Homework 3: Monitoring and Testing (Group) Due: Fri. Sept. 26, 11:59PM

Learning Objectives

- Monitor for optimal light exposure
- Perform symbolic, logic-based diagnosis
- Create and validate tests of the TerraBot agent
- Characterize simulated and real sensor data, using a logging facility
- Design a monitoring approach for a real-world scenario
- Prepare your agent for grow period deployment

This is a **group** assignment consisting of eight (8) parts. The first three parts are related to monitoring and testing. The next three parts are preparation for the first grow period. Part 7 is a scenario problem, similar to what will be included on the midterm and final, and Part 8 is integrating code to prepare your agent for deployment. You should decide with your teammates how to divide up the work on the first six parts, but everyone **is to work together** on Parts 7 and 8. We suggest that each person do one of the first three parts and one of the next three. We also recommend using Github for collaborating on code with your teammates.

The purpose of this assignment is to give you the opportunity to work together as a team to test and tweak your agent code in preparation for the upcoming two-week, hands-off deployment. These activities are based on feedback that we got from previous students on ways to help you think about potential tests to run. Additionally, one problem with a long deployment on remote hardware is that it is hard to debug when something does go wrong. We are also asking you to set up several strategies to monitor your agent over time to make it easier to debug.

NOTE: This is a long, involved assignment, with several parts that must be integrated together before the first grow period begins on September 29 (just three days after the assignment is due!). You have only 11 days to complete the assignment, so START EARLY AND WORK AS A TEAM!!

Much of the assignment makes use of *monitors*. Monitors are similar to behaviors, except that they are enabled on a fixed period (e.g., every two minutes). Architecturally, monitors reside in the executive layer, and so they have access to both the schedule being executed and the behaviors that exist in the behavioral layer. Monitors can access sensor data, but they don't take actions, as behaviors do. Instead, they can modify the behaviors (by adjusting parameters) or can inform the executive when things are going wrong (this latter capability will not be explored in this assignment). In addition, monitors do not typically use FSM's, since they tend to do the same thing each time they are enabled. Monitors can access the sensors (`self.sensors`), the sensor data (`self.sensordata`), the commanded state of the actuators self.actuator_state, the executive layer (`self.executive`), the last time the monitor was invoked (`self.last_time`), and the amount of time that has passed since the last time the monitor was invoked (`self.dt`).

Download and unzip the assignment code from Canvas. Copy the files `behavior.py, greenhouse_agents.py, hardware.py, layers.py, ros_hardware.py` and `schedule.py`, from your `ROS_HW` directory, and copy `greenhouse_behaviors.py` from your `FSM_HW` directory.

The file `monitor.py` contains the general `Monitor` class definition, and all your agent's monitors should be a subclass of `Monitor`. It would help you to study the class definition to understand what it does, what functions need to be defined, and how it differs from the `Behavior` class. In particular, note that you have access to both the sensor values (using `self.sensordata`) and the actuation state (using `self.actuator_state`).

Part 1: Monitoring Insolation (20 points)

For this part, you will modify `light_monitor.py` in the areas delineated by `BEGIN/END STUDENT CODE` and the files `greenhouse_behaviors.py` and `greenhouse_agent.py`, as detailed below.

Up until now, we have had the lights on for an arbitrary amount of time. In actuality, plants do best when they get enough light, but not too much. In this part, you will try to reach a target value of daily light. In particular, you will need to keep track of the amount of light received during a day (the *insolation*), estimate

how much opportunity there is for lighting the rest of the day, and modify the light limits so the `Light` behavior does not add too much light.

`LightMonitor`, defined in `light_monitor.py`, is a subclass of the `Monitor` class (see above). It takes a target value, which is the amount of light per day that the plants should receive. This is a combination of the light from the LEDs and ambient light from outside sources. We have provided a log file of ambient light data taken over the day (`grader_files/ambient.log`) and a function to input it (`read_log_file`). The function `integrate_ambient` computes the ambient light received **per hour** given start (`ts`) and end (`te`) times (note that we are assuming that the ambient light is roughly constant from day to day).

