BoGL Web: An Application for Generating Bond Graphs on the Web

A Major Qualifying Project

Submitted to the Faculty of

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

Margaret Earnest (CS, minor in RBE)

Jakob Misbach (CS)

Anthony Vuolo III (MA and CS)

Date:

April 27, 2023

Report Submitted to:

Professor David C. Brown (CS)

Professor Pradeep Radhakrishnan (ME)

Professor Brigitte Servatius (MA)

I. Abstract

In this project, we designed and implemented a web-based software application, BoGL Web, which allows users to graphically construct system diagrams and bond graphs. These are used in mechanical engineering to model mechanisms and derive state equations from them. The software replicates and extends the existing BoGL Desktop system. Additional features include exporting system diagrams to URLs for future use and undo/redo capability.

II. Acknowledgements

We would like to thank Professor Brown, Professor Radhakrishnan, and Professor Servatius for all their help in this project. We would also like to thank the WPI teams that worked on the *Graph Grammar Based Automated Virtual Lab for Bond Graphs* and *BoGL: An Application for Generating Bond Graphs* MQPs for developing the functionality and appearance of BoGL Desktop, which BoGL Web is based on. Finally, we would like to thank the WPI students of Modeling and Analysis of Mechatronic Systems (ME 4322) who participated in our user survey and helped us test and improve BoGL Web.

III. Authorship

The following table describes which sections of the report each team member wrote. After writing the initial drafts, all team members conducted peer review.

Section	Author
1. Introduction	Jakob Misbach
2. Motivation	Jakob Misbach
3. Background	Anthony Vuolo
3.1 Graphs	Anthony Vuolo
3.2 Bond Graphs	Anthony Vuolo
3.2.1 List of Elements	Anthony Vuolo
3.2.2 Bonds	Anthony Vuolo
3.2.3 Sources of Flow and Effort	Anthony Vuolo
3.2.4 R-Elements	Anthony Vuolo
3.2.5 Storage Elements	Anthony Vuolo
3.2.6 Transformers	Anthony Vuolo
3.2.7 Gyrators	Anthony Vuolo
3.2.8 1-Junctions and -=Junctions	Anthony Vuolo
3.2.9 Domain-Specific Effort-Flow Equations	Anthony Vuolo
3.3 Kirchhoff's Laws	Anthony Vuolo
3.3.1 River Networks	Anthony Vuolo
3.3.2 Kirchhoff's Circuit Laws	Anthony Vuolo
3.3.3 Finding Current and Voltage in an Electrical Network	Anthony Vuolo
3.4 Bond Graphs of Mechanical and Electrical Systems	Anthony Vuolo
3.5 Generation of State Equations	Anthony Vuolo
3.6 BoGL MQP	Anthony Vuolo
4 Project Requirements	Jakob Misbach
5 Literature Review	Jakob Misbach
5.1 ASP.NET	Jakob Misbach
5.1.1 Angular and ASP.NET	Jakob Misbach
5.1.2 React and ASP.NET	Jakob Misbach
5.2 OpenSilver	Jakob Misbach
5.3 Blazor	Jakob Misbach
5.3.1 Blazor WebAssembly	Jakob Misbach
5.3.2 Blazor Server	Jakob Misbach
5.4 UI Component Libraries	Margaret Earnest

5.4.1 UI Component Library Options	Margaret Earnest
5.4.1.1 All Tech Stacks	Margaret Earnest
5.4.1.2 Blazor and OpenSilver	Margaret Earnest
5.4.1.3 React	Margaret Earnest
5.4.1.4 Angular	Margaret Earnest
5.5 Graph Drawing	Anthony Vuolo
5.5.1 Characteristics of Bond Graphs	Anthony Vuolo
5.5.2 Drawing Algorithms: Force-Directed Optimization	Anthony Vuolo
5.5.3 Drawing Algorithms: Spectral Graph Optimization	Anthony Vuolo
6 Methodology	Jakob Misbach
6.1 Topic Research Methodology	Jakob Misbach
6.2 Choosing a UI Component Library	Margaret Earnest
6.2.1 Judging Criteria	Margaret Earnest
6.2.1.1 Three Point Components	Margaret Earnest
6.2.1.2 Two-point Components	Margaret Earnest
6.2.1.3 One Point Components	Margaret Earnest
6.2.2 Choosing Ant Design	Margaret Earnest
6.2.2.1 Decision Matrix	Margaret Earnest
6.2.2.2 Conclusion	Margaret Earnest
6.3 Tech Stack Methodology	Jakob Misbach
6.3.1 Categories	Jakob Misbach
6.3.2 Scoring of Categories	Jakob Misbach
6.3.3 Usability	Jakob Misbach
6.3.3.1 Offline Usability	Jakob Misbach
6.3.3.2 Ability to Replicate UI	Jakob Misbach
6.3.4 Speed	Jakob Misbach
6.3.4.1 Client Load Speed	Jakob Misbach
6.3.4.2 UI Interaction Speed	Jakob Misbach
6.3.4.3 Computation Speed	Jakob Misbach
6.3.5 Space	Jakob Misbach
6.3.5.1 Client-Side	Jakob Misbach
6.3.5.2 Server-Side	Jakob Misbach
6.3.6 Programmability	Jakob Misbach
6.3.6.1 Code Reusability	Jakob Misbach
6.3.6.2 Ease of Writing Code	Jakob Misbach
6.3.6.3 Community Support	Jakob Misbach
6.3.6.4 Server Interaction	Jakob Misbach
6.3.7 Final Scores	Jakob Misbach
6.4 BoGL Desktop Port Methodology	Jakob Misbach
6.5 UI Prototyping Methodology	Margaret Earnest
6.5.1 Top Bar Dropdown Menus	Margaret Earnest

