Homework 5: Execution Monitoring (Group) Due: Nov. 2, 11:59PM

Learning Objectives

- Monitor for optimal light exposure
- Perform state estimation and outlier detection
- Perform symbolic, logic-based diagnosis

This is a group assignment consisting of three programming parts, plus a scenario problem. You should decide with your teammates how to divide the work on the first three parts, but everyone **is to work together** on Part 4 (monitoring design scenario).

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.

Download and unzip the assignment code from Canvas. Place the folder in `~/Desktop/TerraBot/agents/` and link (or copy) the files `layers.py, behavior.py` and `ros_hardware.py` from your `ROS_HW` directory, `greenhouse_behaviors.py` and `ping_behaviors.py` from your `FSM_HW` directory and `camera_behavior.py` from your `CV_HW` directory. **Copy** your most recent versions of `greenhouse_agents.py` and `greenhouse_behaviors.py`, as you will be needing to edit those files.

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.

Part 1: Monitoring Insolation (25 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 (`autograder_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 will 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 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. 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.

Finally, you need to modify the `__init__` function in `greenhouse_agents.py`. In particular, import `light_monitor` and add the following line at the end of the function:

```
self.executiveLayer.setMonitors(self.sensors, [light_monitor.LightMonitor()])
```

where you replace `self.sensors` with the variable you use to save the value of `ros_hardware.ROSSensors()` in your `__init__` function. This adds an instance of `LightMonitor` with the default period of 5 seconds, which is sufficient for this problem.

To test, run `TerraBot.py` with the `-t insolation.tst` argument. You will pass the test if the agent is within 3% of the target value.

Part 2: State Estimation (30 points total)

For this part, you will modify `kalman.py` and `humidity_estimator.py` in the areas delineated by `BEGIN/END STUDENT CODE`. You will also need to modify `greenhouse_agent.py` and `greenhouse_behaviors.py`.

As many of you have seen first-hand with your own TerraBots, sensors are noisy and may get stuck. If you trust the sensor values, your TerraBot may end up doing bad things, such as overwatering or drying out the plants. State estimation is a way of combining sensor information with models of how the system should be working to get a better idea of the true state of the world. In this part, you will develop a *Kalman filter* for estimating humidity and use it in the behaviors to raise and lower humidity, thereby achieving good performance despite noisy or stuck/broken sensors.

Step 1: Implementing Kalman Filter (10 points)

As you learned in class, a Kalman filter consists of a *predict* step, which uses a model of the system dynamics, and an *update* step, which uses sensor observations. In this part, you will use the sensor models you developed in Assignment 4 for the prediction step and the two humidity sensors for the update step.

Refer to slide 16 in the 10-3 lecture to fill in the `predict`, `update` and `estimate` functions of the `KalmanFilter` class in `kalman.py`. Several things to note:

- The standard predict step for a Kalman filter multiplies the previous mean value by a constant a_t (or a matrix if the state is multi-dimensional) to get the new value. However, in this case, the sensor model you learned gives the new value m_t, directly. So, rather than using a_t to compute m_t, you can compute a_t from your prediction function and $m_t - 1$.
- For this assignment, you can assume $c_t = 1$ and $b_t = 0$.
- For this part, you can assume that the process variance is 1 and the observation variance is 4.
- The `estimate` function takes the current time, the state at time $t - 1$ (used to predict the state at time t), and the values of the two humidity sensors at time t. The state is a python dict of sensor, actuator, and behavior values (you can print it out to see all the fields). The `estimate` function should

use those inputs to update the estimated humidity of the Kalman filter. If you don't want to use your own prediction model, you can use the linear regressor that we supply (see the `regression` function).

To test this part, run `python autograder.py -p2 -s1`. You will be graded by how close your estimate comes to the reference solution. **Note:** the reference solution provided may not be optimal – if you can do better than that, it's a good thing.

Step 2: Outlier Rejection (5 points)

Outliers are data points that are far from the mean, and often represent bad sensor readings. It is often beneficial to reject outliers, that is, not incorporate data into the Kalman filter, if they are thought to be outliers. There are several ways to detect outliers, but for low-dimensional spaces (such as here), removing data points that are a few (e.g., 2) standard deviations away from the mean is typically sufficient.

In this part, you will modify the `estimate` function to reject outliers. The `estimate` function takes an optional `outlier_rejection` argument. If `outlier_rejection` is `True`, then your code should reject a sensor reading if it is considered an outlier.

