Measurement System Automation:

Software Testing Plan

11/7/25

Version 1

Team Name: Thermo-Gen

Sponsor: Steve Miller

Mentor: Jeevana Swaroop Kalapala

Team Members: Olivia Vester, Kameron Napier, Gareth Carew

THERMO-GEN

Table of Contents

1. Introduction

In creating the ThermoGen software application, we are motivated by our client's need to automate their thermo-diffusivity testing apparatus. With our client's current testing set-up, they have to manually adjust the components to create the environmental setting that matches the specifications to test their materials. This method of operating is both time consuming and imprecise. With our software application, the client's testing operator will be able to set the pressure and temperature parameters for the environment without having to use manual levers and dials to adjust those parameters. The ultimate goal for the application is to reduce the amount of time the testing operator has to spend preparing the test environment while also not greatly impacting the nature of the test by not taking up a lot of room or being overly complex for the user to interact with.

In testing the ThermoGen software, we are also greatly motivated by our client's needs for saving time and increasing environmental precision. We must test our application thoroughly to make sure it can handle adjustments for our client's average test environments to their most extreme temperature and pressure settings.

Due to the nature of our project, we will not be able to test our software application in a conventional way. Our client's current hardware for testing their materials does not have the capability of being integrated with an electronic system like our application; there are some updated cables and ports that are necessary for integration to be done and once the client has these upgraded system parts installed, our system can just simply be plugged in and used. Since we cannot directly test our system as something fully integrated we will be testing using a simulation we have created for both pressure and temperature.

We will break our testing down into three major parts: unit testing, integration testing, and usability testing. For unit testing, we will test our software to take in data points for our client's most extreme needs. We will be adjusting the system as needed to make sure those requirements can be met. Next, for integration testing, we will not be using a conventional method since we are not able to integrate at this time. Using the simulation we have created that replicates our client's environment, we will be able to see if this application will work. For temperature, there are some parts of the client's current system that do interact with our application and we use that as a basis to tell if our simulation is working appropriately and can eventually just be used in a "plug-and-play" fashion by our client. Lastly, we have usability testing, and here is where we will give our client the system and have them use it as it is right now with little direction to test how intuitive our system is. Based on the client's feedback from all of these tests, we will be making our final adjustments.

Next, we will go into each of these three major testing sections in more detail.

2. Unit Testing

Unit testing is a fundamental software engineering practice aimed at confirming that small blocks of code, typically functions, work correctly in isolation from the rest of the system. The goal of unit testing is to ensure that each unit performs as expected under a variety of conditions, including normal, boundary, and exceptional cases. By identifying issues, unit testing contributes to higher software quality, improved maintainability, and reduced debugging time.

The primary goal of unit testing are as follows:

- Verification of correctness: Validate that each code “unit” produces expected outputs for given inputs.
- Isolation of errors: Identify the location of faults within a system by testing units independently.
- Regression prevention: Detects new defects introduced by code changes through automated re-execution of tests.
- Documentation of behavior: Provide specifications that describe how the code should function.
- Support for refactoring and integration: Enable safe modification and integration by ensuring that existing behavior is preserved.

Python has a number of features to support unit testing including a built-in package called unittest, and many other 3rd-party packages such as pytest and nose2. Unittest is the built-in testing framework that follows the standard unit testing architecture. It supports test case classes, setup and teardown routines, and command-line test discovery. Python’s unittest.mock module allows you to replace components with mock objects, enabling isolated testing of dependencies such as APIs, databases, or file systems.

Some common 3rd-party unit testing packages are pytest and nose2. Both are widely used packages that also follow the standard unit testing architecture and include setup and teardown routines, and command-line functionality. They both also automatically detect test files and directories based on naming conventions.

We have five files that have methods we will want to create unit tests for, notifications.py, pressure_read.py, pressure_write.py, temperature_read.py, and temperature_write.py. In total there are eight functions we need to create unit tests for.

2.1 Functions

read_pressure(self)

- Equivalence Partitions
  - 0 - 760 torr
- Boundary values
  - Min value should be 0 torr

    - Max value should be 760 torr
- Planned tests
    - Test for a normal value (between 0 and 760)
    - Test for boundary values (0 and 760)
    - Test for out of bounds values (-1 and 1000)

_read_gauge_line(self)

- Equivalence Partitions
    - Valid response from gauge
    - Empty or invalid response from gauge
    - Communication error or disconnected gauge
- Boundary values
    - Minimum: Empty string returned
    - Maximum: Long string with valid numeric data
- Planned tests
    - Test with a valid gauge response string e.g. “123.4” expecting correct return of “123.4”
    - Test with an empty response e.g. “” expecting None
    - Test with invalid response data e.g. “abc” expecting None
    - Test with a simulated communication failure e.g. serial port closed expecting graceful handing with None returned

_parse_pressure_from_string(self, s)

