System Verification and Validation Plan
MTOBridge

Team 15, Alpha Software Solutions
Badawy, Adham
Yazdinia, Pedram
Jandric, David
Vakili, Farzad
Vezina, Victor
Chiu, Darren

April 5, 2023

1 Revision History

Date	Version	Notes
October 30	Darren	Non-Functional System Tests
November 1	Pedram	Functional System Tests
November 1	Victor	Unit Tests Intro
November 1	Victor	Non-Functional System Tests
November 2	Adham	Added Sections 3, 4.1, 4.2, 4.4, and 4.7
November 2	Victor	Non-Functional System Tests

Contents

List of Tables

List of Figures

2 Symbols, Abbreviations and Acronyms

Refer to section 1.4 of the SRS.

This document serves as a plan to verify and validate the design documents and outlines a series of system and unit tests for the functional and non functional requirements of the program. It is intended to be implemented as written, and for its results to be reported on in a future document.

3 General Information

3.1 Summary

The software whose Verification and Validation is planned for and referred to throughout this document is the User Interface component of MTO Bridge, a software developed in partnership with the Civil Engineering Department at McMaster university to assist MTO engineers in load rating bridges. Its primary function is to present visually, in an intuitive and digestible way, the results of a simulation written by a Civil Engineering Professor and her lab in MATLAB of the forces imparted on a bridge by a platoon of trucks driving over it. There are two different solvers for two different results about the forces imparted on the bridge. The program will allow users to specify details of the bridge, the truck platoon driving over it, and which of the two results they are interested in knowing more about.

3.2 Objectives

There are two overarching objectives to any V&V plan, ensuring we are actually building what we want to build correctly, or verification, and ensuring that we plan to build is what the client actually wants, or validation. Verification is where a lot of the domain specific definitions of "correct" come in. As far as the verification plans for the design documents, the main goal is to build confidence that the design documents are complete, verifiable, unambiguous, and generally of high quality. The design documents inform the implementation to a huge degree, so ensuring we are building on a solid foundation is one of the main goals of this plan. As we are building a user interface, non functional qualities such as usability and performance/responsiveness are of utmost importance, and many of the tests outlined in this document will center around building confidence that our program has those qualities, or at least exposing if it doesn't early on. As well, like any program, a number

of our tests focus around ensuring that our program as written actually does what we think it does, in a predictable way. Ensuring our program properly executes all the functions we laid out in the design documents, and nothing more or less.

3.3 Relevant Documentation

Almost every design document written for this project will be relevant documentation here, as the V&V plan involves validating the design documents as well as the implementation. Relevant documents are listed below. The SRS: The document involves a plan to Verify & Validate the SRS
The Hazard Analysis: The Hazard Analysis informs the SRS
The Module Guide: This document will be extremely relevant for this plan, particularly for unit testing
The Module Interface Spec: Same as the Module Guide.

4 Plan

The following section outlines a number of verification and validation plans for the design documents and the software. They are intended to act as a guide to be carried out in order to verify the documents.

4.1 Verification and Validation Team

Table 1: V&V Team

Name	Group	Role in V&V
Adham Badawy	Dev Team	Developer, will conduct the bulk of the verification reviews and run System and Unit Tests
David Jandric	Dev Team	Same as Adham Badawy
Victor Vezina	Dev Team	Same as Adham Badawy
Darren Chiu	Dev Team	Same as Adham Badawy
Farzad Vakili	Dev Team	Same as Adham Badawy
Pedram Yazdinia	Dev Team	Same as Adham Badawy
Dr Yang Cancan	Client	Primary source for requirements elicitation and software validation inquiries
Samuel Crawford	Teaching Staff	Provides holistic verification reviews of design documents via a marking rubric

4.2 SRS Verification Plan

For the SRS Verification, we will have a number of main goals, these are, in no particular order:

- Ensuring we have quality requirements
- Ensuring our requirements are complete
- Ensuring the document is free from grammar/spelling mistakes
- Ensuring the document is legible/easy to understand

For ensuring the quality of the requirements, we will conduct a number of reviews with a number of different objectives. The primary reviews will be conducted by the development team ourselves. And will be split into three separate reviews that will have the same methodology, but different goals. These will be as follows

- Checking requirements for an appropriate level of abstraction
- Checking requirements for testability/verifiability
- Checking requirements ambiguity

Two members of the team will be assigned to 2 of the effectively 6 reviews, a compromise between giving each task more than 1 set of eyes and maintaining a feasible schedule for the review. There are, after all, many other parts of this plan to be implemented. The assignments are as follows:

- **Adham&David**: Completeness & Testability
- **Pedram&Victor**: Ease of Understanding & Ambiguity
- **Darren&Farzad**: Spelling & Grammar & Abstraction

As well, these same criteria go for both the functional and non-functional requirements, and both will be reviewed in one pass. The subgroups will then discuss their findings at a meeting where any necessary changes to the SRS will be discussed. External, more holistic reviews of quality will also be conducted by 2 other teams enrolled in the course as well as the teaching staff, and this feedback will also be discussed amongst the group.

For requirements completeness, much of the same methodology of document readthroughs and reviews will occur, however the goal will be to identify any missing functionality through the review. The creation of other documents, such as the Hazard Analysis, is also done with ensuring SRS completeness in mind. The process of drafting these documents can illuminate new requirements previously unnoticed. This is also where the client, the Civil Eng. team we are working with, comes in. Of course, the requirements are based entirely off of discussions with them, however, continued confirmation that the project is implementing everything they want is important. It is not expected that the client will read the entire document, but we plan to prepare a quick summary of the requirements and present it to them and get their feedback. Naturally the client will also play a huge role in Software validation, but this will be discussed in Section 4.7.A final note on completeness,

it is impossible to prove that the SRS is truly, 100% complete, but the goal is that if we do our due diligence, we should get 99% of the way there, and the difference should be negligible.