6.5.2 System Diagram Editor	Margaret Earnest
6.5.3 Integrating the System Diagram Editor with the Element Menu	Margaret Earnest
6.5.4 Dragging Edges Prototype	Margaret Earnest
6.5.5 Tab Prototype	Margaret Earnest
6.6 Development Methodology	Jakob Misbach
6.6.1 Tools Used	Jakob Misbach
6.6.1.1 Organization Tools	Jakob Misbach
6.6.1.2 Version Control	Jakob Misbach
6.6.1.3 Integrated Development Environment	Jakob Misbach
6.6.1.4 Debugging	Jakob Misbach
6.6.1.5 Deployment	Jakob Misbach
6.6.2 Development Order	Jakob Misbach
6.6.2.1 Backend	Jakob Misbach
6.6.2.2 Frontend	Margaret Earnest
7 Design	Jakob Misbach
7.1 Undo/Redo Design	Margaret Earnest
7.1.1 Undo/Redo Action Types	Margaret Earnest
7.1.1.1 Add Selection	Margaret Earnest
7.1.1.2 Delete Selection	Margaret Earnest
7.1.1.3 Change Selection	Margaret Earnest
7.1.1.4 Change Selection Modifier	Margaret Earnest
7.1.1.5 Move Selection	Margaret Earnest
7.1.1.6 Change Selection Velocity	Margaret Earnest
7.1.2 Canvas Change	Anthony Vuolo
7.1.3 Undo/Redo Stack Design: “EditionList”	Anthony Vuolo
7.2 State Equation Design	Anthony Vuolo
7.2.1 Backend Design	Anthony Vuolo
7.2.2 State Equation UI Mockups	Jakob Misbach
7.2.2.1 Requirements	Jakob Misbach
7.2.2.2 Mockups for Adding Values and Labels	Jakob Misbach
7.2.2.3 Mockups for Viewing Differential and State Equations	Jakob Misbach
7.2.2.4 Mockups for Viewing Labels	Jakob Misbach
7.2.2.5 Decision Matrices	Jakob Misbach
7.2.2.6 Recommendations	Jakob Misbach
7.3 Tutorial Design	Jakob Misbach
8 Implementation	Jakob Misbach
8.1 Backend	Jakob Misbach
8.1.1 BoGL Web Backend Data Flow Diagram	Jakob Misbach
8.1.2 Rules and Rulesets Implementation	Jakob Misbach
8.1.2.1 Design Patterns	Jakob Misbach

8.1.2.2 Singleton	Jakob Misbach
8.1.2.3 Our Implementation	Jakob Misbach
8.1.2.4 Loading Rules	Jakob Misbach
8.1.2.5 An Attempt to Speed Up Loading Rules and Rulesets	Jakob Misbach
8.1.3 Graph Processing Implementation	Jakob Misbach
8.1.4 Load from File Implementation	Jakob Misbach
8.1.5 Image Conversion and Export to Image Implementation	Margaret Earnest
8.1.6 Undo/Redo Implementation	Anthony Vuolo
8.1.6.1 Edit Stack Handler	Anthony Vuolo
8.1.6.2 Undo/Redo Interface Elements	Anthony Vuolo
8.1.6.3 Undo/Redo Process	Anthony Vuolo
8.1.7 State Equation Implementation	Anthony Vuolo
8.1.7.1 Expression	Anthony Vuolo
8.1.7.2 Depth-First Search	Anthony Vuolo
8.1.7.3 Bond Graph Wrapper	Anthony Vuolo
8.1.7.4 Causal Wrapper	Anthony Vuolo
8.1.7.5 Substitution of Intermediate Equations	Anthony Vuolo
8.1.7.6 Domain-Specific Variable Substitution	Anthony Vuolo
8.1.7.7 Pseudocode	Anthony Vuolo
8.2 User Interface Implementation	Margaret Earnest
8.2.1 BoGL Web User Interface Control Flow Diagram	Margaret Earnest
8.2.2 User Interface Menus	Margaret Earnest
8.2.2.1 Top Menu Implementation	Margaret Earnest
8.2.2.2 Element Menu Implementation	Margaret Earnest
8.2.2.3 Modifier Menu Implementation	Margaret Earnest
8.2.3 SVG Graphs	Margaret Earnest
8.2.3.1 Generic Graph Display Implementation	Margaret Earnest
8.2.3.2 System Diagram Implementation	Margaret Earnest
8.2.3.3 Bond Graph Display Implementation	Margaret Earnest
8.2.3.4 Pan and Zoom Implementation	Margaret Earnest
8.2.3.5 Element Creation Implementation	Margaret Earnest
8.2.3.6 Edge Creation Implementation	Margaret Earnest
8.2.3.7 Clipboard Integration Implementation	Margaret Earnest
8.2.3.8 Delete Element of Edge Implementation	Margaret Earnest
8.2.3.9 Select Element or Edge Implementation	Margaret Earnest
8.2.3.10 Multi-Elements	Jakob Misbach
8.3 New Features	Jakob Misbach
8.3.1 URL Implementation	Margaret Earnest
8.3.2 State Equation UI Implementation	Margaret Earnest
8.3.3 Graph Drawing Implementation	Jakob Misbach
8.3.4 Tutorial Implementation	Jakob Misbach

8.4 Deployment	Jakob Misbach
9 Testing	Jakob Misbach
9.1 UI Testing	Margaret Earnest
9.2 Backend Testing	Anthony Vuolo
9.2.1 Automated Testing	Anthony Vuolo
9.2.2 Manual Testing	Anthony Vuolo
10 User Evaluations	Anthony Vuolo
10.1 User Evaluation Design	Anthony Vuolo
10.1.1 Goals and Requirements	Anthony Vuolo
10.1.2 Creating the Survey	Anthony Vuolo
10.1.3 Implementing the Survey	Anthony Vuolo
10.2 Analysis of User Feedback	Anthony Vuolo
10.2.1 Discussion of the Data	Anthony Vuolo
10.2.2 Quantitative Data Analysis	Anthony Vuolo
10.2.2.1 System Diagram Statistics	Anthony Vuolo
10.2.2.2 User Satisfaction with Features	Anthony Vuolo
10.2.2.3 Improvements Requested by Users	Anthony Vuolo
10.2.2.4 Bugs Discovered by Users	Anthony Vuolo
10.2.3 Qualitative Data Analysis	Anthony Vuolo
10.2.4 Data Analysis Summary	Anthony Vuolo
11 Conclusion	Jakob Misbach
11.1 Student Reflection	All
12 Future Work	Jakob Misbach
13 References	N/A
14 Appendices	N/A
Appendix A	All
Appendix B	N/A
Appendix C	All
Appendix D	Jakob Misbach

IV. Table of Contents

V. List of Figures

VI. List of Tables

1 Introduction

In 2016, a Major Qualifying Project (MQP) titled *Graph Grammar Based Automated Virtual Lab for Bond Graphs* at Worcester Polytechnic Institute (WPI) developed an application used for generating bond graphs (Grande & Mancini, 2016). This application was built to help Mechanical Engineering (ME) and Robotics Engineering (RBE) students taking ME 4322, Modeling and Analysis of Mechatronic Systems. The application helped students generate bond graphs that they could use to check their homework solutions. The MQP team created over 120 grammar rules which could handle systems within three energy domains. These domains were mechanical, electrical, and hydraulic systems. The team developed a backend which could generate and simplify multiple bond graphs. The user interface developed by the team, pictured in Figure 1, was less functional. The main feature which was missing was the ability to drag and drop components from a canvas on the screen to construct a system diagram. A system diagram is a method of representing the structure of a mechatronic system. It shows the interaction between various elements within this system. This addition in BoGL Desktop, the application created by the previous MQP team, made the system much more accessible to users since it made it easier for system diagrams to be constructed.