You will need to complete the `perceive` and `monitor` functions. The perceive function should set all of the sensor data needed by the monitor, using `self.sensordata`. The monitor function should:

- Keep track of the insolation received so far. **Note:** the `target` value is the amount of light over 24 hours, so you need to divide the perceived light level by 3600 to get it to the same units.
- Compute the optimal light level, given the amount needed to reach the target, the estimated remaining ambient light, and the available time left for the LEDs to be on. We have provided two helper functions: (1) `non_lighting_ambient_insolation`, which, given a start and end time, returns how much ambient light is estimated to be received between those two times when the `LightBehavior` is **not** running, and (2) `lighting_time_left`, which, given a start time, returns how much time is left in the schedule for the `LightBehavior` to run. Using these helper functions you can estimate how much natural light will be received during the times when the `LightBehavior` is not running and compute what the light level (per second) needs to be to reach the target, based on light received so far and what remains to be received.
- Set this new optimal limit. You will need to modify your `Light` behavior in `greenhouse_behaviors.py` to add a function for updating the optimal level (i.e., setting a new value for `self.optimal_level`) that can be called from your light monitor. Be sure to create a deadband around the computed optimal, so the `Light` behavior is not constantly changing the LED values.
- Reset the insolation every day.

Note that the `RaiseTemp` behavior may also add light, depending on conditions. You can either estimate how much light that behavior will add, or let the monitor adjust the light level dynamically, when any "unexpected" addition of light from the `RaiseTemp` behavior occurs.

Once the monitor has been implemented, edit `greenhouse_agents.py` to import `light_monitor` and add the following line at the end of the `LayeredGreenhouseAgent`'s `__init__` function:

```
self.getExecutiveLayer().setMonitors(self.sensors, self.actuators.actuator_state,
                                      [light_monitor.LightMonitor()])
```

where `self.actuators` is the variable you used to save the value of `ROSActuators()`. This adds an instance of `LightMonitor` with the default period of 100 seconds, which is sufficient for this problem.

To test, run: `python TerraBot.py -m sim -s 250 -t <dir>/insolation.tst` (replace `<dir>` with the location of the test file) and run: `python greenhouse_agent.py -m sim -L`. The test will be successful if the insolation is within 3% of the target value.

Part 2: Diagnosis (30 points total)

The objective of this part is to model the greenhouse hardware using a symbolic (logic-based) approach and use that model to diagnose observed exceptions. **For this part, you will modify `adder.py` and `diagnosis.py` in the areas delineated by `BEGIN/END STUDENT CODE`.**

We are using Google `ORtools` for this part, although you are free to use another logic-solver if you prefer. In particular, we have developed code (in `cnf.py`) for converting propositional logic to conjunctive normal form. (CNF), which can then be solved using the ORtools SAT solver. We will be using `ORtools` in subsequent assignments, so it is suggested that you become familiar with this tool.

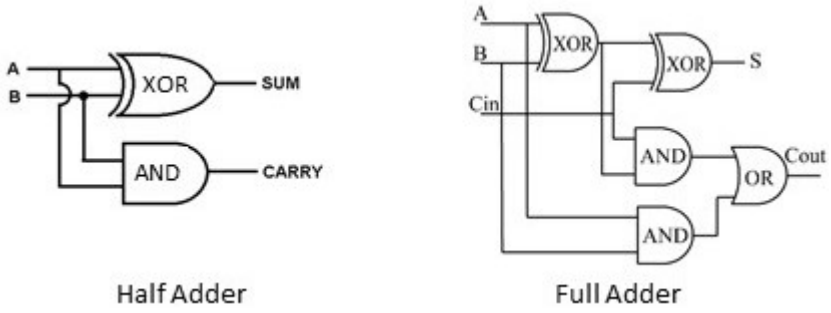

Figure 1: One-Bit Half and Full Adder

Step 1: Adder Circuit (3 points)

The objective of this part is to illustrate how propositional logic can be used to represent hardware and how a SAT solver, such as provided by `ORtools`, can be used to find solutions that satisfy the logical formulae.

One-bit adder circuits (see Figure 1) can be chained together to form multi-bit adders. The half adder is used for the low-order bit, taking in two bits and computing the sum and carry-out. The full adder is used for the rest of the bits, taking in two bits and a carry-in, and computing the sum and carry-out of the three inputs.

The file `adder.py` has the code needed to define half and full adders, chain them together to form n-bit adders, and solve the forward problem (giving n-bit inputs, calculate the n+1-bit output). You can run `python adder.py <addend1> <addend2>` (where the addends are 0-7) to see the result. Study how the logic of the adders are defined, corresponding to Figure 1, how `ORtool` variables are defined, and how the function `add_constraint_to_model` works to add CNFs to a `cp_model` (a part of the `ORtools` SAT solver toolkit).

