Homework 1: Architectures and ROS Due: Sept. 6, 11:59PM

Learning Objectives

- To implement ROS subscribers to receive sensor data and publishers to send actuator commands
- To identify ways in which ROS makes implementing agents more efficient
- To implement behavioral and layered approaches to single-agent architectures
- To identify differences between the execution of behavioral and layered approaches

Greenhouse Agent

In this course, you will be designing and implementing an autonomous greenhouse agent that runs on custom hardware and in a custom simulator that we provide you, which we call the *TerraBot*. In both the real hardware and the simulator, the environment will change on its own – plants will grow, water will evaporate, temperature and humidity will rise and fall, and ambient sunlight will occur daily. In addition, the agent can affect the environment by commanding the greenhouse actuators (LEDs, fans, and water pump), which, in turn, affects the sensor readings (light level, temperature, humidity, soil moisture, soil weight, and reservoir water level) which, in turn, affect the next time step in the agent's behaviors and resulting actuator commands. The greenhouse agent's job is to use the sensor information to determine how to control the greenhouse's actuators in order to grow plants effectively.

This first assignment is designed to help you understand the basic structure of the greenhouse agent as you 1) connect the greenhouse to the sensor data through a ROS interface, and 2) write two different agent architectures for it and assess how they perform on different starting conditions. We provide you with the basic underlying functionality - behaviors and schedule - for you to use. The architectures determine when the behaviors run. Over the course of the semester, you will need to understand how the components interact as you add on functionality to improve your agent's behavior. For now, we will focus on implementing the ROS sensor and actuator commands and analyzing the architectures. For this assignment, everything will be done using the TerraBot simulator.

Baseline Code

In addition to the TerraBot software (see below), the code you will need can be found on Canvas under the ROS assignment. In particular, we provide the following greenhouse-related files for you. We encourage you to spend time reading through the code to understand what it does before starting to implement your own code. Italicized files are ones you will need to edit. Note that for all assignments, the places that you need to edit are bracketed by the comments `BEGIN STUDENT CODE` and `END STUDENT CODE`.

- `hardware.py`, *ros_hardware.py*: We provide the interface and a blank implementation of "hardware" that talks to the sensors and actuators through ROS. You will implement the ROS commands to talk to the simulated (and real) sensors and actuators.
- `behavior.py`, `greenhouse_behavior.py`: We provide a behavior interface and implementations of the six behaviors that the greenhouse executes to keep sensor values in range - raising and lowering temperature, lowering humidity, raising and lowering soil moisture levels, and keeping grow lights on at ideal times. Each behavior must perceive the environment by analyzing sensor data and act by sending action commands to the hardware/simulator. We have implemented a `doStep` function that calls those functions in order, as well as `start` and `pause` functions that set the behaviors in a good state (e.g., turns off actuators) before it stops and starts running them.
- `schedule.py`: This file reads in the schedule provided in `greenhouse_schedule.txt` and can generate a plot of the schedule.
- *layers.py, greenhouse_agent.py*: We have provided code stubs for you to fill in for both the Behavioral and Executive layers. Then you will implement the `BehavioralGreenhouseAgent` and `LayeredGreenhouseAgent`

to use the sensors, actuators, behaviors, and layers to implement each architecture for the TerraBot. The main function runs the layers' `doStep` functions at each timestep until the execution is killed.

- `smarthome_behaviors.py`: used only in Part 4 for analysis.

Part 0: Familiarizing with the TerraBot Software (0 pts)

Loading Software

We provide you with a *VirtualBox* virtual machine preloaded with the software you need for this and future assignments (password: TerraBot). However, you are free to use your own virtual or non-virtual machine with Ubuntu, ROS (Noetic), and our agents loaded onto it.[1] Note that you will have to install the *VirtualBox Extension Pack*, as well (all can be found at https://www.virtualbox.org/wiki/Downloads). The distributed VM was created using VirtualBox 7.0.

NOTE: If use have a Mac M1 or M2 will need to use UTM (https://mac.getutm.app) – see the instructions at https://github.com/reidgs/TerraBot/blob/master/M1SETUP.md.

NOTE: The simulator uses a lot of CPU and memory, so make sure to set the parameters of your virtual machine to optimize performance. The instructions at the end of https://github.com/reidgs/TerraBot/blob/master/VIRTUALBOX.md give you suggestions. If your computer cannot handle the simulator, see if you can borrow one that is more capable.