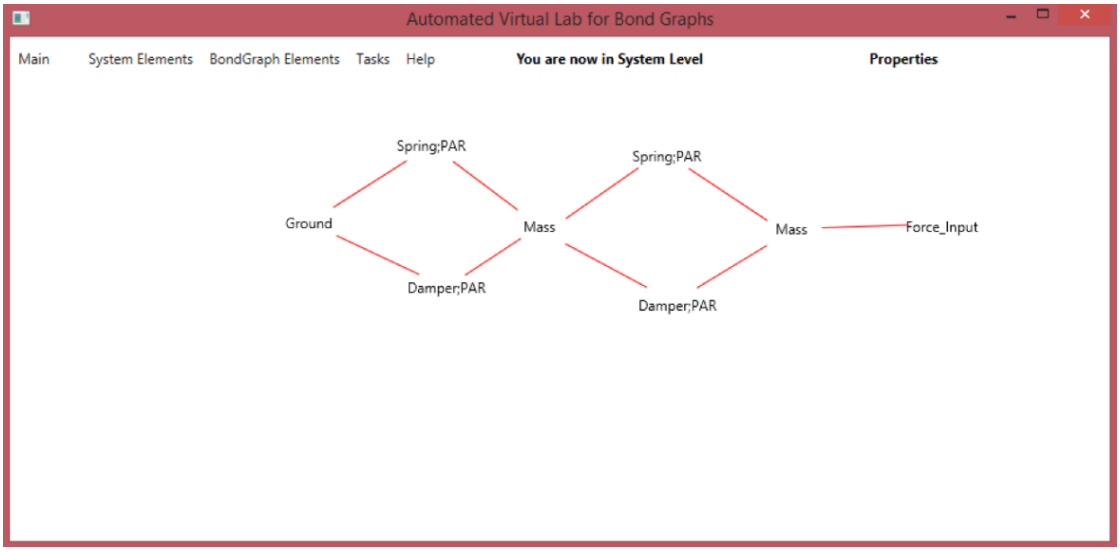


Figure 1: User Interface (UI) from the 2016 MQP.

A second MQP completed in 2020 titled *BoGL: An Application for Generating Bond Graphs* was completed with the goal of improving the user interface (Courville et al., 2020). This MQP created a more intuitive and streamlined UI and completed a user survey to test the effectiveness of the new interface. A clear menu was created with all elements which could be added to the system diagram. Additionally, the team made components of the UI more visible to the user including making the tabs easier to see, adding a clear zoom bar, and a menu to add modifiers to the system. The team also created a website (https://bogl.mech.website/) and a set of tutorials to make the application easier for students to download and understand.

The updated UI can be seen in Figure 2. This new UI includes an updated color scheme when compared to the UI in Figure 1. The biggest change was the ability for users to drag and drop elements of a system onto the canvas. This made it much easier to create system diagrams, and in turn, create bond graphs. Additionally, this new UI includes a menu to add elements to the canvas, a zoom menu, a menu for adding modifiers, and tabs which can be used to switch between the system diagram and the various bond graphs.

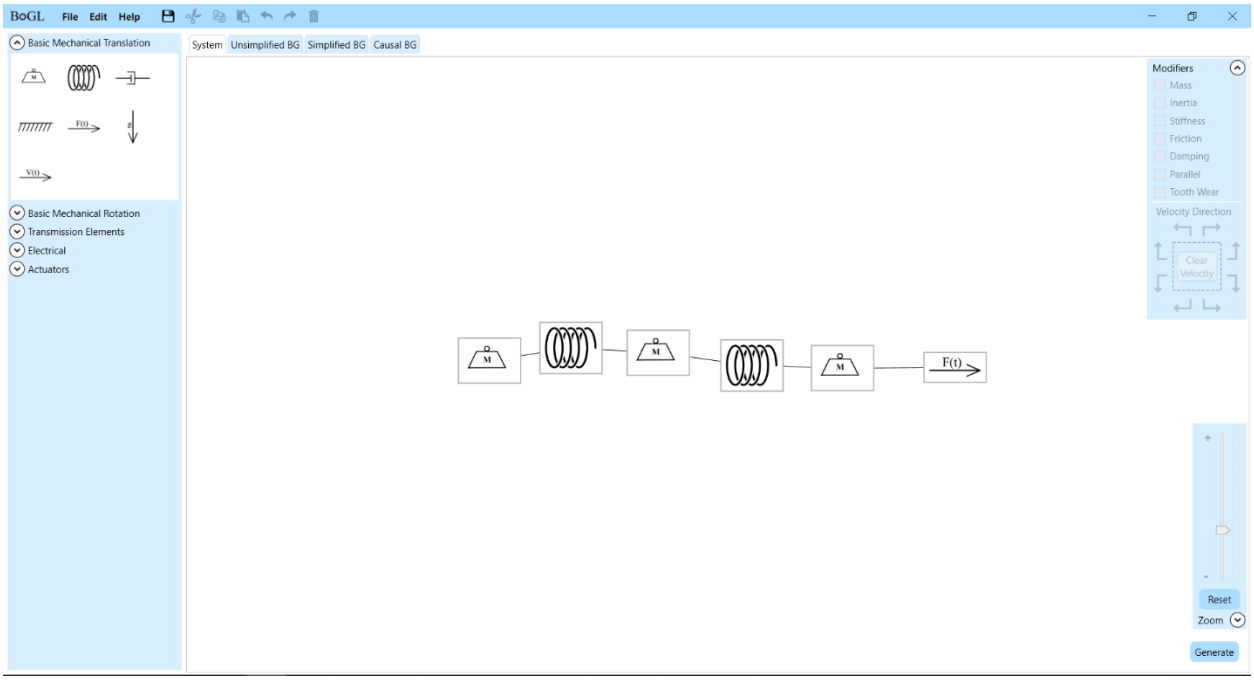


Figure 2: Updated UI from 2020 MQP.

The 2020 MQP noted some areas for improvement, which were either discovered during user testing or through tests which were conducted internally. The first of these issues is the need for an improved layout algorithm for the bond graphs. The algorithm the team used tended to place elements of the bond graph off the screen making it difficult for users to view all elements of the bond graph at once without rearranging its elements. The team also noted that their application did not have support for generating state equations or operating on elements from additional energy domains. Due to the framework the 2020 MQP had been developed in, it could only be run on Windows systems. In our report, we will highlight the increased compatibility of our system as it is web-based. This means that any user with a web browser will be able to access the system allowing for users who are not using Windows to access the system.

