APS106 – Lab #4

Preamble

In this week's lab you will practice using loops and conditional statements to build a system for designed to monitor the value of a sensor over time. *Use appropriate variable names and place comments throughout your program.*

The name of the source file must be "lab4.py".

Deliverables

1. A Python file named lab4.py with your function.

Five test cases are provided on MarkUs to help you prepare your solution. **Passing all these test cases does not guarantee your code is correct.** You will need to develop your own test cases to verify your solution works correctly (see below for more details). Your programs will be graded using ten secret test cases. These test cases will be released after the assignment deadline.

IMPORTANT:

- Do not change the file name or function names
- Do not use input() inside your program

Your task this week will be broken down into two parts. In the first part you will create a function to check whether a sensor measurement is above or below a particular threshold value. In the second part of this lab, you will write functions that will be designed to simulate tests of your sensor system to check whether your thresholding function works correctly. By completing both parts of this assignment, you will get a taste of the design process used by engineers who develop software for interface with physical systems and sensors.

Part 1 – Hysteresis Threshold

For this assignment, we will imagine that we are building a system to monitor the temperature inside a furnace used in chemical industrial processes like cement production. While your colleagues design the electronics and mechanical components for the sensor and heater circuits, you have been asked to write the software that will read measurements from the temperature sensor and determine when to turn the heater on or off. Your team wishes to use the following simple piecewise function to describe the state of the heater as a function of the temperature, T:

$$state(T) = \begin{cases} on & T < T_{desired} \\ off & T \geq T_{desired} \end{cases}$$

that is, when the current temperature is less than the desired temperate, $T_{desired}$, the heater should be on, otherwise the heater should be off.

Your task for this part of the lab will be to write a function, `heat_control_hysteresis_thresh`, that determines whether the heater should be on or off based on the current temperature measurement from the sensor. Before you start, however, your colleagues inform you that the sensor readings are "noisy" (see figure 1) and using the simple piecewise threshold function will result in the heater excessively switching between on and off states.

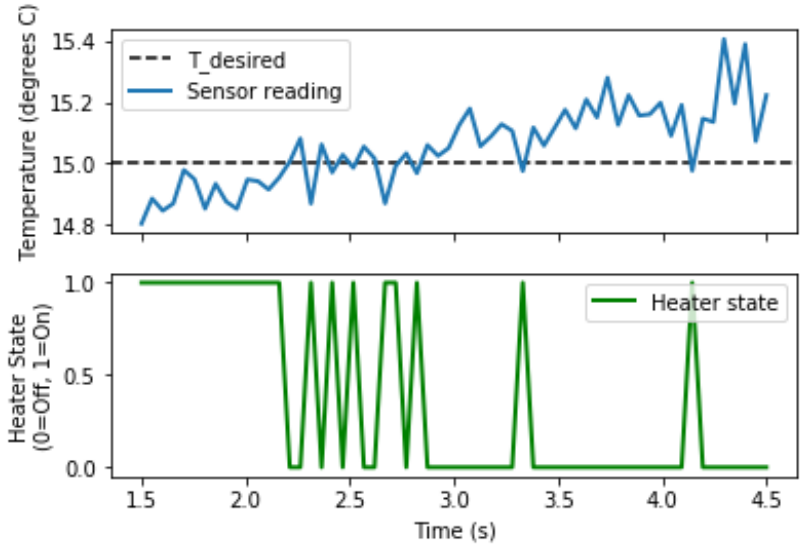

Figure 1. Noise from the sensor readings results in excessive switching of the heater

To avoid this excessive switching, you decide to implement a hysteresis thresholding function:

$$next\_state(T, current\_state) = \begin{cases} on & T < T_{desired} - \alpha \\ off & T > T_{desired} + \alpha \\ current\_state & |T - T_{desired}| \leq \alpha \end{cases}$$

where T is the current temperature reading from the sensor, $T_{desired}$ is the desired temperate, $current\_state$ is the current state of the heater (on or off), $next\_state$ is the next state of the heater (on or off), and α is a constant. Now our function has two thresholds at $T_{desired} \pm \alpha$ and if $|T - T_{desired}| \leq \alpha$, then the heater should remain in the same state (e.g., if the heater is on, it should remain on). Figure 2 shows the behaviour of the heater using this control scheme.