Once you have the virtual machine set up, download the assignment code from Canvas. Place the folder in `~/Desktop/TerraBot/agents/` on the virtual machine. You can use either `FireFox` or `Chrome` to access the Internet. You can use either `vi` or `xemacs` to edit code. The login password is `TerraBot`.

The Robot Operating System (ROS)

ROS is a set of software tools designed to make it easier to implement real-time systems, including agents, by allowing the sensing, actuation, and all intermediate components to be decoupled from each other, but still communicate by sending messages. This design allows for the ability to swap out or add new components without having to redesign the entire system. While ROS enables you to write components in either C++ or Python, in this class, everything we write will be in Python.

Each individual component in ROS is called a `Node`. A ROS Node runs as a separate process on the computer and is meant to encapsulate some algorithm or set of computations. In ROS, we give Nodes names to identify which Node is running. If you look in the file `greenhouse_agent.py`, you will see that we added a `rospy.init_node` command that sets the names of two Nodes to `greenhouseagent_behavioral` and `greenhouseagent_layered`. Similarly, `TerraBot.py` has the same call. If you make a new Node for any reason, you would have to initialize it using the same command and a new name. Typically, a ROS node looks for input and produces output after it finishes its computation. It can do this continuously in a loop, or it can run to completion and exit. Because the Nodes run separately from each other, one Node exiting does not necessarily mean the rest of the Nodes will finish, as well.

In addition, ROS can run using the computer's clock (the default) or it can run using a simulated clock. The latter enables ROS to run faster, or slower, than real time, which can be useful for development and debugging. All the Nodes invoke a call `rospy.set_param("use_sim_time", True)` if the TerraBot is running in simulated mode.

The input into a Node can be in the form of keyboard or mouse clicks, which you likely already know how to write from prior programming classes, or it can be some *message* or *data* that another Node(s) generates. Similarly, the output can be screen-based, or it can involve sending messages to other Nodes. A Node can listen for input by *subscribing* to particular *topics* that match message names and data types. When a Node subscribes to a topic, it has to designate a function handler (termed a *callback*) that ROS will call

[1] See https://github.com/reidgs/TerraBot/blob/master/VIRTUALBOX.md for more details about requirements if you will load everything yourself.

asynchronously (e.g., in a separate thread from your Node's normal function) when ROS receives new data on the topic. ROS receives the data when any Node *publishes* or outputs to the topic. In this way, multiple Nodes can listen to any topic and the Node(s) that publish that topic (yes, multiple Nodes can publish to the same topic) do not have to specifically know who is listening. Similarly, multiple Nodes publishing the same topic do not need to know about each other. We mainly use ROS for its ability to support Nodes with these message-passing topic abilities.

You can learn more about the ROS architecture at http://wiki.ros.org/. In addition, we have recorded lectures that help you understand ROS and the TerraBot software (found on the **Terrabots, ROS, and Growing Plants** module on Canvas). You are strongly encouraged to go through the ROS tutorials (http://wiki.ros.org/ROS/Tutorials), as your understanding of the homework assignments will be much improved if you understand what ROS does and how it works.

The TerraBot

The TerraBot consists of two ROS Nodes (plus the agent you write). The `Arduino` Node receives sensor data from the hardware (or simulator) and publishes it to the `TerraBot` Node. The `Arduino` Node also receives actuator messages from the `TerraBot` Node and executes actuator commands to the hardware (or simulator).

The `TerraBot` Node subscribes to the `Arduino`'s messages, optionally performs some functions on that data, and then publishes them for your agent. In particular, the `TerraBot` Node can add noise or simulate sensors failing, which will be used to test the robustness of your agent. The `TerraBot` Node also subscribes to messages your agent will publish, and forwards them on to the `Arduino` (changing the topic name, so that there is no confusion about which messages are meant for the `TerraBot` and which are meant for the `Arduino`). The `TerraBot` also invokes `roscore`, which is the underlying infrastructure that supports the message passing. A diagram of the connections between Nodes and the messages are at: https://github.com/reidgs/TerraBot/blob/master/system_diagram.jpg.

To run the TerraBot simulator in the virtual machine, open a terminal (ctl-alt-T), cd to the `Desktop/TerraBot` directory and type: `python TerraBot.py -m sim -g`. The `-m sim` option starts up the simulator, which starts up a "fake" Arduino Node, instead of connecting to the real hardware. The `-g` argument tells it to start the graphics, to show you the simulated hardware.

