:

- xml: this is the default output format. The interpretation is given in the XDG format (eXtended Dependency Graph) and XML compliant container.
- amr: the interpretation is given in the Abstract Meaning Representation.
- conll: it is a CoNLL-like tabular representation of the predicates found.
- cfr: the Command Frame Representation it is the RobotCup@Home format for the predicates (frames) produced by the chain.

[language]: the operating language of LU4R. At the moment, only the en (English) version is supported.

[port] the listening port of LU4R.

For example, we use:

```
java -jar -Xmx1G lu4r.jar basic cfr en 9090
```

Remember we use cfr format in the interpreter.

Another option is to use docker, see the Requirements section above to see how to run the LU4R Server with docker.

### 5.1.1 Simulated

**Terminal 2 - MobileSim**

```
MobileSim -m <where the map is located>
```

**Terminal 3 - roslaunch**

```
cd $HOME/catkin_ws #where the workspace is located
catkin_make
source devel/setup.bash
roslaunch lu4r_ros_interface simlab_launch_simulated.launch
lu4r_ip:=[lu4r_ip_address] lu4r_port:=[lu4r_port] semantic_map:=[semantic_map]
```

where:

_lu4r_ip_address: the ip address of LU4R server. If the LU4R and the LU4R ROS interface are on the same machine, ignore this argument, as it is set to 127.0.0.1 by default.

_lu4r_port: the listening port of LU4R server. It is set to 9090 by default.

_semantic_map: the semantic map to be used, among the ones available into semantic_maps. Semantic maps are in JSON format, representing the configuration of the environment (e.g. objects, locations,...) in which the robot is operating. Whenever a simple configuration of LU4R is chosen, the interpretation process is sensitive to different semantic maps. By default "semantic_map1.txt".

Examples:

```
roslaunch lu4r_ros_interface simlab_launch_simulated.launch
```

```
roslaunch lu4r_ros_interface simlab_launch_simulated.launch
```

```
lu4r_ip:=127.0.0.1 lu4r_port:=9090 semantic_map:=semantic_map1.txt
```

### 5.1.2 Connecting directly to the robot

In case of connecting directly to the robot, MobileSim is not executed.

As rosarnl has to be executed in the robot in this case we have prepared another launch file which doesn't execute it, which is called simlab_launch.launch.

We also would have to set up the following variables at the beginning in our computer (once in the terminal where the roslaunch is used):

```
export ROS_IP=XXX.XXX.XXX.XXX # our IP
export ROS_HOSTNAME=XXX.XXX.XXX.XXX # our IP
export ROS_MASTER_URI=http://YYY.YYY.YYY.YYY:11311 # the remote machine IP
```

**Terminal 2 - roslaunch**

```
cd $HOME/catkin_ws #where the workspace is located
catkin_make
source devel/setup.bash
roslaunch lu4r_ros_interface simlab_launch.launch
lu4r_ip:=[lu4r_ip_address] lu4r_port:=[lu4r_port] semantic_map:=[semantic_map]
```

where:

_lu4r_ip_address: the ip address of LU4R server. If the LU4R and the LU4R ROS interface are on the same machine, ignore this argument, as it is set to 127.0.0.1 by default.

_lu4r_port: the listening port of LU4R server. It is set to 9090 by default.

_semantic_map: the semantic map to be used, among the ones available into semantic_maps. Semantic maps are in JSON format, representing the configuration of the environment (e.g. objects, locations,...) in which the robot is operating. Whenever a simple configuration of LU4R is chosen, the interpretation process is sensitive to different semantic maps. By default "semantic_map1.txt".

Examples:

```
roslaunch lu4r_ros_interface simlab_launch.launch
```

```
roslaunch lu4r_ros_interface simlab_launch.launch
```

```
lu4r_ip:=127.0.0.1 lu4r_port:=9090 semantic_map:=semantic_map1.txt
```

Remember we would need to copy the map to the robot computer and add its path to arnl.p file, as shown in Figure 28 at the beginning of Section 5.

