1 Introduction

In this project, you will be combining several of the areas of AI we’ve discussed in the course (in simulated and simplified ways): vision, language, reasoning, and motion, under the ROS framework we’ve been working with.

2 Scenario

Your simulated robot, Robbie, lives in a 2D grid world. Robbie knows he lives on a grid, but doesn’t know his “absolute” position (coordinates) and orientation (direction) - all he knows is what he sees, what you tell him, and what he can remember. So, as he moves around the grid, he will need to keep track of his location internally.

In the program, each square of the grid contains a list of the names of the objects that can be found at that grid location. Object names will consist solely of lowercase letters - no numbers, punctuation, or spaces (so no multi-word names, like “big city” or “electrical motor”). Robbie will only be able to see information about close-by squares, and only in the direction he is facing. He can clearly see the square he is standing in and the one directly in front of him, has “fuzzy” vision of the square two spaces in front of him and the squares immediately to his left and right, and no vision of any other squares.

Robbie is capable of moving forward in the direction he is facing, as well as rotating 90 degrees to the left or right (so he can only face four possible directions).

Here is a table of what is contained at each grid square in our sample simulated world (the robot is at (2,2) with the rug, but it can’t see itself):

<table>
<thead>
<tr>
<th>desk</th>
<th>paper</th>
<th>apple</th>
<th>trashcan</th>
<th>chair</th>
<th>table</th>
<th>bowl</th>
</tr>
</thead>
<tbody>
<tr>
<td>bed</td>
<td></td>
<td></td>
<td></td>
<td>rug</td>
<td></td>
<td>dog</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>box</td>
<td></td>
<td>cat</td>
</tr>
</tbody>
</table>
3 Your Task

Building on the Input-Reasoner-Output structure from the previous projects, design a system that is able to move through a simulated 2D grid-based environment.

3.1 ROS Infrastructure

The ROS infrastructure is divided into five main parts (counting both language nodes as one “part”), connected as per the following diagram:

All messages passed should be Strings (you can pass other types of data, but we’ll keep it simple here).

3.1.1 Language

The language component consists of two nodes, LanguageInputNode and LanguageOutputNode. These nodes will be virtually identical to what you have used for the previous two projects - LanguageInput will just be constantly reading input from the terminal and forwarding what it reads into a ROS topic, and LanguageOutput will be displaying text to the user.

3.1.2 Reasoner

The ReasonerNode will be the “core” of the system. It subscribes to the input topics from LanguageInput and Vision, and publishes to the output topics reasoner_to_output, reasoner_to_vision, and reasoner_to_motion for LanguageOutput, Vision, and Motion.

In the ReasonerNode, you will need to interpret the user’s command, and then devise an action plan to carry out the command. Information about the
commands you will need to implement can be found in the “Commands You Need To Implement” section of this document.

The ReasonerNode should remember what it has seen in its environment as it carries out tasks. For example, if it sees a dog on a square, and then moves far enough away that it is no longer able to see that dog, it should still know that there is a dog there (you can assume that the world won’t change as you navigate through it, although in the real world this of course is not true). So, you will need to have some data structure storing this information.

Keep in mind that ReasonerNode will NOT have direct access to the grid, or even know its own absolute position/orientation - everything it sees is filtered through the VisionNode, and so it will need to build up an internal map of the world as it explores.

NEW: In order to ensure appropriate sequencing, the ReasonerNode will also need to get feedback from the motion_feedback topic of the WorldNode. You can either do this directly (by having the ReasonerNode subscribe to the motion_feedback), or pipe it through an extra subscribe-publish pair via the MotionNode (which requires use of an extra topic, but may be more realistic).

3.1.3 Vision

The VisionNode will not be using actual footage from a camera, nor will it be doing any actual sophisticated processing. Instead, it will be a “dummied” system that just looks at the data for the 2D environment and passes the information it finds back to the ReasonerNode.

The VisionNode should subscribe to a topic from the ReasonerNode which will make vision requests, and should publish its responses to another topic (which will be subscribed to by the ReasonerNode).

The data for the 2D environment is stored in a grid format, which is handled by the WorldNode. The VisionNode will need to subscribe to the world_info topic of the WorldNode, to actually receive this information. Each grid square contains a list of the names of the objects that can be found at that slot of the array. The VisionNode will relay information about each grid square according to the following rules:

- If the grid square is the square the robot is standing on, or is one square away from the robot in the direction the robot is facing, the full contents of the grid square will be relayed.