For this step, **you are to implement a function that goes the other way — computing all possible inputs that can lead to a given output.** The beauty of defining things in this logic-based way is that the exact same model can be used in both directions. The only difference is how the solver works. In particular, for the forward direction, there is only one feasible solution; for the backwards direction, there are many possible solutions (e.g., 6 can be achieved by 0+6, 1+5, 2+4, etc.).

The idea here is to solve using `SearchForAllSolutions`, supplying a `CpSolverSolutionCallback`, which is invoked each time the solver finds a solution. **You are to fill in the code in `output_input_adder` function and the `SolutionCollector` class.** See https://developers.google.com/optimization/cp/cp_solver for hints on how to do this. The `output_input_adder` function should return a list of tuples, each of which is the list of bits in the first addend and the bits in the second, e.g.:

```
[([1, 0, 0], [1, 0, 0]), ([0, 0, 0], [0, 1, 0]), ([0, 1, 0], [0, 0, 0])]
```

Note that the lower-order bits come first, so this list shows the three different ways that 2 ([0, 1, 0]) can be achieved (i.e., 1+1, 0+2, 2+0).

You can test this step by running `python autograder.py -p 2 -s 1`.

The objective of the next 3 steps is to create a model of the greenhouse's hardware and test some simple cases where the hardware is working correctly.

Step 2: Modeling the Greenhouse Relations (6 points)

The greenhouse hardware (see Figure 2) should be modeled using the following set of *components* and *relations*.

- **Objects:** `Outlet, Rasp-Pi, Arduino, Power-Board, Sensor-Board0, Sensor-Board1`
- **Actuators:** `Fans, LEDs, Pump`