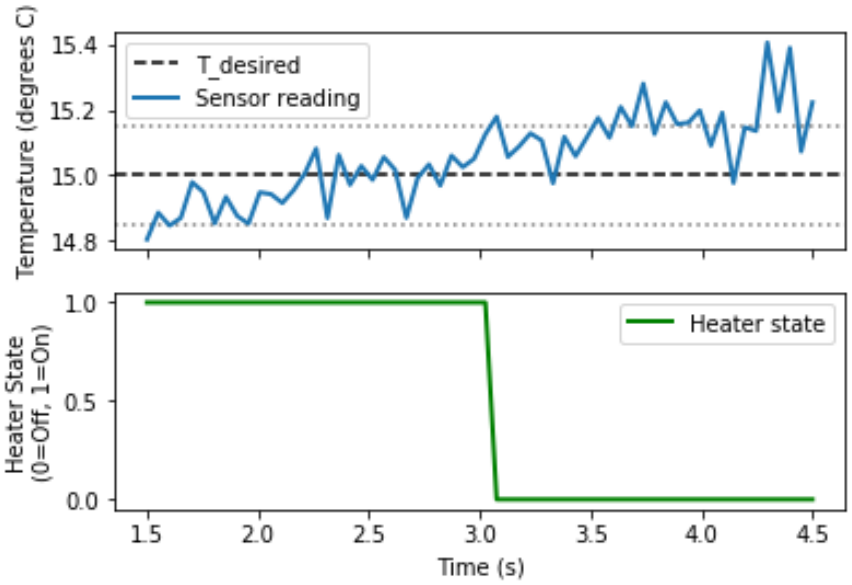

Figure 2. Implementing the hysteresis threshold prevents excessive switching. The grey dotted in the top plot lines represent $T_{desired} \pm \alpha$.

You will now write the function, `heat_control_hysteresis_thresh`, to implement this hysteresis threshold. The function accepts four parameters:

- `t_measured` – a float representing the latest temperature measurement from the sensor
- `current_state` – a boolean representing the current state of the heater (True = on)
- `t_desired` – a float representing the desired temperature
- `alpha` – a float representing the hysteresis threshold range

The function should return a boolean representing the next state (on or off) of the heater.

Exercise: Assuming constant values of $T_{desired}$ and α, this function can be exhaustively tested using 12 unique test cases (i.e., different input combinations). As an exercise define a set of 12 test cases with constant $T_{desired}$ and α values that could be used to test every condition in the hysteresis function equation. Hint: look at the conditional (if) statements and boolean expressions in your function. How could you design inputs to make each one of those False/True? Hint #2: consider the regions defined by the horizontal dashed lines in the top plot of figure 2.

Part 2 – Counting the number of heater state changes

Your engineering team suspects that the lifetime of the furnace heater system will be impacted by the frequency of switching between its on and off states. To be able to gather some data and investigate whether this is true, your team asks you to write another function to track the number of times the heater changes state during use.

As you work through your algorithm and programming plans, you begin to ask yourself an important question: ***how will I test this function?*** The sensor circuit will not be ready for testing for a few weeks. Even when it is ready, you want to be able to test your code without having to depend on whether the sensor is operating correctly. Drawing on your testing and debugging skills from APS106, you decide you will write another function that will simulate the sensor readings for your tests. With direct control over the sensor readings, you will be able to know exactly how many threshold crossings there are and therefore be able to quickly and accurately assess whether your function works.

To simulate the sensor measurements, you decide to use the following mathematical function:

$$s(t) = c_0 t + c_1\sqrt{|t|} + c_2 \sin(t) + c_3 \cos(8t) + c_4$$

where t is the time, $|t|$ is the absolute value of the time, s(t) is the sensor measurement at time t, and $c_{0\text{-}4}$ are constants.

Part 2.1 – Square root approximation using the Newton-Raphson method

For this lab, we will not use the square root function from the math module. Instead, you will write your own function the implement the Newton-Raphson method to approximate the square root of a number. Briefly, this method uses the derivatives of a function to find successively better approximations of roots for a function. If you're interested, you can optionally read more about the method here, but everything you need to know about the algorithm for your code is outlined below.

