Mass Spectrometer Interface

A Desktop Application for Reading Instrument Data

Cousins Photosynthesis Lab in the School of Biological Sciences at WSU



**Team Linnaea Borealis**

****

Kyler Kupp, Erik Holtrop

**TABLE OF CONTENTS**

[**I.**](#_gjdgxs) **Introduction 4**

[I.1.](#_30j0zll) Project Overview 4

[I.2.](#_1fob9te) Test Objectives and Schedule 4

[I.3.](#_3znysh7) Scope 4

[**II.**](#_2et92p0) **Testing Strategy 4**

[**III.**](#_tyjcwt) **Test Plans 4**

[III.1.](#_3dy6vkm) Unit Testing 4

[III.2.](#_1t3h5sf) Integration Testing 4

[III.3.](#_4d34og8) System Testing 4

[III.3.1.](#_2s8eyo1) Functional testing: 4

[III.3.2.](#_3rdcrjn) Performance testing: 4

[III.3.3.](#_26in1rg) User Acceptance Testing 5

[**IV.**](#_lnxbz9) **Environment Requirements 5**

[**V.**](#_1ksv4uv) **Glossary 5**

[**VI.**](#_44sinio) **References 5**

# Introduction

## Project Overview

The Mass Spectrometer Interface is a series of desktop applications designed to facilitate the analysis of mass spectrometry data for researchers in the Cousins Photosynthesis Lab at Washington State University (WSU). This application streamlines data processing for plant respiration studies, enabling efficient measurement of gases like CO₂ and O₂ and their isotopes. It replaces existing proprietary software that outputs extensive, often unnecessary data, focusing instead on essential data analysis functionalities that are critical for researchers. The following are the major functionalities of our modules that require(d) testing.

* **Concentration Calculations**: Module 1 computes gas ratios, such as bicarbonate to carbon dioxide, using real-time data.
* **Isotope Ratios**: Module 3 calculates and plots the portion of CO2 molecules that are 13C(18O)2​, a combination of particular isotopes of carbon and oxygen.
* **Real-time Derivative Calculation**: Module 3 calculates and plots the derivative of the isotope proportion described above in real time.
* **Data Conversion**: To support additional instruments, Module 4 converts raw data from a second mass spectrometer to the data format accepted by Modules 1-3, enabling integration with the lab’s current workflow and maintaining a standard for future projects.
* **Multi-Instrument Data Integration**: Module 5 combines data from multiple instruments, including the LI-COR Leaf-gas Exchange System, a tunable diode laser, and a Picarro device, into one unified data stream.
* **Graphing and Data Manipulation**: Modules 1-3 dynamically plots data streams, allowing users to manipulate graph scales, set plotting speeds, and isolate data segments for analysis. Movable “mean bars” let users select specific data ranges to calculate mean values, improving analysis precision.

## Test Objectives and Schedule

Our testing approach is very integrated into our development process. Developers apply unit testing as they work on a unit, then integration testing when a component satisfies unit tests. System testing is performed at standup meetings as needed.

Our project is compartmentalized into modules, which are our main deliverables. Typically our milestones involve the completion of a module, or a module’s prototype. The following table entails our expectations for production and testing, up to sprint 5.

|  |  |  |
| --- | --- | --- |
| **Milestone/Product** | **Expected Completion** | **Level of Testing** |
| Module 1 | Sprint 1 | User Acceptance |
| Module 3 | Sprint 2 | Unit |
| Module 3 | Sprint 3 | User Acceptance |
| Module 4 Prototype | Sprint 3 | Unit |
| Module 4 | Sprint 4 | Integration |
| Module 4 | Sprint 5 | User Acceptance |
| Module 5 Prototype | Sprint 5 | Unit |

## Scope

This document provides an overview of the steps we take to test different elements of our project. This overview includes the overall flow, the unit tests for individual parts, and integration/system testing for combinations of different parts. The processes outlined are very particular to the context of our project; we’re very aware of our stakeholders and the integration plan that they prefer. Ultimately this document is intended to outline what we consider the ideal methods of testing, in order to keep development in line with those practices.

The secondary intention of the document is for posterity and external communication. Stakeholders and future developers may find themselves curious about the testing process of this project, and this document is the primary resource for that rediscovery.

# Testing Strategy

The following is our loose approach to testing a particular module or feature:

1. Identify the requirement(s) involved in this module/feature. This should either come from the Requirements and Specifications Document or be added to the Requirements and Specifications Document before continuing.
2. Establish the test(s) that will be used. In other words, identify the process of using the module or feature. Document these tests in the Testing Plan Document.
3. Identify any necessary dependencies. This includes other components and input data. Include assumptions about these dependencies in the Testing Plan Document.
4. Build a representation of what acceptable results look like. This must consider our assumptions made in the previous step. For example, an Excel graph of a data acquisition: the particular data acquisition should be clarified in the previous step, with the Excel graph built off it in this step. This mockup(s) should either be included in the Testing Plan Document, or in the relevant module’s “Testing” folder with reference to it in the document.
5. Perform the test(s).
6. If the test(s) is unsuccessful, fix it if possible. If the test(s) is not successful by next standup meeting, prepare a short explanation or document explaining the issue.
7. If the test is successful, move the relevant GitHub issue to Review/QA, or from Review/QA to Done.

Ultimately, our strong connection to our primary stakeholder, Dr. Cousins, allows us to adapt our development process to a more flexible approach that handles opportunities and issues as they come up. Sometimes the requirements are vague, and the following approach may be more effective than generating more specific requirements and a mockup:

1. Implement the most obvious executions of a requirement.
2. Present those executions to the client/stakeholder(s). Receive feedback.
3. If one of the implementations is acceptable, move the relevant GitHub issue to Review/QA, or from Review/QA to Done.
4. If none of the implementations are acceptable, either return with novel implementations or revert to the primary approach, depending on team consensus.