This MQP aims to increase accessibility to BoGL by creating a web application for modeling bond graphs; one which can be used by students instead of requiring them to download a desktop application. We also wanted to update aspects of BoGL Desktop to better fit a website environment, for example, by allowing users to share system diagrams using a Uniform Resource Locators (URL). This would allow users to easily share system diagrams with other BoGL Web, the system which we created, users. These links are easier to share than a .bogl file, which was used in the previous system, since a URL is just a string of text. Finally, we aimed to provide students with an in-app algorithm to generate state equations from a bond graph. By completing these tasks, we will increase the number of WPI students who can use BoGL Web as well as make it available to a larger audience.

This report is organized as follows: Chapter 2 describes the motivation for this project. Chapter 3 provides necessary background on Graphs, Bond Graphs, Mechanical Systems, State Equations, and relevant Major Qualifying Projects which have been previously completed. Chapter 4 provides an overview of project requirements. Chapter 5 provides an overview of important literature including a review of web frameworks, UI component libraries, and graph drawing. Chapter 6 provides an overview of the methodology we used in completing our project including choosing a UI component library, choosing a Tech Stack, testing Blazor's performance, UI prototyping methodology, and development methodology. Chapter 7 details how we designed various aspects of the project including, the Undo/Redo system, the state equation system, and the tutorial. Chapter 8 describes how we implemented all features including replicating existing features from BoGL Desktop and creating new features. Chapter 9 discusses the process we used to test the system. Chapter 10 includes details about user evaluations our

team conducted. Chapter 11 provides a conclusion on the entire project. Chapter 12 provides information on areas for future work.

2 Motivation

In this chapter we will discuss the motivation for creating the BoGL System as well as BoGL Web specifically. We begin with the high-level goals of the BoGL System, and then will discuss the goals of each major feature in BoGL.

The main goal of the BoGL system is to help students learn about bond graphs and check solutions to homework problems. While there is a lot of software on the market for working with mechatronic systems in many capacities, the options which include support for bond graphs are limited. Students are used to having software tools which they can use to augment their learning, and the main goal of BoGL is to fill this niche.

The system build during the 2020 MQP, BoGL Desktop, was created using a framework which limited it to only Windows based computers (Courville et al., 2020). This meant that, while students now had access to software which could generate bond graphs, they may have issues getting it to run. This meant that students who did not have access to Windows computers would either need to use a complex solution to get the software running, or not use the software at all. Since the main goal of BoGL is to create software to help all students, our team worked to translate BoGL Desktop into a new web framework. Since every computer has access to the internet through a web browser, this would mean that all users, with access to a modern web browser, would be able to access the system. This is a vast improvement over the previous system as it adds support for usings using both Mac and Linux systems.

BoGL Web also adds several new features to make BoGL more useful to students. It is often the case then, when working on a homework assignment, or other project, students and professors need to quickly share information about the system they are working on. In BoGL

Desktop, this was by using a .bogl file. This was a special file format created by previous projects which stored all information necessary about a system diagram and could be used to recreate the system on another machine. While this was a reasonable solution for sharing system diagrams on a desktop app, we added an additional method to BoGL Web. This addition was motivated by the need to quickly share system diagrams, and it being widespread practice to use URLs to share information on the internet. We created a method to encode system diagrams within URLs. This allows for a much faster way for users to share system diagrams. Since a URL is just a string of text as opposed to a file, it is much easier to share over systems like email, or other messaging services. The URL approach also makes it easy to submit system diagrams for homework, as a grader would just need to click on the link instead of having to download and open a file. This added efficiency was the main motivation for adding this feature.

A standard feature in most systems is a way to undo and redo actions which the user has made. BoGL Desktop includes this feature as well, however, the undo redo system in BoGL Desktop would only track one change at a time. This means that if the user for example added three elements to the system diagram, then wanted to undo adding the last two, this would not be possible. Instead, the user would either need to delete the extra elements or restart with a blank system diagram. We improve the Undo/Redo experience by allowing for more than just one action to be undone by the user. This is important because the user could be making many changes subsequently. If they find themselves having gone several steps down the wrong path, they may want to undo these steps instead of having to start over and recreate the system again.

In mechanical engineering, the motivation for constructing a bond graph is so that an engineer can easily derive state equations which model the dynamics of a system. While BoGL Desktop was able to generate a bond graph, and some work had been done to add generation of

state equations, this feature was not fully finished, and there was no user interface work done to support this feature. In BoGL Web, we add the ability to generate and display state equations which are generated from the Bond Graphs. This feature is yet to be fully implemented at the time of this report being published. However, we developed several interface mockups for their display. These were evaluated based on different metrics and a finalized design was chosen to implement. We also documented what portion of these mockups were implemented in Section 8.3.2. By adding this feature, students will also be able to check that they are getting the correct state equations as solutions to homework problems. Our team also plans to display the steps that need to be taken to derive the state equations. This means that, if a student does make a mistake, they will be able to look at the steps our system takes to derive the equation and figure out what leads to their issue.

One feature of BoGL Desktop which was important to learning how to use the system were a series of guides which were hosted online (https://bogl.mech.website/help/). These guides described the various parts of the system as well as providing information on how to perform various tasks within the system. While these guides were useful for inexperienced users learning how to use the system, there was nothing present within BoGL Desktop itself which a user could follow to help them learn about the application. We added an interactive tutorial to help users learn how to use the BoGL system. This tutorial would run inside of BoGL Web, allowing a new user to learn about the application without having to visit another website. The tutorial shows a user the important aspects of the website, giving them a quick way to learn how to use BoGL Web without leaving the application.

One consideration which was important for both BoGL Web and BoGL Desktop was how to place the elements of the bond graph after it had been generated. When the bond graph is initially

generated, it is just a list of names, with nothing associating these names to positions on the screen. To solve this problem, both systems employed a layout algorithm which determines where to place elements of the bond graph on the screen. The algorithm in BoGL Desktop would start at some element of the bond graph, and then procedurally place each other element of the graph, one by one, in relation to this initial element. In BoGL Web, we use a different algorithm which was developed by Eades, to find a layout which places elements which are connected closer together, and elements which are not connected farther apart (Di Battista, 1999). Our system also takes the bond graph, once we have found locations for all elements, and centers it on the canvas, which makes the entire bond graph visible to the user. This centering step is an improvement over the system which was used in BoGL Desktop as it would commonly place elements of the bond graph off screen.

In this section, we provided motivation for this project, and all the major features which we have added. In the next section we will provide important background information for our project including information on graphs, Bond Graphs, important mechanical engineering concepts, state equations, and previous work related to this project.