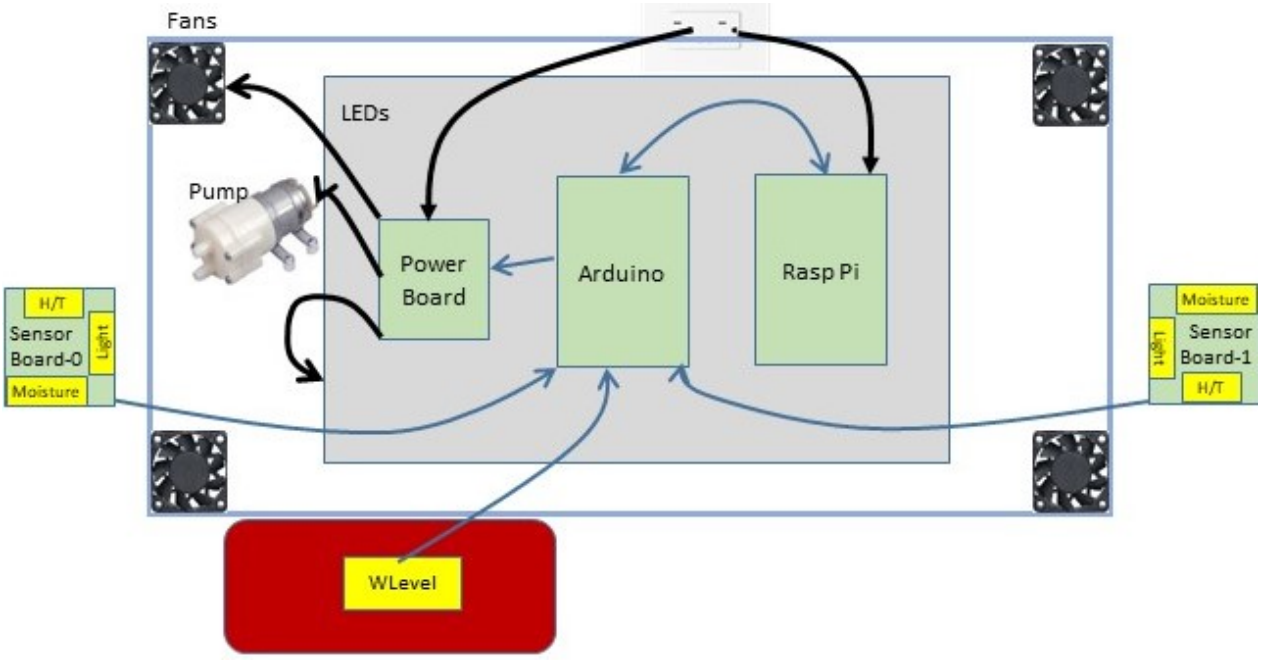

Figure 2: The Greenhouse Hardware

- **Sensors:** `H-T0, H-T1, Light0, Light1, Moisture0, Moisture1, Wlevel`
- **Components:** Is the *union* of objects, actuators, and sensors
- **Relations:**
  - `working(<component>)`
    `True` if the component (e.g., object, actuator, or sensor) is currently operational
  - `connected(<component0>, <component1>)`
    `True` if the connection between the components is not broken. The connections are all the arrows in the diagram, *plus* each `H-T` (humidity-temperature), `Light`, and `Moisture` sensor is connected to its appropriate `Sensor-Board`. For instance, `connected(H-T0, Sensor-Board0)` represents a signal connection and `connected(Power-Board, LEDs)` represents a power connection. The component at the tail of the arrow is `component0`; the component at the head of the arrow is `component1`. **Note #1:** for the connections between a sensor and the sensor board, make the sensor `component0`. **Note #2:** since the connection between the `Arduino` and `Rasp-Pi` is bidirectional, you need to have two connected relations for them.
  - `powered(<object/actuator>)`
    `True` if `Outlet, Rasp-Pi, Power-Board`, or an actuator is receiving power (note: the Ardunio is not explicitly powered).
  - `signal(<sensor/actuator>, <component >)`
    `True` if the component has either received or generated the signal (the sensors generate *sensor* signals and the `Rasp-Pi` generates *actuator* signals). For instance, if `signal(H-T0, H-T0)` is `True`, it indicates that the humidity-temperature sensor has generated a reading. Similarly, if `signal(Fans, Arduino)` is `True`, it indicates that the Arduino has received the "fans on" signal.
  - `expected-result(<actuator>)`
    `True` if turning on the actuator (e.g., `signal(Pump, Rasp-Pi)`) leads to the expected sensor reading (see below for more information).

Note that while the greenhouse actually has four fans, you need to model only one of them; the others are equivalent. Thus, given the diagram above, you should have a total of 78 relations: 16 `working` relations, 17 `connected` relations (12 `signal` connections and 5 `power` connections), 6 `powered` relations, 36 `signal`

relations, and 3 `expected-result` relations. Here is how we get 36 `signal` relations: each of the 7 sensors *generates* a signal; each of the two sensor-boards *receives* 3 signals; the `Arduino` and `Rasp-Pi` each receive 7 sensor signals; the `Rasp-Pi` generates 3 actuator signals; and the `Arduino` and `Power-Board` each receive 3 actuator signals. Since there are so many relations, we recommend writing helper functions that can be used to create different relations of a given type (such as the with `half_adder` and `full_adder` examples).

To help you, we have supplied the code for the `working` relations in `create_working_relations`. You need to fill in the functions `create_connected_relations, create_powered_relations, create_signal_relations,` and `create_expected_result_relations` in `diagnosis.py`. The `create_<x>_relations` functions should all add `ORtool` variables to the `variables` dictionary that is passed in (using the helper functions `create_relation` and `create_relations`). The dictionary should have 78 entries, one for each relation described above.

Note that each relation should be represented as a Boolean variable, whose name is the relation. To enable the autograder to work correctly the relation names must be in a standard format - to that end, use the provided helper functions to generate the relation names (e.g., `working('Outlet')`).

Use Figure 2 as a guide to help you understand which componenets are connected to each other and how electricity and information flow through those connections.

You can test this step by running `python autograder.py -p 2 -s 2`
Note that the autograder will tell you if you have missing or extra relations. While it's a good idea not to have any extra relations, make sure you focus on adding any missing relations, based on the description above. Note also that without any changes, the autograder will give you 16 out of 78, since we have provided the 16 working relations.

Step 3: Modeling the Greenhouse Constraints (12 points)

To model the dynamics of how the greenhouse hardware actually works, you need to define a number of **constraints**. The following constraints are needed to sufficiently model the way electricity and information flow in the greenhouse:

- For each of the 6 `powered` relations, you should have a constraint that represents the condition in which it is powered. Specifically, the `Outlet` is always powered (i.e., `powered('Outlet')` is `True`, but the `Rasp-Pi` and `Power-Board` are powered iff they are connected to the `Outlet` (black arrow in the diagram) *and* the `Outlet` is working. An actuator (`LEDs, Fans, Pump`) is `powered` iff it is `connected` to the `Power-Board` and the `Power-Board` is `powered, working`, and the `Power-Board` has received the appropriate signal for that actuator (e.g., `signal(LEDs, Power-Board)` is `True`). Note: `connected(comp0, comp1)` being `False` indicates that the physical wire associated with that connection is broken.
- For each `signal` that is *received* by a component, you should have a constraint that represents the condition under which the signal is received. Specifically, a signal is successfully sent from `component0` to `component1` iff the components are `connected`, `component0` is `working`, and `component0` has either received or generated the signal. Note that while there are 36 `signal` relations, you will need only 26 constraints, since 10 of the signals are generated, not received.
- For each of the 7 **sensors**, add a constraint that a sensor reading is generated (e.g., `signal(Light1, Light1)`) iff the sensor is `working`. Note that the sensor name and the signal it generates is the same.
- The final set of constraints relates to determining if an actuator is `working`. If the `Rasp-Pi` generates an actuator signal, then `expected-result` is `True` iff the actuator is `powered` and `working` and the `Rasp-Pi` receives one of the signals that is associated with that actuator (specifically, signals `H-T0` and `H-T1` are associated with the `Fans`; `Light0` and `Light1` are associated with the `LEDs`; and `Moisture0`, `Moisture1`, and `Wlevel` are associated with the `Pump`). So, for instance, `expected-result(Fans)` iff `powered(Fans)` and `working(Fans)` and `(signal(H-T0, Rasp-Pi) or signal(H-T1, Rasp-Pi))`. In other words, if the `Rasp-Pi` signals for an actuator to turn on, it can check whether the request has been met by seeing an appropriate sensor change caused by the actuator. Note that latency of effects is ignored, which is different from the actual sensors.

So, overall, you should have 42 constraints – 6 for the `powered` constraints, 26 for the received `signal` constraints, 7 for the sensor generation constraints, and 3 for the `expected-result` constraints. Again, since there are so many constraints, we recommend writing helper functions that can be used to create different constraints of a given type.

You need to fill in the functions `create_signal_constraints`, `create_sensor_generation_constraints`, and `create_expected_result_constraints` in `diagnosis.py`. The `create_<x>_constraints` functions should create CNF formulae and use `add_constraint_to_model` to add them to the `ORtools model`. To help you, we have supplied the code for `create_powered_constraints` (which uses two helper functions: `create_powered_constraint`, for the connection between the `Outlet` and the `Rasp-Pi` and `Power-Board`) and `create_powered_actuator_constraint` (which adds the constraint for actuators).

Run `python autograder.py -p 2 -s 3` to test whether the constraints are all defined correctly. This is tricky to autograde, so it may be that you pass this step but fail in the next step. If that's the case, let us know and we can try to improve the autograder to catch edge cases.

Step 4: Model Inference (2 points)

This step *should* be simple if you passed steps 2 and 3. The idea is to put all the constraints together to see whether the system can make a given inference about what would happen in a specific scenario. If you passed steps 2 and 3, but are failing on this step, it likely means that the autograder for step 3 was not complete enough. Contact us, in that case, and we'll try to improve the autograder.

Run `python autograder.py -p 2 -s 4` to test whether your complete model is defined correctly.

Step 5: Diagnosis (7 points)

Steps 2-4 test whether the model can correctly infer what components are working and connected when certain signals are received by the `Rasp-Pi`. In this part, you will use the same model to create a diagnostic algorithm that indicates what sets of components may *not be* working when some expected signals are *not* received.

For this part, use the approach in Step 1 to create a `SolutionCollector` that is notified when the SAT solver finds a solution. For each solution, the `SolutionCollector` should create a diagnosis consisting of the set of `working` and `connected` relations in each solution that are `False` (make sure each diagnosis is a Python set). When all solutions have been found, it should return a list of possible diagnoses, ordered in ascending length of the diagnosis; that is, all single-fault diagnoses should be first, followed by two-fault diagnoses, etc. Make sure not to include diagnoses that are **supersets** of other diagnoses (that is, they have the same plus more components than another diagnosis, since the simpler/smaller diagnosis already explains the problem). **Fill in the `diagnose` function, which takes a set of observations (relations known to be `True` or `False`), adds them to the model, and then solves for all possible diagnoses.**

Run `python autograder.py -p 2 -s 5` to test whether the diagnosis works correctly.

Part 3: Creating Test Files (20 points)

As discussed in class, extensive testing is extremely important before deploying autonomous agents. We have provided a method for describing tests that the agent needs to pass and automatically determining whether those tests succeed. Once the tests are designed, they replace a lot of the manual effort needed to do the needed testing.