**Terminal 1 - Robot**

```
roscore
```

**Terminal 2 - Robot**

```
cd $HOME/catkin_ws #where the workspace is located
catkin_make
source devel/setup.bash
rosrun rosarnl rosarnl_node
```

## 5.2 Using a terminal for each node

To execute everything we need to use these commands:

**Terminal 1 - LU4R Server**

```
java -jar -Xmx1G lu4r.jar [type] [output] [language] [port] #in lu4r server path
```

where:

[type] is the LU4R operating mode, basic or simple. The basic setting does not contemplate perceptual knowledge during the interpretation process. Conversely, the simple configuration relies on perceptual information, enabling a context-sensitive interpretation of the command at the predicate level.

[output] is the preferred output format of the interpretation. The available formats are:

- xml: this is the default output format. The interpretation is given in the XDG format (eXtended Dependency Graph) and XML compliant container.
- amr: the interpretation is given in the Abstract Meaning Representation.
- conll: it is a CoNLL-like tabular representation of the predicates found.
- cfr: the Command Frame Representation it is the RobotCup@Home format for the predicates (frames) produced by the chain.

[language]: the operating language of LU4R. At the moment, only the en (English) version is supported.

[port] the listening port of LU4R.

For example, we use:

```
java -jar -Xmx1G lu4r.jar basic cfr en 9090
```

Remember we use cfr format in the interpreter.

### 5.2.1 Simulated

**Terminal 2 - roscore**

```
roscore
```

**Terminal 3 - Voice Interface**

```
cd $HOME/catkin_ws #where the workspace is located
catkin_make
source devel/setup.bash
rosrun lu4r_ros_interface voice_interface.py
```

**Terminal 4 - LU4R ROS Interface**

```
cd $HOME/catkin_ws #where the workspace is located
catkin_make
source devel/setup.bash
rosrun lu4r_ros_interface simlab_interface.py _lu4r_ip:=[lu4r_ip_address]
_lu4r_port:=[lu4r_port] _semantic_map:=[semantic_map]
```

where:

_lu4r_ip_address: the ip address of LU4R server. If the LU4R and the LU4R ROS interface are on the same machine, ignore this argument, as it is set to 127.0.0.1 by default.

_lu4r_port: the listening port of LU4R server. It is set to 9090 by default.

_semantic_map: the semantic map to be used, among the ones available into semantic_maps. Semantic maps are in JSON format, representing the configuration of the environment (e.g. objects, locations,...) in which the robot is operating. Whenever a simple configuration of

LU4R is chosen, the interpretation process is sensitive to different semantic maps. By default "semantic_map1.txt".

Examples:

```
rosrun lu4r_ros_interface simlab_interface.py
```

```
rosrun lu4r_ros_interface simlab_interface.py _lu4r_ip:=127.0.0.1
_lu4r_port:=9090 _semantic_map:=semantic\_map1.txt
```

**Terminal 5 - ROS Interpreter**

```
cd $HOME/catkin_ws #where the workspace is located
catkin_make
source devel/setup.bash
rosrun lu4r_ros_interface simlab_interpreter.py
```

**Terminal 6 - MobileSim**

```
MobileSim -m <where the map is located>
```

**Terminal 7 - rosarnl**

```
cd $HOME/catkin_ws #where the workspace is located
catkin_make
source devel/setup.bash
rosrun lu4r_ros_interface simlab_interpreter.py
```

### 5.2.2 Connecting directly to the robot

In case of connecting directly to the robot, MobileSim is not executed.

As rosarnl has to be executed in the robot in this case we have prepared another launch file which doesn't execute it, which is called simlab_launch.launch.

We also would have to set up the following variables at the beginning in our computer (once per terminal in which a ros node will be running):

```
export ROS_IP=XXX.XXX.XXX.XXX # our IP
export ROS_HOSTNAME=XXX.XXX.XXX.XXX # our IP
export ROS_MASTER_URI=http://YYY.YYY.YYY.YYY:11311 # the remote machine IP
```

We use ssh commands to launch terminals from the root. The following commands are executed to run the system:

**Terminal 1 - Robot**

```
roscore
```

**Terminal 2 - Robot**

```
cd $HOME/catkin_ws #where the workspace is located
catkin_make
source devel/setup.bash
rosrun rosarnl rosarnl_node
```

Remember we would need to copy the map to the robot computer and add its path to arnl.p file, as shown in Figure 28 at the beginning of Section 5.