Our delivery process is basically Continuous Delivery. Our client prefers executable files over python scripts, so a new deployment must be manually created by a team member each time. The modular nature of the project lends itself to creating a new iteration of each improved module every sprint. In this regard, our development is continuously integrated with monthly releases.

# Test Plans

This section outlines the comprehensive plan for testing the mass spectrometer interface system, detailing the strategies for unit, integration, system, and user acceptance testing. Each testing phase is designed to identify and address potential faults at different stages of development, from individual software units to the entire integrated system. The testing plan follows a systematic approach, beginning with isolated unit testing to validate the smallest components of the application, followed by integration testing to ensure smooth communication between modules. System testing will then evaluate the application’s compliance with overall requirements, focusing on functional, performance, and stress tests to confirm reliability under varying conditions. Finally, user acceptance testing will involve end-users to validate that the system meets their needs and is ready for operational deployment.

## Unit Testing

The primary objective of unit testing for this system is to validate the functionality and reliability of individual components, or “units,” by isolating them from the rest of the code and checking for bugs or unexpected behavior. Specifically, the unit tests will cover core functionalities, including data parsing from CSV files, data transformations, calculations, and graphical display setup. Unit tests will be designed for each function and method within the modules, verifying both expected outputs and error-handling mechanisms when presented with invalid data inputs. Using Python’s pytest framework, tests will be semi-automated to streamline the process and improve reliability. Mocking will be employed to simulate data inputs and dependencies where necessary, especially for modules that rely on external data sources or interactions. This approach ensures that each unit functions independently and accurately, laying a stable foundation for subsequent integration and system testing phases.

## Integration Testing

The purpose of integration testing in this system is to identify faults that may arise when individual components interact, focusing on groups of components rather than isolated units. This phase will ensure that data flows smoothly between components. For instance, components responsible for parsing CSV files and transforming data will be integrated and tested as a cohesive unit to confirm that each stage performs as expected in the broader workflow. To manage dependencies, a test data set simulating real-world CSV inputs will be used to validate functionality and data consistency across components. Python’s pytest framework will be used for semi-automated integration tests, while pytest-mock will aid in simulating dependencies, ensuring that testing conditions are controlled and predictable.

## System Testing

System testing will be conducted to ensure that the mass spectrometer interface system operates as a cohesive unit, meeting all specified requirements. This phase will involve executing a series of planned tests to validate both functional and non-functional aspects of the system as a whole.

## Functional testing:

In functional testing, we will develop a comprehensive set of test cases based on the functional requirements outlined in the project documentation. Each functional requirement will correspond to at least one test case. Each standalone system will be tested in realistic scenarios to ensure that they meet user expectations. Test cases will be prioritized to focus on critical user paths and high-risk areas, ensuring that the most relevant features are validated first. Any failures or discrepancies found during testing will be documented and addressed promptly to enhance system reliability.

## Performance testing:

To assess the system's performance, we will conduct performance testing that focuses on response times, resource utilization, and overall system stability. This will include stress testing the systems by simulating high-load conditions using large datasets to determine how the system performs under pressure. For example, we will measure the speed at which the system can plot all data points from a large sample at once. Key metrics such as processing speed, memory usage, and data handling capacity will be monitored. If any performance issues arise, they will be investigated and resolved to ensure the system meets the expected performance benchmarks.

## User Acceptance Testing:

User Acceptance Testing will involve lab researchers in evaluating the system based on their operational needs. We will organize testing sessions where lab researchers will perform key tasks, such as loading CSV files, converting data, plotting data, and processing calculations in the same way they would during their academic research. Feedback will be gathered during these sessions to identify any areas requiring adjustment or enhancement. This testing phase is crucial for ensuring that the system is user-friendly and meets the designated requirements. Any issues identified will be prioritized for resolution to ensure the system is fully prepared for operational use.

# Environment Requirements

To ensure comprehensive testing of the mass spectrometer interface system, this section outlines the necessary and desired properties of the testing environment. The setup will enable thorough verification of each module’s functionality, accuracy, and performance in processing and visualizing mass spectrometry data.

The testing environment should be equipped with a Windows 10 or Windows 11 operating system. The environment will rely on Python, version 3.8 or higher, to match the development specifications of the system. Key Python libraries, such as PyQt5 for graphical user interface elements and pandas for handling data and CSV file operations, should be pre-installed. Each module has a requirements.txt file that can be used with pip to install the necessary libraries. Additionally, a sample CSV file will be prepared, including real mass spectrometer data, to evaluate the system’s handling of data directly from a file.

Special tools will aid in testing, such as the pytest framework for semi-automated unit and integration testing. Mocking tools, like unittest.mock or pytest-mock, will be used to simulate various data inputs and scenarios, particularly in modules focused on data handling and conversion. Performance monitoring, using tools like memory\_profiler or timeit, will help assess memory usage and processing time for modules dealing with larger datasets.

Module 4 has extra special requirements for its extra specific use case. The module takes in data from a Mass Spectrometer through an EZ-Tap serial-to-usb listener device. Both the Mass Spectrometer and the EZ-Tap are assumed for the testing and usage of module 4. In order to ensure these conditions, we use remote-access to test on the computer that has the EZ-Tap plugged into it.

These environment specifications aim to provide a controlled, reliable setting that ensures system stability, accuracy, and performance across diverse data scenarios and software setups.

# Glossary

CSV: Stands for comma-separated values which is a simple file format often used for storing data.

Mass Spectrometry: Mass spectrometry is an analytical technique that is used to measure the mass-to-charge ratio of ions.

# References