To facilitate this, we provide two types of files: **baseline files** and **test files**. **Baseline files** start the simulator in different states by enabling you to set the sensor and actuator parameters to desired values. They enable you to test particular cases that might not occur frequently, and they enable you to start in a state that may otherwise take the simulator a long time to reach. Baseline files also enable you to designate how your plants are looking (lanky-tall and unstable due to lack of light, healthy-green and full, or droopy-leaves

due to lack of water). You can see examples of baseline files in `TerraBot/param/default_baseline.bsl`. The `TerraBot/README.md` file contains a full description of the format of baseline files.

Test files enable you to write the expectations that you have for your agent. For example, under what conditions should it always turn on the camera, fan, lights, and pump? How much time from the onset of these conditions do you expect the behaviors to kick in (**hint**: don't forget that when you run in speedup mode, seconds can just fly by)? The slides from the September 17 lecture and the directories `ROS_HW` and `MON_HW` contain `.tst` files that you can use as examples. The `TerraBot/README.md` file contains a full description of the format of test files.

The limits and optimal growing conditions for our plants are outlined in `TerraBot/agents/limits.py`. The *limits* are the allowable sensor values and the *optimal* is when the behaviors should stop actuating. Think about the growing conditions that you would want to test in order to ensure your agent is doing its job. Does the fan come on at the right times? The lights? The water pump?

For this part, **you are to create eight (8) tests.** You may choose to create different baseline files as well. Each test can be in its own `.tst` file, or you can put multiple tests into a single file. You can also define a test file that `includes` other test files (see `TerraBot/README.md`). The tests should enable you to ensure that your behaviors are working properly. Make sure that your tests cover a wide variety of situations that might occur when your agent is deployed (e.g., it is not acceptable to test whether the fans turn on correctly at 8 diffferent times of the day). Document all of your files using comments (#) for what your tests are testing, what you expect the starting state to be, and what you expect your actuators to do as a result. Note that your tests don't have to succeed under all conditions, but they should at least succeed under the baseline conditions that you set and you should understand (and document) the reason that they might fail.

Recall that you run tests by passing the `-t` flag, followed by the test file name, to `TerraBot.py`. Currently, you can run only one test file at a time, but if you use a test file that includes other test files, they will all be run together.

Part 4: Logging Monitor (8 points)

For this part, you will modify `logging_monitor.py` in the areas delineated by `BEGIN/END STUDENT CODE` and the file `greenhouse_agent.py`, as detailed below.

Logging data is an important aspect of understanding how and why your agent is performing (this will become more evident during the grow period and when we cover Machine Learning and explanations). In this part, you will create a **logging monitor** (a subclass of `Monitor`: see the description on page 1). Every 10 seconds, your code should log the current time, all the sensor data, and the actuator states (fan on/off, etc.) to a file. You can use the `Python` logging package or write your own logger.

Once the monitor has been implemented, edit `greenhouse_agents.py` to import `logging_monitor` and add the following line at the end of the `LayeredGreenhouseAgent`'s `__init__` function:

```
self.getExecutiveLayer().setMonitors(self.sensors, self.actuators.actuator_state,
                                     [logging_monitor.LoggingMonitor()])
```

where `self.actuators` is the variable you used to save the value of `ROSActuators()`. This adds an instance of `LoggingMonitor` with the default period of 10 seconds, which is sufficient for this problem (Note: if you have also worked on the insolation monitor, use a list of the monitors passed to `setMonitors`).

Run your agent both in the simulator (maximum speedup 100x) and on the real hardware for at least 24 hours each. Then, create graphs for each sensor (light, temperature, humidity, soil moisture, weight, water level). You may choose to have a different plot for simulated data versus hardware or you can plot them on the same graph. On each graph, also plot the times when the actuators turn on and off. Make sure to label the axes of the graphs and the lines so we can read them.

Finally, analyze the graphs in two ways. First, understand how the actuators impact the state of the TerraBot. For example, when you plot temperature, you'll notice that the temperature should go up when the lights are on and down when the fans are on. What happens when both are on? Second, compare the simulated and real data. What is similar? What is different? Do you need to update any of the limits or thresholds in

your behaviors to reflect your findings? Make those changes as necessary to the version of your code on the TerraBot hardware (not the simulator version)!

Create a document (preferably submit a PDF) with your graphs and your analyses.

Note that the part where you test on the real hardware is very important preparation for the first grow period (which starts 9/29!). Although we want to see 24 hours of data, you should test on the hardware for significantly longer than that to avoid surprises when deployment happens. As discussed in lecture, try to test at different times of the day and under different conditions (e.g., humid, hot, etc.).