The lu4r server must be executed too.

**Terminal 2 - Voice Interface**

```
cd $HOME/catkin_ws #where the workspace is located
catkin_make
source devel/setup.bash
rosrun lu4r_ros_interface voice_interface.py
```

**Terminal 3 - LU4R ROS Interface**

```
cd $HOME/catkin_ws #where the workspace is located
catkin_make
source devel/setup.bash
rosrun lu4r_ros_interface simlab_interface.py _lu4r_ip:=[lu4r_ip_address]
_lu4r_port:=[lu4r_port] _semantic_map:=[semantic_map]
```

where:

_lu4r_ip_address: the ip address of LU4R server. If the LU4R and the LU4R ROS interface are on the same machine, ignore this argument, as it is set to 127.0.0.1 by default.

_lu4r_port: the listening port of LU4R server. It is set to 9090 by default.

_semantic_map: the semantic map to be employed, among the ones available into semantic_maps. Semantic maps are in JSON format, representing the configuration of the environment (e.g. objects, locations,...) in which the robot is operating. Whenever a simple configuration of

LU4R is chosen, the interpretation process is sensitive to different semantic maps. By default "semantic_map1.txt".

Examples:

```
rosrun lu4r_ros_interface simlab_interface.py
```

```
rosrun lu4r_ros_interface simlab_interface.py _lu4r_ip:=127.0.0.1
_lu4r_port:=9090 _semantic_map:=semantic_map1.txt
```

**Terminal 4 - ROS Interpreter**

```
cd $HOME/catkin_ws #where the workspace is located
catkin_make
source devel/setup.bash
rosrun lu4r_ros_interface simlab_interpreter.py
```

## 5.3 Execution

Firstly, we have to remove the robotPose file if it exists.

Once executed the previous commands, we can start giving orders to the robot by pressing Enter in the terminal which is executing the voice interface node or the terminal in which the roslaunch has been used (depending on how it has been executed, see Section 5).

### 5.3.1 Using a simulator

When the command is interpreted we can see how the robot is moving depending on the command. Here we can see some videos in which we run the simulator (Figure 29):

Go command: `https://www.youtube.com/watch?v=3VuTecLK8pE`

Bring command: `https://www.youtube.com/watch?v=REqMjmafAkk`

Release the Kraken command: `https://www.youtube.com/watch?v=KbtCLxBhpTc`

Stop command: `https://www.youtube.com/watch?v=8Qd9AuGeAlk`

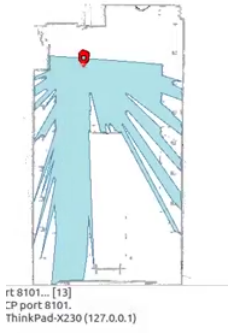


Figure 29: Simulated robot

### 5.3.2 Using the real robot

One of the problems we have faced at the beginning when executing it is that the robot was not moving. That problem appeared because the robot didn't locate itself in the map. So we can use MobileEyes to locate it as a quick fix it this issue takes place.

In the Figure 30 we can see the robot going to a goal.

Figure 30: Real robot

Go command: `https://www.youtube.com/watch?v=pJQY3ACw7CI`

Bring command: `https://www.youtube.com/watch?v=4rOS8xjmGmI`

Release the Kraken command: `https://www.youtube.com/watch?v=q8lmomugFuw`

Stop command: `https://www.youtube.com/watch?v=0e5Zi_e1zEo`

# 6 Conclusions

Natural language is a big computational problem, since it is very difficult to define the whole extent of our language into a computational workspace. Therefore, this domain of knowledge needs to be studied by an AI tool. In this project we have used the LU4R server that will analyze a sentence and give as a result a structured command with objectives and goals. As we can see this is not an easy task, since it has to capture many details.

But the LU4R server is only a small part of this project as we have seen in Figure 1. First of all, the system needs to make a transcript of what the user is saying to the robot. This is done by using the Google Speech Recognition API. This communication with the Google API is done inside a ROS node, once the information provided by the user is processed, this node publishes this processed information into a topic so that the LU4R interface node can read it.