The Newton-Raphson method iteratively converges on the root of a number n using the following series:

$$x_{i+1} = \frac{1}{2}\left(x_i + \frac{n}{x_i}\right)$$

where x_i is the approximation of the root after the i^{th} iteration and n is the number whose root we are trying to approximate. The initial estimate of the root x_0 is usually set to n. Each x_{i+1} is closer to the true value of $\sqrt{n}$ than x_i, or stated mathematically:

$$\left|\sqrt{n} - x_{i+1}\right| < \left|\sqrt{n} - x_i\right|.$$

The iteration stops once the approximation is within a given numerical tolerance, ϵ, of the true root:

$$\left|x_i^2 - n\right| < \epsilon.$$

Below is a worked example where we approximate the root for $n = 4$ and $\epsilon = 0.1$:

Iteration 0

We set $x_0 = n = 4$. Since $|x_0^2 - n| = |4^2 - 4| > 0.1$, we continue to the next iteration.

Iteration 1

$x_1 = \frac{1}{2}\left(x_0 + \frac{n}{x_0}\right) = \frac{1}{2}\left(4 + \frac{4}{4}\right) = 2.5$. Again, $|x_1^2 - n| = |2.5^2 - 4| > 0.1$, so we continue to the next iteration.

Iteration 2

$x_2 = \frac{1}{2}\left(x_1 + \frac{n}{x_1}\right) = \frac{1}{2}\left(2.5 + \frac{4}{2.5}\right) = 2.05$. Again, $|x_2^2 - n| = |2.05^2 - 4| > 0.1$, so we continue to the next iteration.

Iteration 3

$x_3 = \frac{1}{2}\left(x_2 + \frac{n}{x_2}\right) = \frac{1}{2}\left(2.05 + \frac{4}{2.05}\right) \cong 2.00061$. Now, $|x_3^2 - n| = |2.00061^2 - 4| \cong 0.0024 < 0.1$, so our current approximation x_3 is within our tolerance, we stop iterating and x_3 would be returned.

For this part of the lab, you will write the `newton_raphson_sqrt` function. This function has two input parameters:

- `n` – a positive float whose root will be approximated
- `epsilon` – a positive float that sets the error tolerance of the approximation

The function should return the approximated square root of n using the Newton-Raphson method. The returned value should be rounded to 3 decimal places. You may assume both inputs will be non-negative.

Warning: you can find implementations of the Newton-Raphson method online, however, many of these will use different stopping conditions. Make sure to follow the algorithm outlined above.

Part 2.2 – Counting threshold crossings using simulated sensor data

In the final part of this lab, you will write a function to count the number of times the state of the heater changes (i.e., the number of times the temperature crosses our hysteresis threshold) during a given time window.

The function `get_sensor_measurement` is provided to you to use during this part of the lab. You should read the docstring and code to familiarize yourself with the function and how it works. You should not modify this function. Note that this function calls the `newton_raphson_sqrt` function from part 2.1.

Your task is to complete the `thresh_crossing_counter` function. This function has 9 input parameters:

- `temp_desired` – a float representing the desired temperature
- `hyst_alpha` – a float representing the buffer range for the hysteresis threshold
- `t_start` - a float representing the initial time in seconds used to generate the simulated sensor readings
- `t_end` – a float representing the time in seconds where the simulation should stop
- `c0, c1, c2, c3, c4` – the constants used in the equation to generate simulated sensor measurements

Your function should be designed to "read" the sensor every 0.05 seconds. Your function should return the number of times the state of the heater changes between `t_start` and `t_end`. You may assume that $0 \leq t_{start} \leq t_{end}$.

To help you test your function, you may find using an online graphing tool like desmos (https://www.desmos.com/calculator) helpful to create plots of the sensor simulation function with different values of $c_{0\text{-}4}$. You can then count the number of crossings your function should detect.

Part 2.3 – Closing remarks: Welcome to the wonderful world of test stubs!

Some of you may be wondering what completing the last function achieved since it does not "read" real sensor data. If you inspect the code, however, you will notice that if we wanted the code to work with a real sensor, we would only need to change the `get_sensor_measurement` function to read the sensor rather than generate a simulated value. The rest of the code would remain the same! Now, as soon as you are ready to test with the real sensor, you will have confidence that your code works correctly and you can debug any problems that arise more efficiently. This practice of creating functions (often called stubs) to simulate or emulate how other parts of your code will operate enables engineers and programmers to incrementally build and test code without needing the whole project to be complete, which makes debugging and testing much more manageable and efficient. Powerful stuff!