- Equivalence Partitions
    - Valid numeric strings containing pressure values e.g. 123.4
    - Invalid non-numeric strings e.g. “abc”
    - Empty string or None input
- Boundary values
    - Minimum: “0” returns 0.0
    - Maximum: a really large numeric string e.g “999999.9” returns 999999.9
- Planned tests
    - Test with a valid simple numeric string e.g. “100” expecting 100.0
    - Test with a valid numeric string embedded in text e.g. “Pressure=123.45 Torr” expecting 123.45
    - Test with an invalid string e.g. “abc” expecting None
    - Test with an empty string expecting None
    - Test with None input expecting None

_send_valve_command(self)

- Equivalence Partitions

    - True/False
- Planned tests
    - Test when valve is not connected (expecting false)
    - Test when valve is connected (expecting true)

adjust_pressure(self)

- Equivalence Partitions
    - 0 - 760 torr
- Boundary values
    - Min value should be 0 torr
    - Max value should be 760 torr
- Planned tests
    - Test for a normal value (between 0 and 760)
    - Test for boundary values (0 and 760)
    - Test for out of bounds values (-1 and 1000)

temp_to_volt(self, temp)

- Equivalence Partitions
    - 25C - 1000C
- Boundary values
    - Min value should be 25C.
    - Max value should be 1000C.
- Planned tests
    - Test for a normal value (between 25 and 1000)
    - Test for boundary values (25 and 1000)
    - Test for out of bounds values (0 and 1100)

write_volt(self, volt)

- Equivalence Partitions
    - 0V - 80V
- Boundary values
    - Min value should be 0V.
    - Max value should be 80V.
- Planned tests
    - Test for a normal value (between 0 and 80)
    - Test for boundary values (0 and 80)
    - Test for out of bounds values (-1 and 100)

read_volt(self)

- Equivalence Partitions

    - 0V - 80V
- Boundary values
    - Min value should be 0V.
    - Max value should be 80V.
- Planned tests
    - Test for a normal value (between 0 and 80)
    - Test for boundary values (0 and 80)
    - Test for out of bounds values (-1 and 100)

3. Integration Testing

3.1 Overview of Integration Testing

Integration testing verifies that the individual modules and components of the system interact correctly when combined into a complete application. While unit testing focuses on ensuring that isolated functions behave as expected, integration testing ensures that the exchanging of data, control signals, and synchronization functions well. In our solution, integration testing is critical because it involves multiple subsystems working together: temperature sensing and control, pressure measurement and regulation, user interaction through a graphical interface, pressure measurement and control, user interaction through a graphical interface, and real time hardware feedback through external devices. The goal of integration testing is to confirm that each of these components communicates properly and behaves predictably when interacting under realistic operating conditions.

3.2 Approach for Integration Testing

The integration testing strategy follows a bottom up approach, combining smaller verified units into progressively larger subsystems before finally testing the entire system as a whole. This approach is well suited for our solution because it allows early detection of interface and communication issues while maintaining control over potential hardware risks. The testing begins with software only simulations, then gradually introduces hardware components once stable behavior is achieved. By validating simulated operations first the team can ensure the logic is correct before applying the tests to the physical hardware.

Because the system directly interacts with external hardware special attention is given to communication protocols and synchronization timing. The testing process emphasizes verifying that messages sent to connected serial devices are received correctly and produce consistent results. Additionally, the user interface and control threads are tested for data consistency to confirm that sensor readings and target values are accurately shared between the front end and back end processes. This

careful sequencing of tests helps ensure both safety and reliability when the system transitions from simulation to real hardware.

3.3 Subsystem Integration

The first stage of integration testing focuses on the temperature control system, which includes the TemperatureReader, TemperatureWriter, and the main control logic. The objective is to confirm that temperature data, whether simulated or collected from a real thermocouple, flows correctly through the system. The controller's output to the simulated or real power supply is then verified to ensure that it reacts appropriately to the measured temperature. Logging tools capture the relationship between input readings and output voltages, helping identify latency or scaling issues. Once stable operation is confirmed in simulation, hardware devices are connected, and readings are validated against expected thermal responses.

Next, the pressure control system is integrated and tested. This stage combines the PressureReader, PressureWriter, and the central controller. The test verifies that the software can correctly interpret data from the pressure gauge and issue proper open or closed commands to the valve. In simulation mode, the Pi Pico replaces the valve and gauge, allowing the pressure control logic to be tested safely. The Pico's onboard LED provides a visual indicator where it turns on when the simulated pressure decreases and off when pressure increases. During testing, pressure setpoints are provided through the user interface, and logs are reviewed to confirm that LED behavior and simulated pressure trends align with the system's intended control logic. When the real hardware is connected, pressure readings from the gauge are compared to target values to ensure the valve responds within tolerance and stabilizes correctly.

The graphical user interface (GUI) and notification system are integrated once the temperature and pressure subsystems operate reliably. The GUI allows users to set temperature and pressure targets, choose units, and switch between automatic and manual operation modes. Integration testing confirms that user input from the GUI propagates correctly to the controller, and that updates from the sensors are displayed in real time. Particular attention is paid to ensuring that submit buttons and dropdown selections trigger the correct backend functions, that unit conversions are accurate, and that invalid input is handled gracefully. The notification module, which delivers alerts through Windows Toast messages, is tested concurrently to verify that it activates when targets are reached, when errors occur, or when communication is lost. Logging and visual feedback on the GUI are used to confirm that the system maintains consistent state awareness across all layers.