Once again, for grammar/spelling the same methodology applies, however this review process will be entirely internal. 1-2 group members will be assigned to check the document and ensure no errors of this nature exist. This step, while simple, is still important for the understandability and disambiguation of the document.

Finally, the last goal with regards to the SRS; legibility and ease of understanding, is a very subjective one. The main idea here is that the document should present its information in as easy to digest a way as possible. Is related information located near to each other? Is there proper traceability between all parts of the document? Is the document easy to navigate? These are the kinds of questions the assigned group members will seek to answer. Once again, they will report their findings and any recommendations to the group in the nearest meeting to their completion of the review chronologically.

4.3 Design Verification Plan

MTO bridge is bringing value mainly through creating an accessible user-friendly platform for bridge engineers. Therefore, the associated usability verification process should be more elaborate and done with greater insistence. The second most important design aspect is anticipation of change since our development effort is limited by the number of team members as well as the duration of this course. As a result, it's important to verify our design output accommodates for potential future alterations. In our list of design metrics anticipation of change is divided into two design metrics flexibility and maintainability.

List of design metrics (in order of priority):

- **Usability**: How user friendly the system is.
  The Heuristic Evaluation checklist below will be used by some of our and Dr.Yang's team members (a group of 3 to 5 people) on a prototype which closely resembles the final product. Evaluators scrutinize individual elements according to the heuristics. They also examine how

these fit into the overall design, clearly recording all issues encountered. In a future session Evaluators can collate results for analysis and suggest fixes.

  - **Keep users informed about its status** appropriately and promptly.
  - **Show information in ways users understand** from how the real world operates, and in the users' language.
  - **Offer users control** and let them undo errors easily.
  - **Be consistent** so users aren't confused over what different words, icons, etc. mean.
  - **Prevent errors** – a system should either avoid conditions where errors arise or warn users before they take risky actions (e.g., "Are you sure you want to do this?" messages).
  - **Have visible information, instructions, etc. to let users recognize options, actions, etc.** instead of forcing them to rely on memory.
  - **Be flexible** so experienced users find faster ways to attain goals.
  - **Have no clutter**, containing only relevant information for current tasks.
  - **Provide plain-language help** regarding errors and solutions.

- **Flexibility:** The ease of expanding the product with new feature sets and capabilities.
  We shall refer the module guide (MG) and determine based on the stretch goals section of the problem statement and goals document the feasibility of decomposition as well as design flexibility. For example, one of our stretch goals was the ability to draw two dimensional models of bridges and have analysis performed based on the sketches. Graphical additions of this sort which might be common for our application would require a software design that decouples the "view" component as much as possible from other components. Our second stretch goal was having a customizable UI, consequently our sub-architecture for the "view" component should accommodate for a plug-and-play behaviour. A team members and volunteer class members from a different group

will be divided into two groups and each group is responsible for verification of flexibility of one stretch goal. They can provide us with feedback in separate meetings, after they were given time to examine our Module Guide, on whether our modular structure and overall architecture can facilitate for this design metric.

- **Maintainability:** The ease of coding bug repairs and adding minor features.
  Maintainability is greatly dependent on principles such as information hiding, code reusability and single-responsibility. The MG will be examined to particularly check for proper decomposition of modules and the "secrets" they hide through their interface. Base classes and inheritance should be utilized in cases where a general module can be decomposed into smaller sub-classes. The MIS (Module Interface Specification) shall be analyzed to ensure that every function is only responsible for doing one thing and if its other functionalities can be separated using new methods. Code inspections will be performed to identify repetitive logic and separate them using a helper function or class. The code inspections, MG and MIS analysis are going to be conducted by all team members. MG and MIS analysis will commence as soon as they are completed and code inspections begin when the first prototype is created. Each team member is going to be responsible for a component and its code inspection (same for MG and MIS assessments).

4.4 Verification and Validation Plan Verification Plan

Similar to other sections, the V&V verification plan involves a set of reviews conducted by our team and others. The goals for these reviews will be as follows:

- Ensure the plan is complete
- Ensure the plan is unambiguous
- Ensure the plan is feasible
- Ensure the plan actually accomplishes the objectives it says it does

For all of these reviews there are three major divisions for the document. Verification/Validation Plans, System Tests, and Unit Tests. These divisions exist as the definition of "complete, unambiguous, and feasible" is likely to be different, or at least they might be achieved differently for each of these sections, particularly between the plans and the tests. Howver, due to time constraints it would be infeasible to conduct 12 different reviews on this document, so each review will be assigned to one team member who will cover all three sections for their specific review. As for the SRS each member will complete their analysis, document their findings, and report them as well as any recommendations for changes to the group at the nearest available meeting. The assignments are as follows:

- **Victor**: Completeness
- **Pedram**: Unambiguousness
- **Darren**: Feasibility
- **Adham**: Overall accomplishment of Objectives

There will also be a more holistic and comprehensive single review that will be performed by the teaching staff for the capstone course, and some feedback from 1-2 other groups will be generated as well. Some more detailed notes on each of the goals follows below.