Run `python autograder.py -p2 -s2` to check this part. You will be graded by how close your estimate comes to the reference solution. You can also run `python autograder.py -p1` to check both of the first two subparts. **Note:** the reference solution provided may not be optimal – if you can do better than that, it's a good thing.

Step 3: Calibrating the Models (5 points)

The Kalman filter needs a model of the process noise and the sensor noises. In the first two steps, we assumed the noise was constant and provided a reasonable value to use. In this part, you will use the simulator to model the sensor noise. Because the humidity value changes rather slowly, you can collect a minute or two of data under different conditions and compute the variance. To do this, you need to include the argument `-i humidity.inf` when you run `TerraBot.py`. You can model the variance as either a constant or a function of state features, whichever you feel is most appropriate. Include the file `kalman.txt` in your submission that describes how you calibrated the model and what form of model you used.

Step 4: Using Estimated State (10 points)

For this step, you are to integrate the state estimator into your TerraBot agent. While it could be implemented as a `Behavior`, it is more natural to implement it as a `Monitor`, since it is doing the same thing all the time and doesn't have to transition to other states.

You will need to:

- Fill out the `activate, perceive`, and `monitor` functions in `humidity_estimator.py`. In particular, the monitor function should run the Kalman filter to estimate the humidity of the greenhouse and add the estimate to the `sensordata` dictionary. You can use `'humidity_raw'` to get both humidity sensor values (note that `'humidity'` gives the *average* of the two values).
- Modify your `LowerHumid` behavior in `greenhouse_behaviors.py` to use this estimate, if it is available, rather than the raw humidity sensor data. Note that if the monitor is not running, the behavior should fall back to using the raw sensor data.
- Modify `greenhouse_agent.py` to add the `HumidityEstimator` as a monitor. The monitor period should be at least 10 seconds, and no longer than a minute.

For this step, you may use your own sensor model that you developed for assignment 4 or the linear regressor that we supplied. If you use the latter, the features (in order) are `humid, deltaTime, led, pump, fan`). Note that for doing prediction, you should use the mean value of the Kalman filter for the `humid` feature, rather than the current sensor value. Also, you should keep track of the previous state and use that to predict the current state, rather than using current sensor and actuator values. Also note that `deltaTime` is the

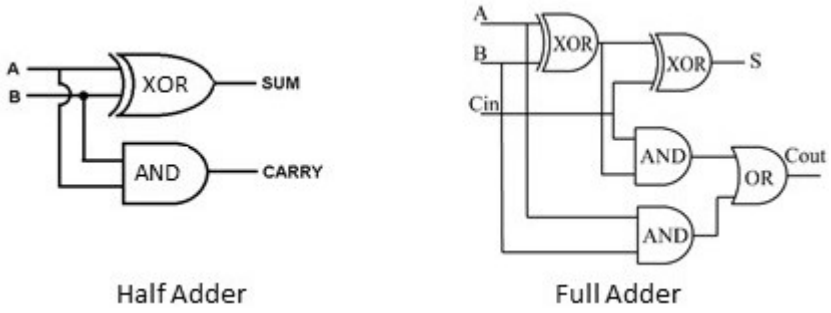

Figure 1: One-Bit Half and Full Adder

difference between the last time the monitor was run and the current time (the `self.dt` parameter of the monitor – see `doMonitor` in `monitory.py` if you want to know how it is set).

To test the ability of your estimator to handle noisy and broken/stuck humidity sensors, run `TerraBot.py` with the `-t humidity_estimator.tst` argument. To give you an idea of how you should be doing, when we run without using the monitor, the refsol gets 33 test failures (saving the output from `TerraBot.py` and grepping for "FAIL"), while using the monitor produces only 10 failures. Note, your results may vary, especially since there is sensor noise, but you should be getting similar results. **Note that the refsol is tested with a speedup of 200 - faster speedup may decrease performance.**

Part 3: Diagnosis (35 points total)

For this part, you will modify `adder.py` and `diagnosis.py` in the areas delineated by `BEGIN/END STUDENT CODE`.

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.

We suggest 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.