- If the grid square is two squares away from the robot in the direction the robot is facing, or if the square is one square away in an adjacent direction (so left and right, but not behind), a special string “UNKNOWNF”, “UNKNOWNL”, or “UNKNOWNR” (depending on whether the object is left, right, or center) will be relayed for each object in that grid square.

- In any other case, no information will be relayed.
The information should be passed as a String consisting of all the names of the relevant objects, separated by spaces.

This will mostly be handled by the WorldNode - all the VisionNode needs to do is make an appropriate request, and relay what it finds.

(In a real setting, the VisionNode would receive images from the world, and so you would have to do some further processing to extract the objects from the image, but here we’ve just explicitly encoded the objects.)

3.1.4 Motion

The MotionNode subscribes to a topic reasoner_to_motion from the ReasonerNode, and publishes to the WorldNode. Whenever the MotionNode receives a movement command from the ReasonerNode, it should “execute” that command by publishing a message to the robot_moves topic of the WorldNode.

The messages that can be published to the robot_moves topic are as follows:

- “forward” - causes the robot to move forward one grid square (if it won’t run off the grid)
- “left” - causes the robot to turn left by 90 degrees
- “right” - causes the robot to turn right by 90 degrees

3.1.5 World

The world represents the environment of the robot. In practice, this won’t be a ROS node, but instead will just be the actual world or a simulation, but here we’ll just represent it with a ROS node for convenience. We will provide you with a WorldNode, although the one we provide for development will be different than the one we test your code on (the topics will be the same, but the world grid will be different so you can’t just hardcode your whole scenario).

The WorldNode subscribes to two topics, robot_moves and request_world_info. It publishes to two topics, world_info and motion_feedback. The publishing occurs only upon receiving a request through request_world_info or robot_moves.

When the WorldNode receives a message to robot_moves, it updates the internal position of the robot accordingly (see specification in the Motion section above), NEW: and publishes a message to motion_feedback indicating what move was made (or an error if the robot attempted to move out of bounds).

When the WorldNode receives a message to request_world_info, it publishes the appropriate information to the appropriate topic. For this project, only one request type is possible, although you could imagine a more complicated setting:

- If the message says “get robot vision”, information about what the robot can see is requested. So, if the square in front of the robot contains a chair, a pencil, and a dog, and the square to the right of the robot contains some object, a message like “chair pencil dog UNKNOWNR” will be published to the world_info topic.
3.2 Commands You Need To Implement

1. “Move forward” - robot should move forward one square (MotionNode should send the appropriate signal to WorldNode).

2. “Rotate left” - robot should rotate left (MotionNode should send the appropriate signal to WorldNode).

3. “Rotate right” - robot should rotate right (MotionNode should send the appropriate signal to WorldNode).

4. “Do you see a/an X” (replace X by an object; you may assume all object names consist only of lowercase letters, and are only one word long) - robot should output text (in LanguageOutputNode) of either “Yes I see a/an X” or “No I do not see a/an X”, depending on whether an X is directly visible from the robot’s current position (on the same square as the robot, or on the square directly in front of the robot). This should be determined by consulting the VisionNode.

5. “Is there a/an X” (replace X by an object under the same stipulations as in the previous command) - robot should output text (in LanguageOutputNode) of either “Yes there is a/an X” or “I don’t know if there is a/an X”. This is the main challenge of the project.

First, it should check to see if the robot already knows about an X by looking at its stored knowledge.

Then, if it doesn’t currently know about an X, it should explore the UNKNOWNs it currently knows about until it either finds an X or runs out of UNKNOWNs. This will require issuing an appropriate series of movement and rotation commands (you may assume that all grid squares in the rectangle between the robot’s current position and the position of the UNKNOWN are traversable, to simplify things) to get to each UNKNOWN; new UNKNOWNs should be added along the way, as each movement should be accompanied by requests to the VisionNode to get the current view.

If it didn’t find an X, and doesn’t know about any other UNKNOWNs that could possibly be an X, it should quit exploring and say “I don’t know if there is a/an X”.