As far as completeness, the main aim here is to ensure that the plans and tests as written, if executed, would genuinely function as a comprehensive verification and validation of the program. As well, this plan should be easy to follow by a third party with no information provided besides this document and a primer on how to run the code. Similarly to the SRS verification, 100% completeness can never be guaranteed no matter how many reviews we do, but an effort to strive towards that goes a long way.

For ambiguity, this connects to what was talked about at the end of the completeness section. This plan is worthless if we can't actually follow it

and draw conclusions from it, so it needs to be clear what is to be done, how it is to be done, and what we should expect to get out of it. Making the plan unambiguous also helps this document to serve as a reliable test of the program. That is, if there's only one way to interpret the plans, and the document is well thought out, the plan should return mostly the same results if given the same input program and documents, every time.

Feasibility is simple, but important. There is always more testing that can be done, more through reviews that can be conducted, infinitely recursing verification plans and plans to verify the plans and plans to verify those plans, but at a certain point the recursion must stop and considerations for the fact that this plan is written with the intent to actually be carried out to the tee by a group of students with time constraints and other priorities must be made. A balance must be struck between accomplishing the objectives set out in this document to the best of our abilities, and not overestimating what those abilities actually are.

The last goal is somewhat related to all three of the previous ones. It basically functions as a final review of the document following the other three to double check the entire document more holistically, and ensure that it accomplishes the objectives laid out in section 3.2. If the document is complete, unambiguous, and feasible, this goal is likely already accomplished, but a holistic review will both not take too much time to complete following the information gathered from the previous three and also serve as a wrap up for the verification.

4.5 Implementation Verification Plan

- **Development tests:** Conducted with unit and integration level testing verifying the specified behavior of the sub-assemblies on the Module Interface Specification. Refer to sections 5 and 6 of this document for a detailed plan of system and unit tests.

- **Prototype testing:** Conducted with prototypes that closely resemble the final product. Each prototype created will immediately undergo exploratory testing where a potential user which in this case can be one of Dr.Yang's civil engineering students is encouraged to try different sections of the applications and complete specific tasks while being

monitored by the developers taking notes. Prototype testing can be conducted on an individual part or on the completed system under any conditions.

- **Analysis:** Since much of the effort is front-end related static analyzers might not be effective. Code inspection will be done by other team members when new implementation is getting merged to the sourcecode by a developer. We are going to put in place Github's merge validation mechanism where a developer should do a pull request so that their code can be validated by other team members before the new additions are submitted.

4.6 Automated Testing and Verification Tools

Linter
The IDEs we are using (Visual Studio and Visual Studio Code) have their own code linters, which are enabled by default. We may consider using clang-tidy for error checking, as well as looking questionable code constructs. We will only use it if the standard IDE linting it not sufficient. This tool is fairly standard, with integration in many IDEs. We can also have it look for code that is against certain coding standards, for example Google coding conventions. In order to maintain consistent code formatting, we will use clang-format for formatting. Using this linter, we can also easily select a coding standard to follow, which will come from a file in our repo. Many IDEs also support the use of clang-format.

Unit Testing Framework
We will be using a combination of two testing libraries to test our code. The first is Qt Test, a testing library made specifically for the Qt GUI framework we will be using. This testing library will be used to test the GUI elements of our project. The second testing library we will be using is Google Test. This library will allow us to test the other parts of our code in great detail, and offers many advanced features that we can use during development (report generation, death tests, automatic test discovery, etc.).

Code Coverage Measuring Tools
Since we will using GCC for compilation, we will be using GCOV to test our code coverage. We will use it as it comes standard with GCC, and will hook

in directly with our compilation.

4.7 Software Validation Plan

There will be multiple steps to validating our software, however they will all have the same goal. Answering this question: Are we building/did we build what our client wanted us to build, or has there been some terrible misunderstanding? We plan to have the validation process be a continuous one throughout the development of the project, as waiting until the program is mostly done and checking with the client once we have something concrete to show can lead to catastrophic refactoring if it turns out we weren't on the right track.

We already have biweekly meetings scheduled with the client for the lifespan of the project, every Friday at 10:00-11:00am. In the earlier stages of the project, these served as requirements elicitation meetings and an opportunity to deepen our understanding of the problem. Now that the project is well underway, however, these same meetings will switch to a primarily software validation role. As we make progress on the project, and lock in design decisions, we can run them by our primary stakeholder to ensure our vision remains aligned with theirs and make any relevant tweaks as necessary along the way, rather than all at once near the end. This keeps us flexible and avoids any nightmare "back to the drawing board" scenarios.

It was mentioned in section 4.2 that we plan on creating a summary of the main parts of the SRS and presenting to our client for a document verification purpose, but this exact methodology in a slightly more general sense would work perfectly for software verification as well. Distill what we are currently doing into a presentable set of ideas and receive feedback from the client on how well this solves their problem and how it could be improved/what to avoid.

The implementation of this section of the V&V plan may sound a little vague in comparison to others, and that's because it is. It's difficult to predict exactly how client meetings may affect our next steps or even refactor our requirements entirely, so the best way to track and report on the implementation of this section is likely in a set of meeting minutes from each of the biweekly meetings and subsequent team follow up meetings, cut down to

include only relevant V&V information, in the V&V report.

