## Software Testing

To ensure the visualisation works as expected it must be adequately tested. The project utilises both automated and manual testing.

Automated testing is performed by software and runs when a change is made to the code. It is extremely useful as it is able to test the underlying code of the software, ensuring that it works as expected. Even if the failure condition is hard to replicate when interacting with the software manually.

On the other hand, manual testing is performed by a user who interacts with the front end of the software and verifies it operates as per the requirements. This type of testing can take a long time and cannot test the underlying code, only what the user can interact with [[1]](#footnote-1).

### Automated Testing

Automated testing is extremely useful during software development to ensure that future changes do not cause issues with previously developed features. The project utilises three, main, types of automated testing: unit testing, end-to-end testing and performance testing. The implementation of these techniques is described below.

#### Unit Testing

Unit tests are performed directly on a ‘unit’ of an application, typically a function or class [1]. Their purpose is to input data to the unit and compare the output to a known correct value. This allows them to assert that the unit works as expected. Unit tests should be run independently of other parts of the application as they only evaluate if a single unit works correctly in isolation.

One example from my project where I am utilising unit tests is to test if my complex number functions work as expected. I have set up unit tests to execute the function with set inputs and compare the output of my complex function with known correct values from when I executed the same calculation via MATLAB.

Below is an example of tests which are used to test my complex multiplication and division function. The third parameter of each TestVector object is the known correct value from MATLAB. Importantly, in JavaScript all numbers are stored as 64-bit floating point values. This means that values that are cannot be expressed in base 2 are unable to be stored precisely.

For example, the value is unable to be expressed in base 2, and is instead stored as . This presents major issues when attempting to test the equality of values, as JavaScript can state that . This was an issue when comparing my results from MATLAB to those generated in JavaScript. I resolved this issue by allowing a tolerance of correct values, checking if the final value was within of the expected value.

The unit testing module I built for this project can be found in the testing section of the Appendix.

Text

Description automatically generated

Figure 1 – Examples of complex number unit testing vectors

#### End-to-end Testing

End-to-end tests test that the application can meet the requirements by testing the entire workflow. The application has a testing mode which can be run to simulate various input configurations that the user may choose and ensures that the same, correct results are output on the final graph. This ensures that the entire application workflow works as intended and ensures that future changes do not alter known previously correct configurations.

For example, the testing vector below sets the user inputs to the specified values and compares the result to the expected output.

Graphical user interface, text

Description automatically generated

Figure 2 – An example of an end-to-end testing vector

When the test is executed the application updates to display the test inputs and automatically checks the output against the expected output defined in the test vector.

A screenshot of a computer

Description automatically generated with low confidence

Figure 3 – The above test vector being executed

The end-to-end testing module and additional test vectors can be found in the testing section of the appendix.

#### Performance Testing

It is important that when the user interacts with the visualisation that it quickly responds and updates the appropriate graphs. An earlier version of my application had severe performance issues where the user would interact with the visualisation, and it would take around 7 seconds to perform the necessary calculations and update the graphs to display the information to the user.

In order to help diagnose this issue I performed performance testing on my application to evaluate what functions of the program were causing the largest impact on performance. To do this, I used profiling tools to record exactly when each function was called and for how long it was executed.

I decided to use the Chromium performance profiler. Which is built into the DevTools on Chromium browsers. I did this as Chromium browsers have the highest desktop market share in 2023[3]. This means that it is likely that it is the most common browser students will be using to access the visualisation. So, it makes sense to use this tool to optimise the visualisation for Chromium browsers as it is likely to be the browser most students use to access the visualisation.

A picture containing graphical user interface

Description automatically generated

Figure 4 – Profiling chart taken pre-optimisation

As you can see from the profiler around 6 seconds of processing time was spent on the ‘plotDataWave’ function, this is far too long and indicates there is a performance problem with this function. We can further break this down and see that the issue resulted from how frequently the ‘getPositionAtTime’ function was being called.

Before performing performance testing I expected the performance bottleneck to be caused by the DFT function as this seemed to be the most computationally intensive function operating with time complexity. However, performance testing revealed this was not the case. This highlights the importance of performance testing as if I had not performed it I could have wasted time optimising part of the code with little impact on performance.



Figure 5 – Breakdown of function execution time

Having found that the bottleneck lay with plotting the graphs rather than with the computationally intensive DFT function, I focused my attention here and re-wrote how the graphs were to be updated. Instead of plotting each point on the graph individually I decided to instead just plot the data points and draw lines between the points.

I repeated the same interaction with the visualisation and again recorded the profiling, with the results shown below.

Timeline

Description automatically generated

Figure 6 –– Profiling chart, taken post-optimisation

This time only 33ms was spent on the task, which is a substantial improvement from 7 seconds. Now the functions which caused the largest impact on performance were the DFT and IDFT functions, which is what I would expect.

Due to performance testing, the user experience when using the visualisation was substantially improved and allows students to interact with the visualisation and receive feedback from their inputs almost immediately.

Doing performance testing was imperative for me to correctly identify which points of the code should be optimised and which operations would not affect performance. If I did not perform performance testing, I would have incorrectly assumed that the DFT was the part of the software which had the biggest impact on performance and wasted development time wrongly optimising that part of the software.

# References

[1] S. Pittet. *The different types of software testing.* Atlassian.com. [Online]. Available: <https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing> [Accessed 10 April 2023]

[2] A. Meixner. *The Test Attributes of Controllability and Observability.* accendoreliability.com. [Online]. Available: <https://accendoreliability.com/test-attributes-controllability-observability/> [Accessed 12 April 2023]

[3] Kinsta. *Global Desktop Browser Market Share for 2023.* kinsta.com. [Online]. Available: [Global Desktop Browser Market Share for 2023 - Kinsta®](https://kinsta.com/browser-market-share/) [Accessed 25 April 2023].

1. This is related to concepts known as observability and controllability [2]. Since manual tests typically cannot observe data when an application is running, they cannot test that the individual components are working as intended. This is less of an issue with automated testing as they typically can interact with the underlying code. [↑](#footnote-ref-1)