3 Background

In this section, we provide background on several important topics that were relevant to this project, such as bond graphs and the previous BoGL MQP (Courville et al., 2020).

3.1 Graphs

All definitions are taken from the Handbook of Combinatorics (Graham et al., 1995).

- A *graph* G is a nonempty set $V(\mathrm{G})$ of elements, called *vertices* or *nodes*, and a set $E(\mathrm{G})$ of elements called *edges*, together with a relation of *incidence* that associates with each edge two vertices, called its *endpoints*.
- An edge with identical endpoints is a *loop*.
- Two or more edges with the same pair of endpoints are *multiple edges*.
- The two endpoints of an edge are *joined* by the edge and *adjacent* to one another; adjacent vertices are also referred to as *neighbors*.

- A *simple graph* is one with neither loops nor multiple edges.
- A graph is *finite* if both its vertex set, and its edge set are finite.
- A graph is *bipartite* if its vertex set can be partitioned into subsets X and Y so that each edge has one endpoint in X and one endpoint in Y.
- The *degree* of a vertex v in a graph G is the number of edges of G incident with v, a loop counting as two edges.
- A *leaf* is a vertex of degree one.
- Let G and H be graphs. If $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$, the graph H is called a *subgraph* of G. This relationship is denoted by $H \subseteq G$.

- A subgraph I of G such that $V(I) = V(G)$ is said to *span* G and is classified as a *spanning subgraph*.
- A *walk, W,* in a graph G is defined as a sequence $W \coloneqq v_0, e_1, v_1, \dots, e_n, v_n$ where $v_0, v_1, \dots, v_n$ are vertices of G, $e_1, \dots, e_n$ are edges of G, and v_{i-1}, v_i are the endpoints of $e_i, 1 \leq i \leq n$. The vertex v_0 is the *tail* of W and the vertex v_n its *head*.
- The walk, *W,* is *open* if $v_0 \neq v_n$ and *closed* if $v_0 = v_n$
- The walk, W, is a *trail* if its edges $e_1, \dots, e_n$ are distinct.
- A walk, *W,* is a *path* if its vertices $v_0, \dots, v_n$ are distinct.
- A closed trail of positive length whose vertices (apart from its endpoints) are distinct is called a *circuit* or *cycle*.
- A graph is *connected* if any two of its vertices are connected by a path.
- A graph is *acyclic* if it contains no circuit.
- A *tree* is a connected acyclic graph.
- A *spanning tree* is a spanning subgraph which is a tree.
- A *directed graph*, or *digraph*, D is a graph G where each edge is assigned a direction, one endpoint being designed its *tail* and the other its *head*.
- Two or more edges with the same tail and head are *multiple edges*.
- A *strict digraph* is one with neither loops nor multiple edges.
- The *underlying graph* $G(D)$ of a digraph D is the graph obtained from D by ignoring the orientations of its edges.
- An *oriented graph* is a digraph whose underlying graph is simple.
- The *indegree* of v in D is the number of edges of D whose head is v.
- A *source* is a vertex of indegree zero.

- A *directed walk* in a digraph D is a sequence $W \coloneqq v_0, e_1, v_1, \dots, e_n, v_n$ where $v_0, v_1, \dots, v_n$ are vertices of D, $e_1, \dots, e_n$ are edges of D, and v_{i-1}, v_i are the *tail* and *head* of $e_i, 1 \leq i \leq n$, respectively. The vertex v_0 is the tail of W and the vertex v_n its head. *Directed trail, directed path,* and *directed circuit* are defined analogously.
- A digraph is *acyclic* if it contains no directed circuit.
- A *graph labeling* is an assignment of labels to the vertices, edges, or both, subject to certain conditions.

3.2 Bond Graphs

Bond graphs were introduced by Henry Paynter to describe dynamical mechanical or electrical systems (Broenink, 1999). A node in a bond graph is called an *element* and an edge a *bond*. An element can be one of the following: an R-element; a source of flow or effort; an I-storage element; a C-storage element; a transformer; a gyrator; a 0-junction; or 1-junction. We describe each type of element in detail in the following sections.

3.2.1 List of Elements

This list below provides the different types of elements and the notation used for each of them in Bond Graphs.

- *Source of effort*, denoted by 'Se.'
- *Source of flow*, denoted by 'Sf.'
- *R-element*, denoted by 'R.'
- *I-storage element*, denoted by 'I.'
- *C-storage element*, denoted by 'C.'

- *Transformer*, denoted by ‘TF.’
- *Gyrator*, denoted by ‘GY.’
- *1-junction*, denoted by ‘1’.
- *0-junction*, denoted by ‘0’.

Exactly what each of these types represent physically will be presented below.

3.2.2 Bonds

Bonds are denoted by a line with a half-arrow at the head. The direction of the half-arrow is called *flow direction*. The second direction is distinct from flow direction; it is called *causality* and is denoted by a short perpendicular bar on the bond, or *causal stroke*. Each bond is labeled with an *effort* and a *flow*. This effort is always labeled on the left or on top of the bond, and the flow below or to the right of the bond. The flow direction half-arrow will always be on the same side of the bond as this flow label. The horizontal orientation of a bond is shown in Figure 3 and the vertical orientation in Figure 4.

In the *translational* mechanical domain (*translational domain)*, the effort is called *force* F and the flow is called *velocity* v. In the *rotational* mechanical domain (or *rotational domain*), the effort is called *torque* τ and the flow is called *angular velocity* ω. In the *electrical* domain, the effort is called *voltage* V and the flow is called *current* i.

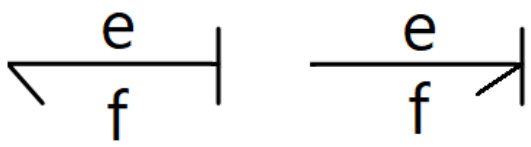


Figure 3: The two possible orientations of flow direction (half-arrow) and causal stroke on a horizontal bond.

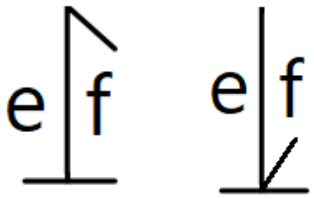


Figure 4: The two possible orientations of flow direction (half-arrow) and causal stroke on a vertical bond.