5 System Test Description

5.1 Tests for Functional Requirements

Below are system tests for the functional requirements in the SRS for MTO-Bridge. Each functional requirement has at least one system test. The identifiers follow a pattern of functional requirement number, system test number. For example, FR1.ST2 refers to the second system test for functional requirement 1.
As well, many of the tests below utilize the same valid or invalid truck or bridge configuration for input. For the sake of space and legibility, these template configurations are listed below, and will be referred to by their names in the input section of the system tests. They are TruckConfig1, TruckConfig2, BridgeConfig1, and BridgeConfig2, SolverConfigConcerned, and SolverConfigCritical, 1 refers to a valid configuration, 2 to an invalid one.
TruckConfig1

- NumberOfTrucks: 3
- Headway: 15
- AxleLoads: [52 52 52 52]
- AxleSpacings: [7 3 2]

TruckConfig2

- NumberOfTrucks: I
- Headway: Love
- AxleLoads: Software
- AxleSpacings: Enginering

BridgeConfig1

- NumberOfSpans: 3
- SpanLength: [40 40 40]

- ConcernedSection: 10
- DiscretizationLength: 0.1m

BridgeConfig2

- NumberOfSpans: Cats
- SpanLength: hate
- ConcernedSection: getting
- DiscretizationLength: wet

SolverConfigConcerned

- ForceType: Positive Moment
- SolverType: Concerned Section

SolverConfigCritical

- ForceType: Negative Moment
- SolverType: Critical Section

5.1.1 MATLAB

MATLAB Communication

1. FR1.ST1

   Control: Manual

   Initial State: None

   Input: None

   Output: MATLAB engine started

   Test Case Derivation: The first step in the communication is to get the MATLAB engine to start.

   How test will be performed: The test will instantiate our MATLAB engine wrapper.

2. FR1.ST2

   Control: Manual

   Initial State: None

   Input: TruckConfig1, BridgeConfig1, and SolverConfigConcerned

   Output: Any result from standard out or standard error

   Test Case Derivation: We just need to confirm that the engine is processing our input in some way, whether it gives an error or not.

   How test will be performed: Start the MATLAB engine through our wrapper, create some default truck, bridge, and analysis configurations, and try to run the analysis.

5.1.2 Truck Configuration

Truck Configuration Input

1. FR2.ST1

   Control: Manual

   Initial State: GUI exists

   Input: Input corresponding to TruckConfig1

   Output: GUI fields are filled

   Test Case Derivation: If the data of the input fields can be entered, then the user should also be able to fill in the fields.

   How test will be performed: Input fields will filled, and checked if they have the correct value. That is, a value that corresponds to the input provided.

2. FR2.ST2

   Control: Manual

   Initial State: GUI exists

   Input: Input corresponding to TruckConfig2

Output: GUI fields are filled, any invalid ones are highlighted as incorrect

Test Case Derivation: The user should be able to enter data, but be informed if the entered information is invalid.

How test will be performed: Input fields will filled with incorrect information, and the fields should be highlighted as incorrect.

Truck Platoon Visualization

1. FR3.ST1

   Control: Manual

   Initial State: GUI exists

   Input: TruckConfig1

   Output: A valid visualization of the truck platoon shown

   Test Case Derivation: If a valid truck configuration is given, the user should be able to visualize the truck platoon.

   How test will be performed: Given a valid truck configuration, a visualization will be generated. The visualization will be saved as a picture and compared to the expected visualization.

2. FR3.ST2

   Control: Manual

   Initial State: GUI exists

   Input: TruckConfig2

   Output: A message informing the user of an incorrect truck configuration

   Test Case Derivation: If an invalid truck configuration is given, the user should be informed that the visualization cannot be created without a valid truck configuration.

   How test will be performed: Given an invalid configuration, the program will not show a visualization, but a message informing the user that a visualization cannot be created/shown until the configuration is valid.

Save Truck Configuration

1. FR4.ST1

   Control: Manual

   Initial State: GUI exists

   Input: TruckConfig1

   Output: A file with the correctly saved configuration

   Test Case Derivation: The program should be able to take the existing truck configuration and save it so that it can be loaded or used later.

   How test will be performed: Given a truck configuration, a file will be created. Its contents will be compared with the expected contents.

Load Truck Configuration

1. FR5.ST1

   Control: Manual

   Initial State: GUI exists, a truck configuration file exists

   Input: Truck configuration file created from TruckConfig1

   Output: Truck configuration input fields will be filled correctly

   Test Case Derivation: After loading a previously saved truck configuration file, the configuration should apply and fill the input fields for the truck configuration.

   How test will be performed: Given a truck configuration, a file will be created. Its contents will be compared with the expected contents.

5.1.3 Bridge Configuration

Bridge Configuration Input

1. FR6.ST1

   Control: Manual

Initial State: GUI exists

Input: Input corresponding to BridgeConfig1.NumberOfSpans and BridgeConfig1.SpanLength

Output: GUI fields are filled

Test Case Derivation: If the data of the input fields can be entered, then the user should also be able to fill in the fields.

How test will be performed: Input fields will filled, and checked if they have the correct value, that is, a value that corresponds to the input provided

2. FR6.ST2

Control: Manual

Initial State: GUI exists

Input: Input corresponding to BridgeConfig2.NumberOfSpans and BridgeConfig2.SpanLength

Output: GUI fields are filled, any invalid ones highlighted as incorrect

Test Case Derivation: The user should be able to enter data, but be informed if the entered information is invalid.

How test will be performed: Input fields will filled with incorrect information, and the fields should be highlighted as incorrect.

Bridge Visualization

1. FR7.ST1

Control: Manual

Initial State: GUI exists

Input: BridgeConfig1

Output: A valid visualization of the bridge shown

Test Case Derivation: If a valid bridge configuration is given, the user should be able to visualize the bridge.

How test will be performed: Given a valid bridge configuration, a visualization will be generated. The visualization will be saved as a picture and compared to the expected visualization.