3.4 System Integration and Verification

Once all subsystems have been individually validated, full system integration testing combines every module into a complete operational environment. This stage focuses on verifying the end-to-end data pipeline from sensors and simulated inputs to the graphical display and notification outputs. In both hardware and simulation modes, the system is run with varying target conditions to ensure consistent performance under different loads. The test procedure confirms that each component's output becomes another's valid input, with correct units, timing, and responses. Performance is evaluated based on criteria such as response latency, communication reliability, and system stability over extended operation.

Automated tools like pytest and unittest mock are used to simulate serial communication and validate timing and message formats. During GUI testing, PySide6's debugging feature helps track signals to ensure that user actions trigger the intended control logic. Serial monitors and logging utilities record all communication with hardware devices to verify accuracy and order of transmitted data. Key metrics for success include consistent data transfer, synchronization within half a second between modules, and correct activation of feedback or safety mechanisms in abnormal conditions.

Integration testing also evaluates fault tolerance by introducing simulated hardware disconnections and unexpected data. These tests confirm that the software handles errors gracefully by halting unsafe operations and issuing clear alerts to the user. Extended tests lasting several hours validate system endurance and ensure that no performance degradation, such as memory leaks or delayed responses, occurs during long term operation.

Ultimately, this integration testing ensures that the temperature, pressure, and interface modules function cohesively as one system. The combination of bottom up assembly, simulation based safety testing, and gradual hardware integration provides a structured and reliable method for confirming correct operation. By thoroughly validating communication and synchronization between modules, the project minimizes the risk of failure in real world conditions and ensures a robust, responsive, and user friendly control system.

4. Usability Testing

To go about our team's usability testing for the ThermoGen application, we will first need to meet with our client face-to-face. First, before explaining our procedure, we must go over usability testing in general. In this module of testing, we are working with the user, in this case our client, to see if they can effectively access the provided functionality. This testing module aims to examine the effectiveness, understandability,

and ease of use of the platform we have created. For the purposes of our testing, we will conduct the unit testing procedure in three parts.

First, we will conduct our initial product test with the client. At this test, we will provide initial set-up instructions for our client and let them use the system as it currently exists with little-to-no guidance beyond the initial set-up portion of the test. The purpose of this “hands-off” approach is to see how intuitive our system is on its own. During this, we will take notes on where the client gets hung-up, has questions, has concerns, etc. Following this initial usability test, we will have a team-to-client discussion about what they like, what they do not like, and what they would like to see changed in the product before the next testing session. This meeting will serve the purpose of understanding where the client’s requests and needs have been met with the application and where the application may be lacking. With the feedback gathered from the initial test, the team will spend a brief period of “refinement time” adjusting the product slightly based on the initial feedback.

In our second iteration of testing we will meet with the client once again and repeat the previous test with the refined version of our application based on the feedback from user testing session one. By this test, the product should be able to both meet the needs/requirements of the client and be easy to use. Any further critiques made by the client will be taken into consideration and will be adjusted for in the final “refinement-time” period. This test section’s main purpose is to see if the client likes how the team has adjusted the application and to see if it is easier to use. This session will be the last major test for application adjustments before we present the finalized product.

Our third section of usability testing will be based on the user manual. As part of the final product the team will provide the client with a user manual that guides them on the use of the application. In our final usability testing session we will provide the client with the user manual and allow them to guide themselves through the setup and usage of the application. The purpose of this section is to test the effectiveness of the user manual as it will be the bridge between our product and the client being able to use it. During this test we will take note of any client feedback and adjust the manual as needed. This test will be the last test for the product before passing off to the client; because of this, any notes for final adjustments will be thoroughly documented and confirmed with the client so that the final product meets their specifications and standards.

Though this testing takes place in 3 large steps, the team intends for this portion of testing to be completed within a 2 week time frame to ensure quick response times to client feedback and on-time product delivery.

5. Conclusion

In summary, the testing plan for the ThermoGen application provides a comprehensive strategy for ensuring that the software meets both functional and performance expectations before deployment. Through unit testing, individual components such as pressure and temperature readers and writers are verified to perform reliably under both normal and boundary conditions. Integration testing then ensures that these components communicate effectively, maintaining synchronization between sensors, control systems, and the graphical user interface, both in simulated and real hardware environments. Finally, usability testing bridges the gap between engineering and end user experience, confirming that the system is intuitive, efficient, and aligned with the client's operational workflow.

Together, these layers of testing form a structured approach that reduces the likelihood of system errors, improves maintainability, and validates that the software performs safely and predictably when interacting with external devices. By combining simulation based testing with real hardware verification, the plan mitigates risk during development while still ensuring accurate, real world performance once the system is fully integrated. This methodical approach provides confidence that ThermoGen will deliver a precise, time saving, and user friendly solution for automating thermal diffusivity testing, ultimately fulfilling the client's need for efficiency and environmental control.