3.2.3 Sources of Flow and Effort

Sources of effort are leaf nodes – more specifically, source nodes – denoted ‘Se.’ In the translational domain, this is a *source of force* (such as gravity, or someone pushing on a mass) and is labeled ‘Se:F’. In the rotational domain, this is a *source of torque* (such as someone turning a crankshaft) and is labeled ‘Se:τ’. In the electrical domain, this is a *source of voltage* (such as a battery or wall outlet) and is labeled ‘Se:V’ (see Figure 5). A *source of flow* is a source node denoted ‘Sf.’ In the translational domain, this is a *source of velocity* and is labeled ‘Sf:v’. In the rotational domain, this is a *source of angular velocity* and is labeled ‘Sf:ω’. In the electrical domain, this is a *source of current* and is labeled ‘Sf:i’. The flow direction of any bond incident to a source will always point away from that source.

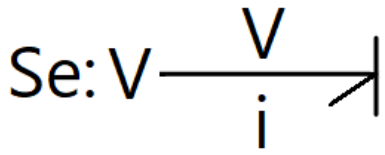


Figure 5: A source of effort in the electrical domain, such as a battery.

3.2.4 R-Elements

R-elements (*resistive* element, or *free energy dissipator*) are leaf nodes. In the mechanical domains, an R-element is a *damper*, such as a dashpot. In the translational domain, it is denoted in the bond graph as "R:b" and has an associated translational *damping constant b*. In the rotational domain, the damping constant is *D*, and the element is labeled 'R:D.' In the electrical domain, the R-element is a *resistor* with a *resistance R*, and the element is labeled 'R:R.' The flow direction of the incident bond will always point toward the R-element.

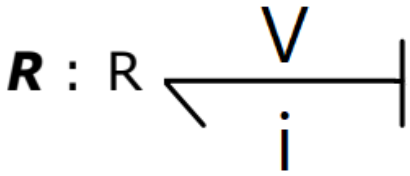


Figure 6: An R-element (resistor) in the electrical domain.

3.2.5 Storage Elements

I-elements (or *inertial* elements) are indicated by 'I' and are incident to exactly one bond. In the mechanical translational domain, an I-element has an associated *mass m* and is labeled

'I:m' (see Figure 7). In the rotational domain, an I-element has a *mass moment of inertia J* and is labeled 'I:J'. In the electrical domain, an I-element is an *inductor* with *inductance L* and is labeled 'I:L'.

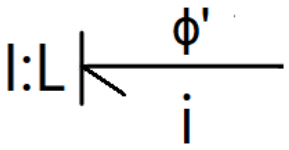


Figure 7: An I-element (inductor) in the electrical domain.

C-storage elements (or C-elements) are indicated by 'C' and are incident to exactly one bond. In the mechanical domains, a C-element is a *spring* (see Figure 8). In the translational domain, this spring has a *stiffness* or *spring constant k* and is labeled 'C:k'. In the rotational domain, the spring has a *torsional spring constant* K_t and is labeled 'C:K_t'. In the electrical domain, a C-element is a *capacitor* with *capacitance* C and is labeled 'C:C'.

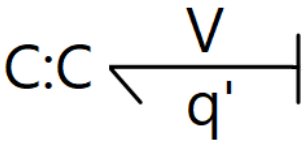


Figure 8: A C-element (capacitor) in the electrical domain.

3.2.6 Transformers

Transformer elements are labeled with a 'TF' in the bond graph. A transformer in the translational domain will be labeled 'TF:R'. A transformer in the rotational domain (such as a

lever or *gear*) will be labeled ‘TF:n’. An *electrical transformer* in the electrical domain is written as ‘TF:D’.

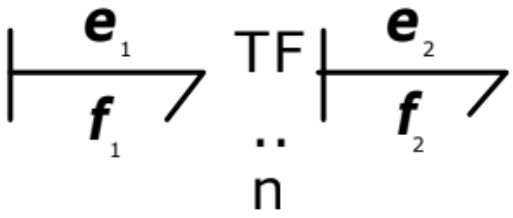


Figure 9: A transformer whose incident bonds have the causal stroke and half-arrow located on opposite ends.

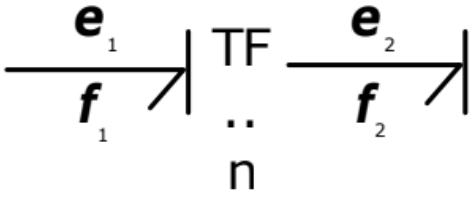


Figure 10: A transformer whose incident bonds have the causal stroke and half-arrow located on the same end.

A transformer is incident to exactly two bonds, one with a causal stroke adjacent to it and one with the causal stroke on the other side of the bond. These two bonds always belong to the same domain (e.g., electrical). One bond will have flow direction toward it, and one will have flow direction away. The two possible orientations of flow direction and causal stroke are shown in Figure 9 and Figure 10.

3.2.7 Gyrators

Gyrator elements (such as some kinds of *pump*) are labeled as ‘GY:r’ in the bond graph. Gyrators are incident to exactly two bonds that belong to different domains. The causal strokes of those two bonds will either both point away from the gyrator (see Figure 11) or both point towards the gyrator (see Figure 12). The gyrator has an associated *gyrator ratio* written as r.

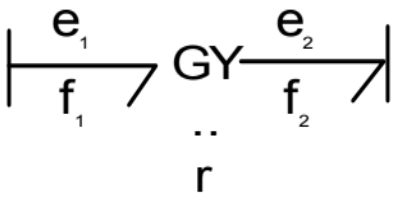


Figure 11: A gyrator with incident bonds whose causal strokes are not adjacent to the gyrator.

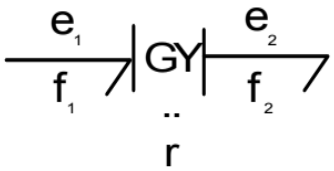


Figure 12: A gyrator with incident bonds whose causal strokes are adjacent to the gyrator.

3.2.8 1-Junctions and 0-Junctions

Junctions are incident to two or more bonds, all of which belong to the same domain. Every bond incident to a *1-junction* has the same flow as the 1-junction (see Figure 13). In the translational domain, a 1-junction is labeled ‘1 (v)’, or ‘1 (v_i)’ with some subscript i if there is more than one such junction. In the rotational domain, it is labeled ‘1 (ω)’ or ‘1 (ω_i)’. In the

electrical domain, it is labeled '1 (i)' or '1 (i_i)'. Also, the sum of the efforts of all bonds with inward-facing flow direction (*incoming* effort) is equal to the sum of the efforts of all bonds with outward-facing flow direction (*outgoing* effort).