2. FR7.ST2

   Control: Manual

   Initial State: GUI exists

   Input: BridgeConfig2

   Output: A message informing the user of an incorrect bridge configuration

   Test Case Derivation: If an invalid bridge configuration is given, the user should be informed that the visualization cannot be created without a valid bridge configuration.

   How test will be performed: Given an invalid configuration, the program will not show a visualization, but a message informing the user that a visualization cannot be created/shown until the configuration is valid.

Save Bridge Configuration

1. FR8.ST1

   Control: Manual

   Initial State: GUI exists

   Input: BridgeConfig1

   Output: A file with the correctly saved configuration

   Test Case Derivation: The program should be able to take the existing bridge configuration and save it so that it can be loaded or used later.

   How test will be performed: Given a bridge configuration, a file will be created. Its contents will be compared with the expected contents.

Load Bridge Configuration

1. FR9.ST1

   Control: Manual

   Initial State: GUI exists, a bridge configuration file exists

   Input: Bridge configuration file created from BridgeConfig1

   Output: Bridge configuration input fields will be filled correctly

   Test Case Derivation: After loading a previously saved truck configuration file, the configuration should apply and fill the input fields for the truck configuration.

   How test will be performed: Given a bridge configuration, a file will be created. Its contents will be compared with the expected contents.

5.1.4 Solver Setup

Solver Selection

1. FR10.ST1

   Control: Manual

   Initial State: None

   Input: SolverConfigConcerned or SolverConfigCritical

   Output: Calculation results according to the selected solver

   Test Case Derivation: It is important to ensure the program is properly accepting the user's solver choice, as concerned and critical section will have different results.

   How test will be performed: The test involves running a series of predetermined calculations using both solvers and then comparing with the expected output.

Section of Concern

1. FR11.ST1

   Control: Manual

   Initial State: GUI exists

   Input: Input Corresponding to BridgeConfig1-¿ConcernedSection

   Output: GUI field is filled

   Test Case Derivation: If the data of the input fields can be entered, then the user should also be able to fill in the fields.

   How test will be performed: Input fields will be filled, and checked if they have the correct value.

Discretization Length

1. FR12.ST1

   Control: Manual

   Initial State: GUI exists

   Input: Input Corresponding to BridgeConfig1-¿Discretization Length

   Output: GUI field is filled

   Test Case Derivation: If the data of the input fields can be entered, then the user should also be able to fill in the fields.

   How test will be performed: Input fields will be filled, and checked if they have the correct value.

Force Type

1. FR13.ST1

   Control: Manual

   Initial State: GUI exists

   Input: Input Corresponding to SolverConfigConcerned-¿ForceType where ForceType = Positive Moment

   Output: GUI field is filled

Test Case Derivation: The user can select the type of force to calculate for. If they select the force type it should reflect in the GUI by filling the field properly.

How test will be performed: Input fields will filled, and checked if they have the correct value.

2. FR13.ST2

   Control: Manual

   Initial State: GUI exists

   Input: Input Corresponding to SolverConfigConcerned-¿ForceType where ForceType = Negative Moment

   Output: GUI field is filled

   Test Case Derivation: The user can select the type of force to calculate for. If they select the force type it should reflect in the GUI by filling the field properly.

   How test will be performed: Input fields will filled, and checked if they have the correct value.

3. FR13.ST3

   Control: Manual

   Initial State: GUI exists

   Input: Input Corresponding to SolverConfigConcerned-¿ForceType where ForceType = Shear

   Output: GUI field is filled

   Test Case Derivation: The user can select the type of force to calculate for. If they select the force type it should reflect in the GUI by filling the field properly.

   How test will be performed: Input fields will filled, and checked if they have the correct value.

4. FR13.ST4

Control: Manual

Initial State: None

Input: TruckConfig1, BridgeConfig1, and SolverConfigConcerned

Output: Calculation results based on the selected parameters

Test Case Derivation: The current implementation of the engine allows for calculation using positive and negative moment

How test will be performed: The test will be performed by using our pool of pre-determined bridge and truck platoon configs to test for both positive and negative moment given a certain force type.

5.1.5 Result Visualization

Concerned Section Result Visualization

1. FR14.ST1

   Control: Manual

   Initial State: None

   Input: calculations based on the concerned section

   Output: Animation and visualization highlighting the resulting properties on the specified bridge and load

   Test Case Derivation: As one of the main purposes of the program, the visualization will be interactive to show various views

   How test will be performed: Relying on the pre-determined and validated data, we will graph the visualizations of each through the system as well as manually through the data produced by the engine. We can then compare the the expected and actual result across all available views.

Critical Section Result Visualization

1. FR15.ST1

   Control: Manual

Initial State: None

Input: calculations based on the discretization length

Output: Animation and visualization highlighting the resulting properties on the specified bridge and load

Test Case Derivation: As one of the main purposes of the program, the visualization will be interactive to show various views

How test will be performed: Relying on the pre-determined and validated data, we will graph the visualizations of each through the system as well as manually through the data produced by the engine. We can then compare the the expected and actual result across all available views.

5.1.6 Exectution Flow Logging

Report Generation

1. FR16.ST1

   Control: Manual

   Initial State: None

   Input: Input to indicate logging flags

   Output: text output, "logs"

   Test Case Derivation: With the engine built in another framework, its important to log every state change throughout the calculations as triggered by the system

   How test will be performed: While running multiple calculations, the system is asked to comprehensively log the flow in the engine. The system will be interrupted at times to test the robustness as well. The resulting logs should be clearly highlighting the steps taken including any errors or warnings.