**CLARIFICATION:** Your robot should not do any extra exploration - we are simulating a “lazy” robot here. Once it has exhausted its list of UNKNOWNs (including the ones it finds on the way), that should mark the end of the exploration - it should not wander about randomly, or attempt to “cover every square” (nowhere did we even say that the grid is finite!). Also, when “looking around” after each movement, all we mean is that you should query the VisionNode - you don’t have to have the robot spin around 360 degrees after every movement or anything extreme like that.
4 What we have provided

We have provided the basic input-reasoner-output framework from previous projects, along with a sample WorldNode (which will act the same way as the one we will use to test your code, but with a different map). You will need to complete the ReasonerNode, and write your own VisionNode and MotionNode.

5 Hand-in and Grading

Submit your ROS nodes (the .py files) in a compressed zipped folder on ELMS, by 11:59PM on May 13th, 2017. Each of your nodes will be tested in isolation - we will dummy the messages being passed to and from the nodes. So make sure you don’t take any shortcuts!

6 Extending the project (OPTIONAL)

Here are some ideas for ways you could expand the project beyond what we require for this class. If you implement any of these, make sure that your code still satisfies the base requirements of the project, and document what you do clearly (include a text file or pdf in your submission describing what you did and how we can access or utilize it); if we deem that you’ve gone above and beyond to a sufficient degree, you may earn some extra credit points!

- Add in a “Go to the X” command, which will require the robot to first find an X, and then move to it.
- Add in some class-based reasoning, as in project 5. For example, if the robot sees a dog in front of it, and is asked whether it can see a mammal, it would be neat if it were able to conclude “yes”.
- Add a graphical display for the robot’s model of the world, so we can see into its head.
- Connect the vision node to a source of actual images (a camera, or just some dummy images), and have it do some real computer vision (with Chainer or similar).
- Connect the language input to your computer microphone and some speech parsing software, and pipe the language output to a text-to-speech program.
- Other options are possible! Use your creativity.
7 Hints

- Make sure that you follow the instructions and check your vision every time your robot moves - you might discover new UNKNOWNs along the way to an UNKNOWN.

- The previous bullet point only applies to vision in the natural direction of motion. So, you are NOT required to, say, spin around 360 degrees every time you move forward, but if you are on the way to an UNKNOWN and you happen to pass an object directly to your left that you haven’t seen before you should definitely record it as an UNKNOWN for later.

- You don’t need to know the grid size or boundaries, nor do you need to know the position of the robot on the grid. (Think about it - if the robot is on the edge of the grid, it has no reason to even try to walk off, since the only reason to move is to get closer to some UNKNOWN you want to investigate).

- The WorldNode we use to grade your code may have a completely different internal structure from the sample we provided. It might be stored as a dictionary or a graph, it might have a totally different set of helper methods or notation for indicating the robot’s position and direction, and it might even be finite or infinite in size. What WILL be the same is the protocol it uses for subscribing and publishing to ROS topics - this is what you should focus on. That being said, please feel free to modify the WorldNode to your liking for your own testing - just make sure those core ROS communications are preserved.

- As you may have gathered, it’s very possible that your search procedure results in you missing certain objects entirely. This is perfectly fine - people miss things all the time, and it’s impossible to search ”everywhere” in the case of an infinite or intractably large grid. So if you find that your robot is unable to find an object that is ”right there”, test it out yourself to see if that object really should have been visible at any point during its travels.

- EDIT: When moving towards an UNKNOWN, don’t move away from your goal. For example, if the UNKNOWN you are investigating is up and to the right of your current location, do NOT move down or left while getting to that UNKNOWN - you have no guarantee that that’s even allowed! If the UNKNOWN is directly to your left, you should just rotate left, rather than rotating right three times (this last sentence is not strictly required, but is encouraged in the spirit of the project).

- Even under the restrictions given, there will often be multiple routes you can take to investigate the various UNKNOWNs, depending on the order in which you decide to visit unknown nodes and depending on how you implement your pathing. It might seem like this would be hard to evaluate,
but in many cases you can at least know information like "the robot must encounter either X or Y in the process of searching for Z" or "the robot can never encounter A in the process of searching for B" (try working through the example in the spec and sample World, with the robot positioned at (2,2) and facing North, to see what I mean).

- You may be able to get away with storing less information than you receive from the vision node (e.g. once you’ve seen a dog, you don’t necessarily have to remember where it was, only that you’ve seen a dog), although this might change if you implement some of the optional extensions.