Note: In the past, some groups have opted to run logging as a separate ROS node, so that it can be run remotely and log data to a local machine. You are free to do so, if you prefer. In such a case, make sure that you can (easily) specify whether the TerraBot is running in the simulator or the actual hardware. See `greenhouse_agent.py` for an example of how to specify this on the command line (recall that you run the agent with the simulator using the `-m sim` flag).

Part 5: Acquiring Images (8 points)

For this part, you will modify `camera_behavior.py` in the areas delineated by `BEGIN/END STUDENT CODE` and the file `greenhouse_agent.py`, as detailed below.

As mentioned, the TerraBot has a color camera that enables your agent to acquire images and process them. In this part, you will write a behavior to acquire and save images.

Your agent requests an image by calling `actuators.doActions` with the {`"camera": path_name`} tuple as its last argument. `path_name` is a string consisting of the directory and file name where the image should be stored.

The camera behavior to implement is as follows:

1. For consistency, adjust the light level to some reasonable value – full on is typically too bright for taking good pictures. A light level of around 400-600 is good for image collection (**hint**: look at how the `Light` behavior is implemented). Note that the TerraBot tries to adjust the shutter speed based on the light level. Since it takes a bit of time for the light level to be registered, wait a few seconds after finishing light adjustment before requesting an image. You can use the `stream-video` program that you experimented with in class to see about how long it takes your camera to adjust the shutter (each camera is a bit different).
2. Once your agent requests an image, you need to wait until the file shows up. This is because it takes some time for the TerraBot to acquire the image and write it to disk. Wait for at least 10 seconds and then use `path.exists(pathname)` to check whether the file has shown up. **Note that the image appears much faster in the simulator than on the TerraBot hardware. You should check to make sure that you are waiting long enough – if not, the image won't be read in properly.**
3. If the file does not show up, wait for 20 seconds and try again, but if you've tried unsuccessfully three times in a row, give up and print a warning message.
4. Acquire at most one image each time the behavior is invoked.
5. Acquire at most three images each day.

Start by drawing the FSM, labeling the states, conditions, and before/after functions. Then, use what you learned in Assignment 2 to create an FSM that implements the above behavior.

Once the behavior has been implemented, edit `greenhouse_agent.py` to import `camera_behavior` and add `camera_behavior.TakeImage(self)` to the list of behaviors in `LayeredGreenhouseAgent`.

Some things to note:

- If the path name is relative, it is relative to where `TerraBot.py` is run from, **not** where your agent is. You are probably best off using an absolute path name.
- **Make sure that the directory you are writing to has been created already** – the command will fail if the directory does not exist!

- Make sure your image file names are different from one another, to avoid overwriting them (a good strategy is to include the `unix_time` in the names).
- The `pathname` should not contain spaces or special characters, other than dashes or underlines.
- Test that your behavior works both using the simulator and the hardware. In the past, a common source of failure during the first grow period is not successfully collecting images.
- In the past, some teams have found that the adaptive shutter speed did not work for them when tested on the TerraBot hardware. If you want to use a fixed shutter speed, invoke `TerraBot.py` with the `-f` option.

This behavior is hard to test using tester files, since there are no sensor readings that get set. We have, however, provided you with `take_image.tst` that tests some of the parts of the behavior. In particular, it tests whether the light is within range (400-600), that the image file is successfully written to disk, and that exactly 3 images get taken each day. You can test whether it handles failure conditions correctly by requesting an image to be written to a non-existent directory. (**hint**: use baseline files to set up the correct conditions for testing, without having to wait for the appropriate times to occur).

To test the behavior, run `python TerraBot.py -m sim -s <speedup> -t <dir>/take_image.tst` in one terminal and `python greenhouse_agent.py -L -m sim` in another terminal. Note that the camera behavior has a number of steps that happen every few seconds – although it will take longer to test, we don't recommend speedup of more than 250. However, if things fail unexpectedly, you can try using a slower speedup. You can also speed up testing by (temporarily) removing all lines in `greenhouse_schedule.txt`, except for the `LightBehavior` and `TakeImageBehavior` entries (since the simulator slows down when the fans or pump are turned on).

Make sure you run for several simulated days – a common bug is to forget to reset the image counter and after 3 images the first day it never takes another image.