5.1.7 Error Handling

1. FR17.ST1

Control: Manual

Initial State: GUI exists

Input: TruckConfig1, BridgeConfig1, SolverConfigConcerned

Output: An animation of the calculation results where the animation and the calculation results graph are to scale.

Test Case Derivation: as the program will be animating the results of the platoon's trip over the bridge, it is important for visual clarity that the animation displayed matches up with the cartestian results displayed, scale-wise.

How test will be performed: The calculation will be run with valid inputs and the resulting animation will be compared to the x-axis of the graph, with the result of this comparison compared to the orcale.

1. FR18.ST1

Control: Manual

Initial State: MATLAB and GUI Threads are initialized

Input: TruckConfig2, BridgeConfig2, SolverConfigConcerned

Output: MATLAB thread crashes, remainder of the program continues running as normal.

Test Case Derivation: As this is a multithreaded program, it is important to ensure that it is thread safe, and that one thread crashing will not affect the other.

How test will be performed: With an invalid set of inputs the caluclation will call the MATLAB functions, which will cause the MATLAB to crash, the status of the GUI thread will then be observed.

1. FR19.ST1

Control: Manual

Initial State: GUI exists

Input: TruckConfig1 but with NumberOfTrucks changed to be 0 instead of 3

Output: The input is rejected and not passed on to the solver component

Test Case Derivation: The program cannot allow users to enter invalid values, as this could lead to inaccurate results at best, or even NANs at worst if the values are too wild or if a division by 0 occurs.

How test will be performed: The invalid TruckConfig1 tweak will be inputted to the system and what the system does with that input will be observed and compared to the oracle.

1. FR20.ST1

   Control: Manual

   Initial State: GUI Exists

   Input: TruckConfig2

   Output: The input is rejected and not passed on to the solver component

   Test Case Derivation: The program cannot allow users to enter invalid values, as this could lead to inaccurate results at best, or even NANs at worst if the values are too wild or if a division by 0 occurs.

   How test will be performed: The invalid TruckConfig2 that has strings for fields will be inputted to the system and what the system does with that input will be observed and compared to the oracle.

5.2 Tests for Nonfunctional Requirements

5.2.1 Look and Feel Requirements

1. NFR1.ST1

   Type: Manual, Static

   Initial State: Bridge UI mockups containing graphic elements will be prepared

   Input/Condition: Civil engineers are presented with a UI mockup as seen in Figure 1

Output/Results: The engineers can correctly identify what part of a bridge each UI element corresponds to in 90% of cases.

How test will be performed: The developers will present a UI mockup to civil engineers and will assess how accurately the engineers can identify the UI elements.

5.2.2 Usability and Humanity Requirements

1. NFR2.ST1

   Type: Manual, Dynamic

   Initial State: Application is opened with no past data

   Input/Condition: Civil engineers are presented with the application and a brief explanation

   Output/Results: 90% of the civil engineers are able to perform bridge analysis within 5 minutes of introduction

   How test will be performed: Civil engineers will be presented with the application and a brief explanation by the developers, then the amount of time it takes them to complete a bridge analysis will be measured.

2. NFR3.ST1

   Type: Manual, Dynamic

   Initial State: Application is opened

   Input/Condition: Application is running on a computer with 1280x720 display

   Output/Results: All buttons and animation window remain visible and accessible

   How test will be performed: The application will be manually opened on a 1280x720 display and checked to ensure that all buttons and animation windows are still visible.

3. NFR3.ST2

   Repeat test NFR.ST3 for 1366x768 display

4. NFR3.ST3

   Repeat test NFR.ST3 for 1920x1080 display

5. NFR3.ST4

   Repeat test NFR.ST3 for 2560x1440 display

6. NFR3.ST5

   Repeat test NFR.ST3 for 3840x2160 display

7. NFR4.ST1

   Type: Manual, Dynamic

   Initial State: Application is opened with no past data

   Input/Condition: Font size setting is changed to 8pt

   Output/Results: All buttons and animation window remain visible and accessible

   How test will be performed: The application will be manually opened and set to use 8pt font, then checked to ensure that all buttons and animation windows are still visible.

8. NFR4.ST2

   Repeat test NFR.ST8 with font size 16pt

9. NFR4.ST3

   Repeat test NFR.ST8 with font size 24pt

10. NFR4.ST4

    Repeat test NFR.ST8 with font size 32pt

11. NFR5.ST1

    Type: Manual, Dynamic

    Initial State: Application is not installed

    Input/Condition: Application is downloaded and installed

    Output/Results: Application is ready to use within 30 minutes

    How test will be performed: The application will be manually downloaded (on an internet connection with at least a 10Mbps download speed), installed, and run. This process will be timed to ensure that it takes less than 30 minutes.

12. NFR6.ST1

    Type: Manual, Static

    Initial State: Visually similar UI elements are grouped into categories by developers

    Input/Condition: Civil engineer are asked to associate UI elements with their respective categories

    Output/Results: Civil engineer sorts at least 90% of UI elements into their predefined categories

    How test will be performed: The developers will present a UI mockup to civil engineers and will assess how accurately the engineers can correctly group the UI elements.

5.2.3 Performance Requirements

1. NFR7.ST1

    Type: Manual, Dynamic

    Initial State: Application is opened with no past data

    Input/Condition: The application is run with invalid inputs (zeroes, negative numbers, strings, etc.)