Step 1: Adder Circuit (5 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

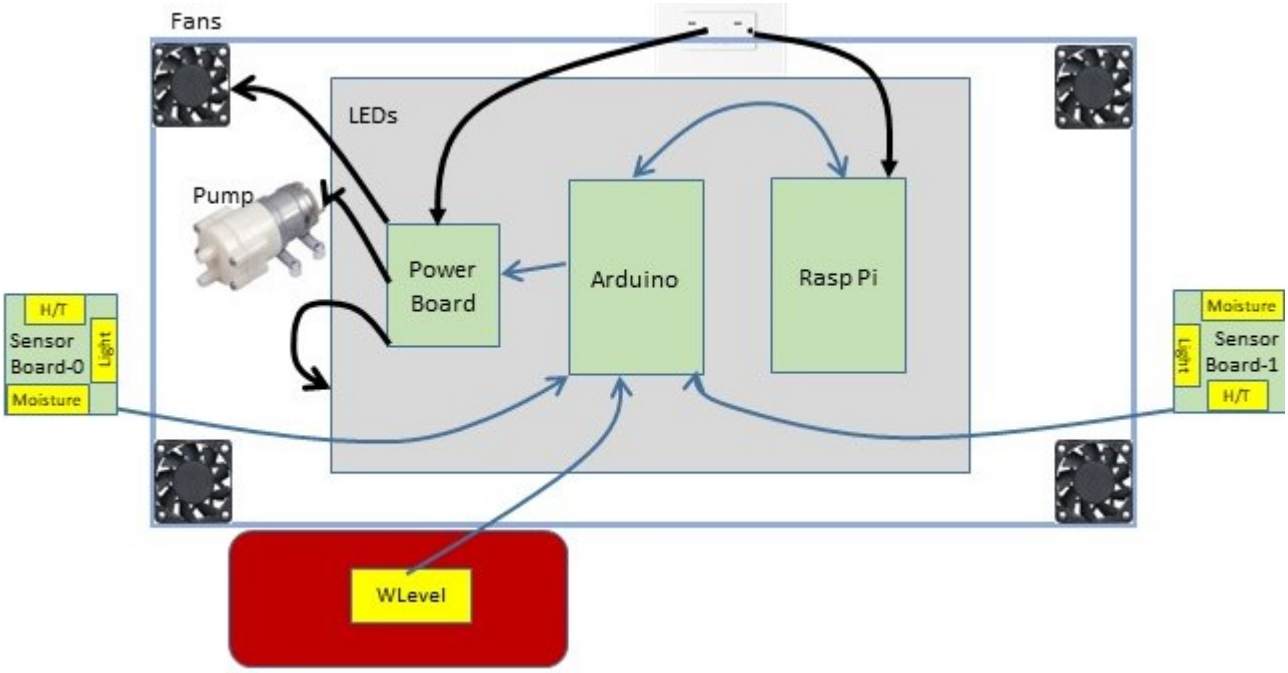

Figure 2: The Greenhouse Hardware

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.

You can test this step by running `python autograder.py -p3 -s1`.

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 (5 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`
- **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 should 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 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 sensor generates a signal (7 total), 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')`).

You can test this step by running `python autograder.py -p3 -s2`
Note that the autograder will tell you if you have missing or extra relations. It's a good idea not to have any extra relations, but you should focus on adding any missing relations, based on the description above.

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 `powered` relation, you should have a constraint that represents the condition in which it is powered. Specifically, the `Outlet` is always powered, 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`).

- 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 **sensor**, add a constraint that a sensor reading is generated (e.g., `signal(Light1, Light1)`) iff the sensor is `working`.
- 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 the `working` relations in `create_powered_constraints`.

Run `python autograder.py -p3 -s3` to test whether the constraints are all defined correctly. This is a 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 (3 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 -p3 -s4` to test whether you complete model is defined correctly.

Step 5: Diagnosis (10 points)

Steps 2-4 tested 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, finds the set of `working` and `connected` relations in each solution that are `False`, and returns a list of possible diagnoses (make sure each diagnosis is a python set). The list should be ordered in the 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. Also, 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 -p3 -s5` to test whether the diagnosis works correctly.

Part 4: Execution Monitoring Scenario (10 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 titled `scenario.txt`.

Rubric:

- (6 pts) Provide 2 details for each of 3 tests, according to the scenario (1pt/detail)
- (2 pts) Choose one of the tests and provide 2 reasons why it is preferred
- (2 pts) For the other 2 tests, provide their pros and cons relative to the chosen test

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 three tests that you 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 at all).

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.

Submission

Submit the following files to Canvas: `adder.py`, `diagnosis.py`, `greenhouse_agent.py`, `greenhouse_behaviors.py`, `humidity_estimator.py`, `kalman.py`, `kalman.txt`, `light_monitor.py`, `README`, and `scenario.txt`.

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