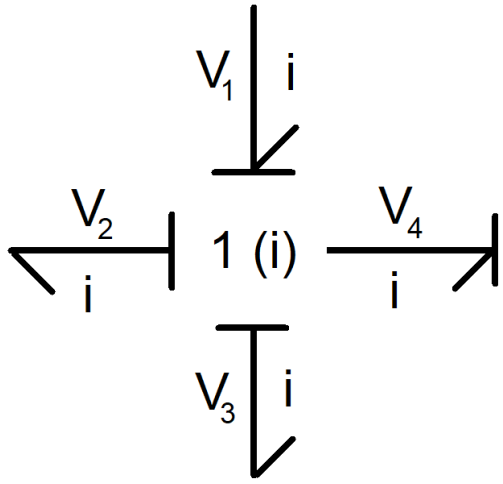


Figure 13: A 1-junction in the electrical domain.

Conversely, the efforts of all bonds incident to a *0-junction* are equal (see Figure 14). This effort is assigned to the 0-junction. Analogously to the 1-junction, the net incoming flow for a 0-junction is equal to the outgoing flow. In the translational domain, a 0-junction is labeled '0 (F)' or '0 (F_i)' with a subscript as before. In the rotational domain, it is labeled '0 (τ)' or '0 (τ_i)'. In the electrical domain, it is labeled '0 (V)' or '0 (V_i)'.

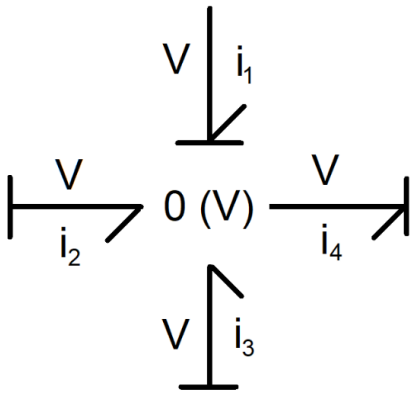


Figure 14: A 0-junction in the electrical domain.

3.2.9 Domain-Specific Effort-Flow Equations

We provide a list of all constants and equations specific to the translational mechanical, rotational mechanical, and electrical domains in a bond graph. “Displacement” and “momentum” are used for the conserved quantities for I- and C-elements in any domain; the apostrophe notation (e.g., x') indicates the derivative of that conserved quantity with respect to time. For the equations associated with transformers, the effort and flow e_1 and f_1 belong to the bond with flow direction pointed into the transformer. For the equations associated with 0- and 1-junctions, i, j are indices of arbitrary incident bonds.

Table 1: A list of all domain-specific bond graph symbols.

Symbol	*Translational*	*Rotational*	*Electrical*
Flow	v	ω	i
Effort	F	τ	V
R-element	b	D	R
I-element	m	I	L
C-element	k	κ	C
Transformer	D	R	T
Gyrator	r	r	r
Momentum	p	L	Φ
Displacement	x	θ	q

Table 2: A list of all domain-specific effort-flow equations for each element.

Element	*Translational*	*Rotational*	*Electrical*
R-element	$F = bv$	$\tau = D\omega$	$V = Ri$
I-element	$v = p/m$ $F = p'$	$\omega = L/I$ $\tau = L'$	$i = \Phi/L$ $V = \Phi'$
C-element	$v = x'$ $F = kx$	$\omega = \theta'$ $\tau = \kappa\theta$	$i = q'$ $V = q/C$
Transformer	$v_1 = v_2/D$ $F_1 = DF_2$	$\omega_1 = \omega_2/R$ $\tau_1 = R\tau_2$	$i_1 = i_2/T$ $V_1 = TV_2$
0-junction	$\sum v_i = 0$ $F_i = F_j$	$\sum \omega_i = 0$ $\tau_i = \tau_j$	$\sum i_i = 0$ $V_i = V_j$
1-junction	$v_i = v_j$ $\sum F_i = 0$	$\omega_i = \omega_j$ $\sum \tau_i = 0$	$i_i = i_j$ $\sum V_i = 0$

We can simplify the gyrator equations to $e_1 = rf_2$ and $e_2 = rf_1$, with e_1, f_1 and e_2, f_2 being the effort and flow on each of the two incident bonds. Since these equations are symmetric, the choice of which bond has which effort-flow pair is arbitrary.

3.3 Kirchhoff's Laws

In this section, we explain Kirchhoff's Laws and how we can use them to compute current in an electrical network. Kirchhoff's Laws describe the rules governing effort and flow in bond graph junctions but deal only with simple electrical networks. We introduce these laws and how to use them as a sort of a warmup exercise in doing calculations with bond graphs. Before delving into Kirchhoff's Laws, we will briefly discuss river networks to explain the origin of some circuit terminology.

3.3.1 River Networks

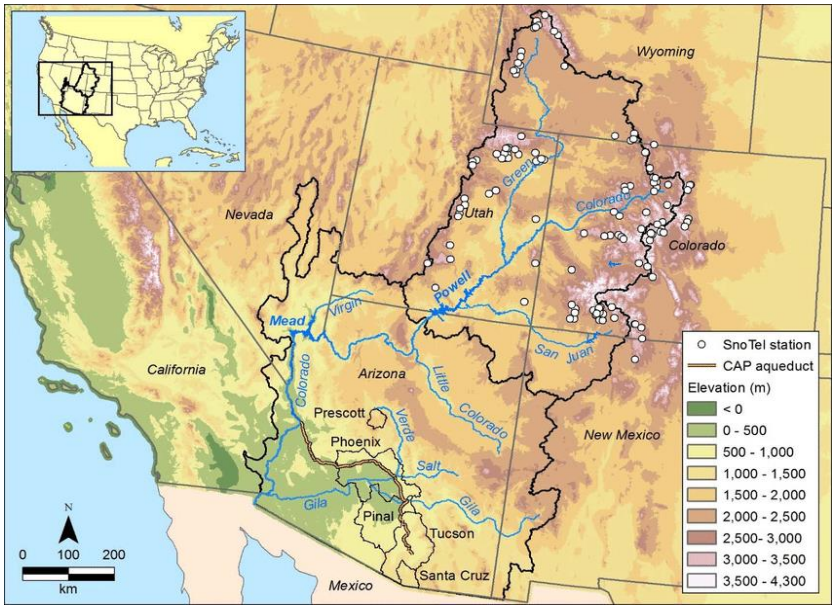


Figure 15: A topographical map of the Colorado River in the western United States (Scanlon et al., 2015).

The Colorado River, pictured in Figure 15, starts in northern Colorado, high in the Rocky Mountains. The river travels southwest from there, cutting through Utah and forming part of the

border between Arizona, Nevada, and California before emptying into the Sea of Cortez. Many *tributary* rivers can be shown in the figure, splitting off to form miniature river networks. Ground elevation in the Rockies is typically between 7000 and 14000 feet above sea level.

Figure 16: An intersection in the Colorado River (Squillace & Harper, 2022).