The LU4R interface passes the received transcript to the LU4R server, that will analyze semantically said transcript and will return a command with instructions that can be easily manipulated. Once the LU4R server finishes, the information is published into another topic so that the CFR interpreter node can put these commands into action.

The interpreter knows the structure of each command given by the LU4R server and will add to the command queue whatever the user has ordered the robot to do. The orders are given to the rosarnl node that will control the motion and navigation of the robot for each different command. Once a command has been done, the next command from the queue is issued and the order is analyzed by the interpreter that will give the needed directions to the rosarnl node.

As it is obvious, the simulation done by computer is not the same done by a real robot. The robot commits more errors due to navigation issues, but it still accomplishes each command issued by the user.

One thing this project does not really take into account is the whole context of the environment. For future improvements, a prolog facts database could indeed make the robot understand better it's surrounding and establish a little dialogue with the user to better understand what it's purpose is. For instance, if there were to be more than one bottle in the environment, and we say "*Bring me the bottle!*", then, the robot would not know which one to bring since it is not well specified. With the prolog database, the robot would know that there are two bottles in the environment, and would ask the user which one would they like it to bring.

Nonetheless, this project implements a voice interpreter for a Pioneer-3DX with a basic semantic analysis that will give basic commands to a robot using a complete ROS system.

# References

[1] ARNL Localization - MobileRobots Research and Academic Customer Support. http://robots.mobilerobots.com/wiki/ARNL_Localization.

[2] Docker. https://www.docker.com/.

[3] FrameNet. https://framenet.icsi.berkeley.edu/fndrupal/.

[4] How does Google voice recognition work? | en-articles. http://www.mibqyyo.com/en-articles/2016/04/14/how-does-google-voice-recognition-work/.

[5] HuRIC (Human Robot Interaction Corpus) | Semantic Analytics Group @ Uniroma2. http://sag.art.uniroma2.it/demo-software/huric/.

[6] Monte carlo localization. https://campusvirtual.uclm.es/pluginfile.php/2648124/mod_resource/content/6/unit4_4-Monte_Carlo_Localization.pdf.

[7] Pioneer 3-dx user's guide. http://www.mobilerobots.com/Libraries/Downloads/Pioneer3DX-P3DX-RevA.sflb.ashx.

[8] ROS.org | Powering the world's robots. http://www.ros.org/.

[9] How Speech Recognition Works. https://electronics.howstuffworks.com/gadgets/high-tech-gadgets/speech-recognition.htm, November 2006.

[10] Dockerfile. https://docs.docker.com/engine/reference/builder/, May 2018.

[11] Monte Carlo algorithm. https://en.wikipedia.org/w/index.php?title=Monte_Carlo_algorithm&oldid=841336209, May 2018. Page Version ID: 841336209.

[12] Monte Carlo localization, January 2018. Page Version ID: 822240294.

[13] Recursive Bayesian estimation. https://en.wikipedia.org/w/index.php?title=Recursive_Bayesian_estimation&oldid=820010737, January 2018. Page Version ID: 820010737.

[14] Emanuele Bastianelli, Danilo Croce, Andrea Vanzo, Roberto Basili, and Daniele Nardi. A Discriminative Approach to Grounded Spoken Language Understanding in Interactive Robotic. page 7.

[15] Giuseppe Castellucci Danilo Croce Daniele Nardi Andrea Vanzo Roberto Basili, Emanuele Bastianelli. Lu4r project: adaptive spoken language understanding for robots. http://sag.art.uniroma2.it/lu4r.html.

[16] Anthony Zhang (Uberi). Speech recognition repository. https://github.com/Uberi/speech_recognition.

[17] Ismael García Varea. Campusvirtual: Robótica autónoma. https://campusvirtual.uclm.es/course/view.php?id=27421.

[18] Ismael García Varea. Lab assignment 2: Robot motion. localization, mapping and navigation. https://campusvirtual.uclm.es/mod/assign/view.php?id=1253361.

[19] Ismael García Varea. Lab assignment 5: Slam. https://campusvirtual.uclm.es/mod/assign/view.php?id=1303322.