    Output/Results: The application does not freeze or crash

How test will be performed: The application will be automatically run with many different invalid inputs, and it will be ensured that the application does not freeze or crash

2. NFR8.ST1

   Type: Manual, Dynamic

   Initial State: Application is installed but the MATLAB files are not installed

   Input/Condition: The application is opened

   Output/Results: The application will display an error regarding the missing MATLAB files

   How test will be performed: The application will be manually installed without the required MATLAB files, and then it will be opened

3. NFR9.ST1

   Type: Manual, Dynamic

   Initial State: Application is opened with no past data

   Input/Condition: Various UI elements are interacted with

   Output/Results: The UI elements react to the interaction within 100ms

   How test will be performed: Various interactions with UI elements will be simulated on a computer with hardware similar to that of an MTO engineer's, and the speed of the UI response will be measured

4. NFR10.ST1

   Type: Manual, Dynamic

   Initial State: Application is opened with no past data

   Input/Condition: Various slightly tweaked but still valid versions of TruckConfig1, BridgeConfig1, and SolverConfigConcerned

   Output/Results: Total execution time of calculations will not exceed underlying MATLAB script's execution time by more than 10%

How test will be performed: The calculation will be run using the various inputs and timed on a computer with hardware similar to that of an MTO engineer's.

5.2.4 Maintainability and Support Requirements

1. NFR13.ST1

   Type: Manual, Static

   Initial State: The program code is contained in multiple files

   Input/Condition: Measure code file length in lines

   Output/Results: At least 75% of files will contain 750 lines of code or less

   How test will be performed: File legnth will be manually reviewed

2. NFR13.ST2

   Type: Manual, Static

   Initial State: The program code is contained in multiple files

   Input/Condition: Measure method length in lines

   Output/Results: At least 75% of methods will contain 75 lines of code or less

   How test will be performed: Method length will be manually reviewed

3. NFR13.ST3

   Type: Automatic, Static

   Initial State: The program code is contained in multiple files

   Input/Condition: Measure length of code lines in characters

   Output/Results: At least 75% of lines will contain 120 characters or less

   How test will be performed: The clang-tidy linter will be used to automatically measure line length

4. NFR13.ST4

   Type: Automatic, Static

   Initial State: The program code is contained in multiple files

   Input/Condition: Measure nesting depth of methods

   Output/Results: At least 75% of methods will have a maximum nesting depth of 5 or less

   How test will be performed: The clang-tidy linter will be used to automatically measure method nesting depth

5.3 Traceability Between Test Cases and Requirements

Table 2: Functional System Test Traceability

Requirement	**System Tests**
FR1	FR1.ST1, FR1.ST2
FR2	FR2.ST1, FR2.ST2
FR3	FR3.ST1, FR3.ST2
FR4	FR4.ST1
FR5	FR5.ST1
FR6	FR6.ST1, FR6.ST2
FR7	FR7.ST1, FR7.ST2
FR8	FR8.ST1
FR9	FR9.ST1, FR9.ST2
FR10	FR10.ST1
FR11	FR11.ST1
FR12	FR6.ST1
FR13	FR13.ST1, FR13.ST2
FR14	FR14.ST1
FR15	FR15.ST1
FR16	FR16.ST1
FR17	FR17.ST1
FR18	FR18.ST1
FR19	FR19.ST1
FR20	FR20.ST1

Table 3: Non-Functional System Test Traceability

Requirement	**System Tests**
NFR1	NFR1.ST1
NFR2	NFR2.ST1
NFR3	NFR3.ST1, NFR3.ST2, NFR3.ST3, NFR3.ST4, NFR3.ST5
NFR4	NFR4.ST1, NFR4.ST2, NFR4.ST3, NFR4.ST4
NFR5	NFR5.ST1
NFR6	NFR6.ST1
NFR7	NFR7.ST1
NFR8	NFR8.ST1
NFR9	NFR9.ST1
NFR10	NFR10.ST1
NFR11	FR3.ST1, FR7.ST1, FR10.ST1, FR11.ST1, FR12.ST1, FR13.ST1, FR13.ST2
NFR12	NFR9.ST1, NFR10.ST1
NFR13	NFR13.ST1, NFR13.ST2, NFR13.ST3, NFR13.ST4

6 Unit Test Description

As outlined in our Development Plan, we will be using Qt Test to conduct our unit testing. This testing framework will allow us to create and run unit tests quickly and easily.

The general philosophy for the unit testing will be to test each individual fucntion/method of each module in two main categories: ensuring it provides the expected output when a valid input is provided, and ensuring it properly handles the error when an invalid input is required. For the sake of brevity, and to avoid repeating information, an exhaustive list of all unit tests will not be created in this document. Instead, to view the unit tests utilzed to test the implementation, please check this folder on our Github.

6.1 Unit Testing Scope

The main note here is that all backend MATLAB functions are out of scope for our testing, as these were developed by Dr Yang and her team. Our base assumption for testing is that these functions will work as intended in all cases, and if something goes wrong, it is the fault of our own implementation.

7 Appendix

This is where you can place additional information.