Part 6: Email Behavior (8 points)

For this part, you will modify `email_behavior.py` in the areas delineated by `BEGIN/END STUDENT CODE` and the file `greenhouse_agent.py`, as detailed below.

In the past, we have found it very useful to see what the TerraBot is doing every day. To this end, you will create an **email behavior** that, once a day, finds the latest image (collected using the camera behavior in Part 5) and collects the latest sensor readings and email your group, the professor, and the TAs with:

- A header, which includes your group name and the date
- The latest sensor data from each sensor
- The latest image, as an attachment

You may choose to create the email using HTML so that you can format the information nicely, but you may also just email us a plain text file (formatted in some way). The file `TerraBot/lib/send_email.py` provides some starter code for you, that you can use as you wish (or modify for your own copy). This website also has a nice explanation about sending emails via Python and how to add attachments: https://realpython.com/python-send-email.

Once the behavior has been implemented, edit `greenhouse_agent.py` to import `email_behavior` and add `email_behavior.Email(self)` to the list of behaviors in `LayeredGreenhouseAgent`.

Notes:

- Each team has their own email account of the form `terrabot<n>@outlook.com` (except for TerraBot6, whose address is `terrabot.6@outlook.com`). To send email, you need a password and a cache token. We will send each team their own information. You need to use the password in the `send` function, defined in `lib/send_email.py` place the token file into `TerraBot/param/token_cache.json`.
- `greenhouse_schedule.txt` already includes a time to enable the email behavior. If you want it to run at a different time of day, feel free to change the schedule.

- The deployed version of the email behavior depends on Part 5. If you are working on this part before the camera behavior is available, use the `interactive_agent` to take an image manually, and use that to test your code. But, once the camera behavior is in place, make sure you are getting the latest image from the correct directory.
- As you test this behavior, please do not email the professor or the TAs! Add us to the recipient list only for deployment (but, don't forget to add us in before deployment!). We suggest that you have a flag in your behavior that indicates whether to include the instructors – set it appropriately, depending on if you are testing or deploying.
- You can, however, send us a test email if you want to verify that it contains adequate information, in a reasonable format.

Part 7: Execution Monitoring Scenario (6 points)

This part will give you more practice on the types of case-study problems that the midterm and final exam cover. For the following scenario, the team should work together to answer the question in accordance with the rubric below. Submit the response in a file named `scenario.txt`.

Scenario: Suppose we place an additional temperature sensor in your greenhouse. At noon one day, this additional sensor starts reading a very high value over the allowable temperature limit on the TerraBot.

- Describe two tests that your agent (with additional software) could perform to understand what was going on; provide sufficient detail about what that test entails (what sensor data is used, what computation, etc.)
- For each test, describe what the test indicates is wrong and how to potentially recover from it (if possible).

Rubric:

- (4 pts) Provide 2 details for each of the 2 tests, according to the scenario (1pt/detail)
- (1 pt) Choose one of the tests and provide 2 reasons why it is preferred
- (1 pt) Provide the pro and con of the chosen test, relative to the other test

You may assume that you can use any of the other sensors on the TerraBot, including the camera and microphone, but that you cannot add any other sensors.

Part 8: Preparing for Deployment (0 points)

The first grow period starts on Sunday, September 29! Your agent is expected to run autonomously for at least two weeks, without human intervention. To prepare for that, you need to do the following:

- Integrate your various code implementations, especially `greenhouse_behaviors.py`. You can decide to use one person's solution, or integrate the team's solutions, but it is important that you have just one agent that is debugged and ready to run on September 29 – We will be planting then, regardless of your agent's availability!!
- Make sure your deployed agent includes the `camera_behavior` and the `email_behavior`. You may also incorporate the `logging_monitor` and `light_monitor`. In that past, students have found the former extremely useful in debugging their system as it runs; the `light_monitor` will be needed for Grow Period B, but incorporating it now can help in debugging it for the future.
- If your analysis of the sensor values in Part 4 indicates that the limits provided are not, in fact, optimal, you are welcome to change those thresholds to what you think are better values.

Submission

Submit the following files to Canvas: `adder.py, camera_behavior.py, diagnosis.py, email_behavior.py, greenhouse_agent.py, greenhouse_behaviors.py, light_monitor.py, logging_monitor.py`, your test (`.tst`) and baseline (`.bsl`) files (with embedded comments), and `scenario.txt`.

When added to a directory with the rest of the agent files, your code should run without error.

Note: several parts modify `greenhouse_agent.py`; make sure that all the changes are integrated together in the file that gets submitted!