Since it takes time to start `roscore` and the Nodes, you'll see it print "waiting for nodes" and then "System started". While you don't see anything happening on screen, nodes are still passing messages back and forth to each other all the time. You can see what all the nodes are publishing by opening a new terminal window and typing: `rostopic list`. You can pick any of these topics and watch the messages by typing: `rostopic echo /[name_of_topic]`. This command subscribes to a message and then prints it to the terminal. For instance, try the `/humid_output` topic, which the TerraBot publishes about the two humidity sensors for your agent to use in your behaviors. You can see that the data is held in the data variable of a data class (`data.data[0]` and `data.data[1]`) – one for each humidity sensor in the greenhouse. Note that plants and greenhouses change very slowly so you won't see the numbers change very often. Type Control-C to quit the `rostopic` echo.

You can see from the system diagram that your agent should subscribe to the TerraBot's "output" messages and publish the "input" messages. You should *NOT* subscribe to the Arduino's "raw" messages. In future assignments, we will be adding noise to the sensor data and we do not want you listening to the raw data. In this assignment, you will write the function callbacks to save the subscribed data so that your behaviors can use them and then publish commands to send the actuator data back to the TerraBot (and then to the real or simulated Arduino Node). After you write the callbacks, you will then implement the architectures and main ROS loop.

To quit the TerraBot program, type 'q' then Enter. If you Ctrl-C out of the program, run `./cleanup_processes` to ensure that the separate Nodes (processes) have all been properly terminated – ROS gets confused if there are multiple Nodes of the same name running concurrently.

Part 1: ROS Subscribers and Publishers (30 pts)

Now you will fill in `ros_hardware.py` to subscribe and publish to the TerraBot. We will walk through one subscriber for the light and one publisher for the LEDs, then you can complete the rest of them.

In the `ROSSensor's init` function, you will see a series of subscribers declared by calling:

```
rospy.Subscriber('light_output', Int32MultiArray, callback_name)
```

In this case, ROS will look for all messages on the topic `light_output` (note: it is an output of TerraBot, though it is an input to your agent), which have the datatype `Int32MultiArray` (32 bit integer arrays), and each time some Node publishes a message of that type, it will invoke `callback_name` and pass in the data. **Because these callbacks are run in a separate thread from the rest of your agent, you want them to be very fast – they should only save the data you need and then return.** Define a function with your callback's name, taking as input a data variable. Repeat this pattern for `'temp_output'`, `'smoist_output'`, `'level_output'`, `'weight_output'`, and `'humid_output'`.

Values of data.data: For light, humidity, weight, soil moisture, and temperature, the values are arrays (lists). Save both the raw data values and their average to the respective class variables defined for you. For example, `self.light_level_raw` should be set to `data.data` and `self.light_level` should be set to the average of the two. For the first few assignments, the behaviors will use only the average value, the raw values will be used later, when we get into execution monitoring.

The one exception is sensing the water level in the reservoir. There is only a single sensor measuring the water level. Its type is `Float32` and the raw value and the perceived value should be the same.

That's it. Your agent can now listen for messages. You should now have 6 Subscriber calls (we provided) and 6 callbacks (you implemented), setting 12 state variables.

To test your subscribers, run the TerraBot in one terminal with a speedup of 100 (no need to run the graphics), and in a second terminal window run: `python autograder.py -p 1`. The autograder will let you know if your subscribers are working correctly.

Now, let's set up our publishers in the `ROSActuators` class. In your `init` function, you will need to create a publisher for each topic you would like to publish. To do this, we call:

```
self.actuators['led'] = rospy.Publisher('led_input', Int32, latch=True, queue_size=1)
```

which tells ROS to send messages to the 'led_input' topic of type 32-bit integer (note: it is an input to TerraBot though it is an output from your agent). Additionally, it should keep that value (latched) even if your agent stops publishing the message and it should notify all the callbacks listening to that value each time it is published rather than queue the value for later. The publishers are maintained in a dictionary (`self.actuators`), so that when you need to issue an actuator command, you will have easy access to the publisher (see below).

In `doActions(actions_tuple)` the input is a tuple of `(behaviorName, time, act_dict)`, where `act_dict` is a dictionary whose keys are actuator names ('led', 'fan', 'wpump', 'ping') and whose item values are the commands to be sent (LEDs get sent integers, the rest get sent Booleans). In order to publish, you need to call `self.actuators[name].publish(val)`, where `name` is a key in the `act_dict` and `value` is the associated value. This should be done for all items in the dictionary. You should also save the current state of the actuators to the `self.actuator_state` dictionary as you publish the values.