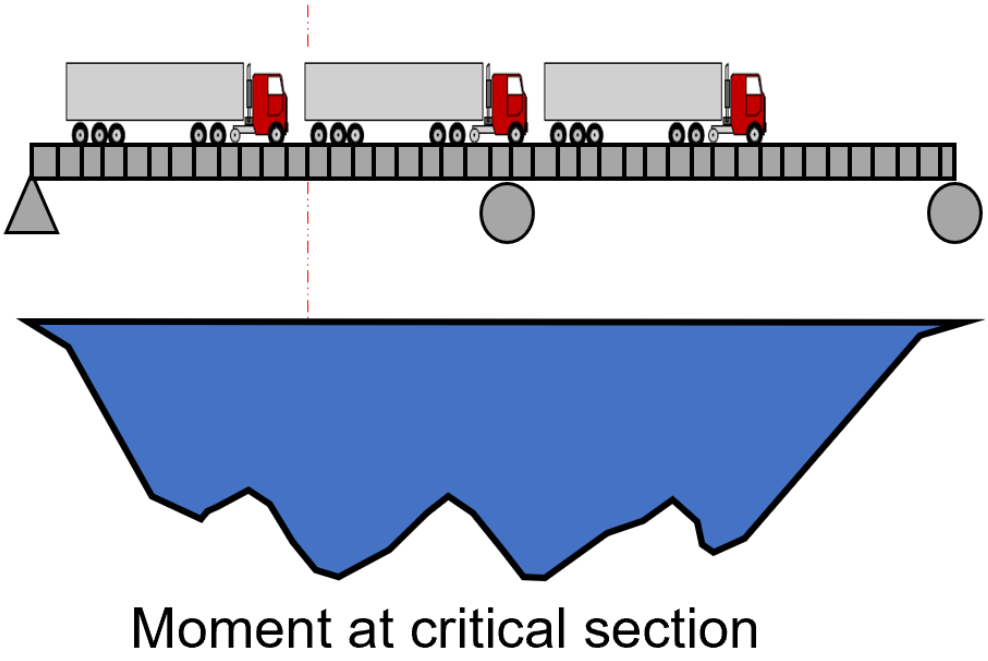

Figure 1: UI mockup

7.1 Reflection

The information in this section will be used to evaluate the team members on the graduate attribute of Lifelong Learning. Please answer the following questions:

1. What knowledge and skills will the team collectively need to acquire to successfully complete the verification and validation of your project? Examples of possible knowledge and skills include dynamic testing knowledge, static testing knowledge, specific tool usage etc. You should look to identify at least one item for each team member.

2. For each of the knowledge areas and skills identified in the previous question, what are at least two approaches to acquiring the knowledge or mastering the skill? Of the identified approaches, which will each team member pursue, and why did they make this choice?

Darren
Most Important Learning(s): We intend to use tools for validation including setting up CI/CD on the repo and clang-tidy for code analysis. Although I've used these or similar tools in the past, I haven't personally prepared them before.
Plan to learn what is needed: Some approaches for learning this is to research into the respective tools and reviewing past projects that used them, or to participate in their implementation and collaborate with team members on this. These aren't mutually exclusive, but I intend to mainly focus on learning through participating in the implementation and working with team members, since I feel that will be more effective in bringing results to the project.

Adham
Most Important Learning(s): My biggest blindspot with regards to the V&V plan is the finer points of software testing, I think. While i am still relatively familiar with concepts such as different types of testing, coverage metrics, and what have you, I'm definitely more than a little rusty.
Plan to learn what is needed: I still have access to my 3S03(software testing) slides, so I plan to use those to brush up on the concepts and techniques. As well, there are a wealth of online videos and resources that I can use should i require more information than what was in my course. Primary source is

definitely going to be the slides, though. Probably slides-teammates-internet, in that order.

David
Most Important Learning(s): I think I need to learn a lot more about testing and test driven development. Also I've never used a static analysis tool other than linters, so I need to learn what rules and static analysis is necessary.
Plan to learn what is needed: To learn more about testing I'm going to look over some old course material, and refer to online resources. I think I will also need to visit the documentation for Google Test because it's a testing framework I've never used. To learn more about static analysis I think online resources will be the best bet, and also visiting the documentation for clang-tidy will help with knowing the types of static analysis that are available to us.

Victor
Most Important Learning(s): Something I need to learn a lot about is the Qt Test library that we will be using for UI testing. I only have cursory knowledge of the Qt library, so the testing framework made to work with it is very foreign to me. I have also never worked with a testing framework designed to test UIs, so I believe that will take some time to get used to.
Plan to learn what is needed: I plan to take two approaches to learn about Qt Test: research and learning by doing. First, I will perform some preliminary research into how to use the library and what functions it can and can't perform. This research will give me the initial knowledge I need to start working with the library. I will then learn more about the framework through using it, especially when working with David who has used it before. I find that learning by doing is very effective when it comes to software, especially libraries, and it works even better when working with someone who has experience.

Farzad
Most Important Learning(s): I realized I need to learn much more about automated testing. Specifically tools that can perform a set of testing procedures on the UI. Redoing all of the previous test cases on by one is going to be time consuming and error prone, while not doing it can put the robustness and correctness of the program at risk.

Plan to learn what is needed: Research different approaches that experienced front-end developers have employed to test their UI. Afterwards, understand which approach/tool is most applicable to our case and dive deep in their documentation and use cases.Finally, I have to set it up and test it myself to see if they are effective.

Pedram
Most Important Learning(s): For this part of the project, specifically the testing and verification, I need to increase my dynamic testing knowledge of C++ as well as Qt which will be the main development framework. I need to spend the time understanding testing automation at different levels including unit and integration level testing. Static Analysis tools are also very common and include diagrams meant to highlight different components of the system.
Plan to learn what is needed: In terms of the next steps and my plan to achieve what's needed, I believe I will mostly rely on explorative testing to learn and test different components at the same time. I will also take advantage of online resources as well as documentations. Each member has also had a unique experience with C++ testing which can potentially help us enable each other in further verifying the design.