To test your publishers, run the TerraBot with a speedup of 100, and in a second terminal window run: `python autograder.py -p 2`. The autograder will let you know if your publishers are working correctly. In addition, if you run the graphics (with the -g option) you can see in the graphics (top left) that all three actuators (fan, water pump, and LEDs) should turn on and off periodically (the pump should turn on and off every 2 seconds, the fans every 4 seconds, and the light level should change every 6 seconds). You can also turn on your audio output to hear the fans and water pump!

Part 2: Behavioral Architecture Approach (30 pts)

In this part, you will implement a simple Behavioral architecture, in which the agent runs one step of each behavior at a time in a loop. As you learned, in the Behavioral approach each of the behaviors are responsible for:

1. querying the sensor class to get the latest sensor data;
2. analyzing that data for perceiving the environment;
3. determining which state they are in (e.g., the temperature is too low, too high, or perfect); and
4. using that state to determine what actuators should be on and off.

Your first job is to implement the `BehavioralLayer` in `layers.py`. In the `init` function, you will find the sensor and actuator classes that should be set in each behavior and list of behaviors that could be run. The variable `self.enabled` is a list of behaviors that are currently enabled. There, you can also initialize any additional variables that you might need for subsequent functions.

The next step is to write code to start a behavior (function `startBehavior`). Given a name, get the associated behavior (`getBehavior`) and, if there is a behavior with that name and it is not currently enabled (`isEnabled`), then start it and add it to the list of enabled behaviors (`self.enabled`). Similarly, implement `pauseBehavior` to pause the named behavior, if it exists and is currently enabled. Make sure to update your data structures accordingly.

We have provided you with a `doStep` function that, for each enabled behavior, gets the sensor data, analyzes the data ("perceives" the environment) and then acts based on those perceptions (see `behavior.py` for more details).

Once your `BehavioralLayer` is implemented, you can implement the `BehavioralGreenhouseAgent` (in `greenhouse_agent.py`. In the `init` function, you need to instantiate the `ROSSensors, ROSAcutators`, and all of the behaviors in `greenhouse_behaviors.py` and `ping_behavior.py`, saving each of them as instance variables of the `BehavioralGreenhouseAgent` class. Then, instantiate the `BehavioralLayer` by passing it the `ROSSensors, ROSAcutators`, and list of behaviors.

Take some time to understand each of these classes. Note that the ping behavior's purpose is to tell the TerraBot that your agent hasn't crashed. It is important to keep it running. Your agent should pass the list of instantiated Behaviors to the BehavioralLayer.

The `main` function of the agent first checks that sensor data has been received, then starts up all the behaviors and runs in a loop until ROS quits. At each iteration, it runs a step of the behavioral layer and then pauses for a second (we use `rospy.sleep(1)`), because it takes into account if your simulator is running in real time or in faster than real time.

At this point, you can run your ROS-based greenhouse agent. In one terminal window, run:

```
python TerraBot.py -m sim -g -s 500
```

and in another window run:

```
python greenhouse_agent.py -B -m sim
```

You should see that all of the behaviors get initially enabled (and never get disabled). You should also see actuators turning on and off and, after a few days of simulated time, plants appearing. Don't forget that you can also call `rostopic echo` to confirm that your agent is publishing all of its messages.

Also, note that when the pump is on, the simulator automatically changes to real time (speedup of 1) and when the fan is on it reduces to a speedup of 100 – this is because those changes happen relatively quickly, and the behaviors need to get sensor data at an appropriate rate.

Part 3: Layered Architecture Approach (30 pts)

In this part, you will implement the Layered agent approach. In the Layered approach, the agent uses three layers:

- **Planning:** generates schedules for the Executive. The schedule is a dictionary mapping behavior names to a list of tuples of start and end times. This class is provided to you for your convenience.
- **Executive:** monitors the schedule, enabling and pausing behaviors at scheduled times.
- **Behavioral:** runs each enabled behavior. Basically, the same as the Behavioral architecture.

In the `BehavioralLayer` class in `layers.py`, check to make sure you have implemented the `startBehavior` and `pauseBehavior` functions so that they call `start()` or `pause()` only if they are currently paused or enabled respectively. While you already implemented these functions in Part 2, the Behavioral approach does not pause behaviors, so this functionality has not yet been tested.

In the `ExecutiveLayer` class in `layers.py`, implement the `doStep` function, which should determine the behaviors to enable or pause, based on the schedule provided and the current time. The schedule is a dictionary mapping behavior names (not the actual behaviors) to a list of tuples of start and end times. The behaviors that should be running are those in which start time $\leq$ current time $<$ end time, for any of the start/end tuples associated with that behavior (NOTE: the argument passed into the `doStep` function is the time *in seconds* since midnight, but the schedule uses time *in minutes* since midnight).

Your `doStep` function should ensure that behaviors are started exactly once and stopped exactly once per tuple in the schedule. You may use any additional data structures and functions if you choose (initializing them in the class's `init` function). It is important to note that behaviors that are no longer scheduled to run should be stopped (paused) before any new behaviors are started (enabled). This is because the planned schedule assumes that certain behaviors should not be operating at the same time, and that pausing one behavior puts the TerraBot into a state that other behaviors assume exists when they start (e.g., that the pump or fans are off).

Once you have implemented the Executive layer, you need to write the `LayeredGreenhouseAgent` (in `greenhouse_agent.py`). As with the `BehavioralGreenhouseAgent`, the `LayeredGreenhouseAgent` needs to initialize the sensors, actuators, behaviors (including ping), and all the layers and connect them appropriately using provided setters. In particular, you need to `setBehavioralLayer` and `setPlanningLayer` for the executive, and `setExecutive` for the planning layer. Don't forget to run `getNewSchedule()` (defined in the `PlanningLayer`) to read in the schedule that the planner will use. In the Layered approach, the Executive will start and stop the Behaviors at scheduled times, so unlike the `BehavioralAgent` you should not start all behaviors before running the main loop. In the `LayeredGreenhouseAgent` main loop, you need to run each layer's `doStep()` at each time step. Typically, the layers would be run as separate threads each at a different rate, but for simplicity the greenhouse agent runs them sequentially at each iteration of the main loop.

At this point, you can run your ROS-based greenhouse agent. In one Terminal window, run:

```
python TerraBot.py -m sim -g -s 500
```

and in another window run:

```
python greenhouse_agent.py -L -m sim
```

You should observe that your behaviors are enabled/started and disabled/paused at the right times in the schedule. You should also see the actuators turning on and off and, after a few days of simulated time, plants appearing. Don't forget that you can also call `rostopic echo` to confirm that your agent is publishing all of its messages.

Part 4: Compare and Contrast (10 pts)

We have provide a file (`assignment1.tst`) that monitors the state of the greenhouse and reports whether the various test conditions are being met. For this part, run the TerraBot with the test file:

`python TerraBot.py -m sim -g -s 500 -t [path_to_file]/assignment1.tst`. **Note:** This test

takes fairly long to run (3 simulated days), but if you run faster than 500 speedup, outcomes tend to get unpredictable. You may find that even 500 speedup is too fast for your own computer, in which case reduce to around 200.

In another terminal first run your Behavioral agent. Keep running until the TerraBot test file quits. Then, run the TerraBot and your Layered agent. After each test, observe which tests succeeded and failed. Note that the test output indicates when each test rule is activated and whether it ends in success or failure. In addition to indicating success or failure, the output indicates which rule succeeded or failed.

Then, in a separate `readme.txt` file, answer the following questions.

1. Does the order that the behaviors are run affect the `BehavioralAgent`'s actuators? Given that the behaviors all run simultaneously, what can happen to the actuator values at times?
2. How is the schedule organized? What behaviors are allowed to run at the same time? Why would we have organized it that way? How does schedule affect the Layered approach's actuator commands?
3. Which architectural approach is more reasonable for the greenhouse agent, and why?
4. Consider a new smart home with a smart coffee maker, dimmable lighting, and heating/air conditioning (called hvac). The smarthome has access to limited sensor data – time, coffee cup size selected, coffee pod present, and smart home occupancy (whether there are people in the house: True=1 or False=0). We created three behaviors controlling each of the three actuators viewable in `smarthome_behaviors.py`. Inspect the `smarthome_behaviors.py` code, and think about the similarities and differences between this application and the greenhouse. Which architecture may be more suitable for this smarthome application and why?

Submission

Submit the four files – `greenhouse_agent.py, ros_hardware.py, layers.py`, and `readme.txt` – to Canvas. When we copy these files into a blank assignment directory, they should run without error. The assignment will be graded out of 100 points. Parts 1, 2, and 3 will be graded using the test files. Part 4 will be graded manually. Make sure the `readme.txt` file is easy to read and understand.