The flow of water in a river is called *current*. Current is measured as the change in water volume over a period, such as gallons per second, which pass through a specific point in the river. Consider the currents of different tributaries at a specified intersection in the Colorado River, such as the one shown in Figure 16. If the current entering the intersection is greater than the current leaving it, the extra water would accumulate at the intersection until it floods. If the incoming current is too low, the intersection would eventually dry up. For such an intersection to remain at a stable water level, the incoming current must be equal to the outgoing current. This property of river networks and the word "current" are shared with electrical networks, as described in the next section.

3.3.2 Kirchhoff's Circuit Laws

Kirchhoff's Laws were established by Gustav Kirchhoff to describe electrical networks. The flow of electricity is called *current*, a nod to the analogous concept of flow in river networks. One can also determine the properties of a circuit by observing the *voltage* between two points. Kirchhoff's laws defining current, and voltage are as follows:

1. Current law: *at a junction, the incoming current must equal the outgoing current.*
2. Voltage law: *around a closed loop, the voltage drop must be zero* (Wilson, 2010).

Consider that in a bond graph, the electrical domain has current as a flow and voltage as an effort. At a 0-junction, the incoming flow is equal to the outgoing flow. We can then say that a junction in an electrical network is equivalent to a 0-junction in the bond graph for an electrical network, as they are both described by Kirchhoff's first circuit law. At a 1-junction, the incoming effort equals the outgoing effort; we can then say that the net change in effort is zero. Then a closed loop in an electrical network is equivalent to a 1-junction in the electrical domain.

3.3.3 Finding Current and Voltage in an Electrical Network

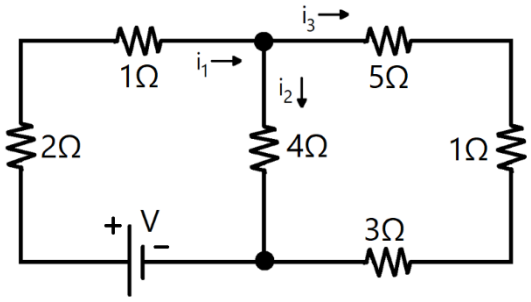


Figure 17*: An example electrical network.*

We are given the electrical circuit shown in *Figure* 17. The jagged line symbols indicate resistors. Each resistor is labeled with its respective resistance; for example, the leftmost resistor has a resistance of 2 ohms (2Ω). Given this information and a voltage of $V = 75$ volts across the battery, we want to find the current through each resistor. We can do so using Kirchhoff's Laws above. There are three different currents to compute in the network, given as i_1, i_2, and i_3 in the diagram and arbitrarily assigned a current direction.

To use Kirchhoff's voltage law to compute the voltage drops, we first establish a relationship between current and voltage in terms of resistance. We can do so using Ohm's Law, which states that the drop in voltage V across a resistor with resistance R and current i through the resistor satisfies $V = iR$. This is shown visually in Figure 18. In this diagram, the voltage drop occurs from point a to point b and includes a resistor of R ohms.

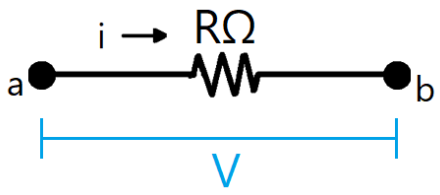


Figure 18: A diagram showing a voltage drop over a resistor in an electrical network.

Knowing this, we can now generate a system of equations for the network in Figure 17. Since there are three currents to compute, we must have a system of three equations, each generated by one of Kirchhoff's Laws. We start by computing the number of equations derived from the voltage law, so we need to know the number of closed loops in the diagram. To obtain this information, we can complete the following procedure:

1. Find a spanning tree. For our given network, we will let each horizontal or vertical line designate an edge in the graph, and each intersection of two edges be a vertex.
2. For each edge in the graph not in the tree, temporarily add that edge back to the diagram and record the cycle created as a cycle equation.

We start with the spanning tree of the network, shown in Figure 19.

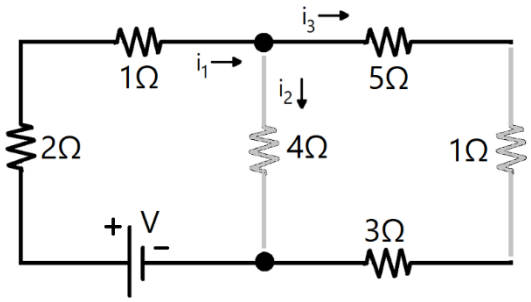


Figure 19: The minimum spanning tree of the electrical network.

Since we have removed two edges, we will have two cycle equations. To generate the first, we add the edge with the 4Ω resistor back to the diagram – the result is shown in Figure 20 – and record the cycle as a cycle equation.

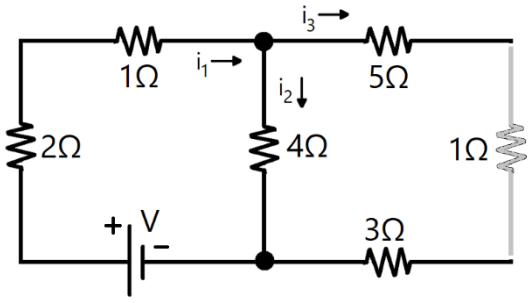


Figure 20: The electrical network, with only the first cycle given.

Using Kirchhoff's voltage law, we can encode the voltages around the cycle in an equation as follows:

$$75 - 2i_1 - i_1 - 4i_2 = 0$$

We then replace the other edge we initially removed to form the spanning tree, resulting in the diagram shown in Figure 21.

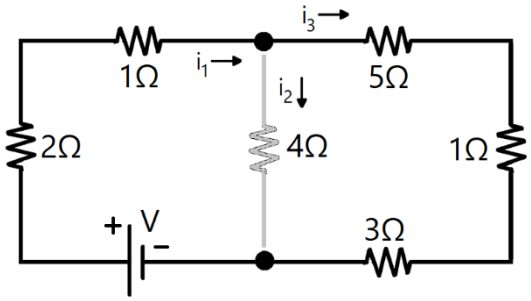


Figure 21: The electrical network, with only the second cycle given.

The equation we derive from this cycle is as follows:

$$75 - 2i_1 - i_1 - 5i_3 - i_3 - 3i_3 = 0$$

Since we have exhausted all cycles in the graph, any remaining equations must be formed by using Kirchhoff's current law to determine current values at junctions. Since we have three unknown currents and two cycle equations, there remains one equation to generate at a junction. Without loss of generality, we choose the junction adjacent to the voltage source and record the incoming and outgoing currents as a single equation:

$$i_2 + i_3 = i_1$$

We then set up our three equations as a linear system to solve for the currents.

$$\begin{cases} 75 - 2i_1 - i_1 - 4i_2 = 0 \\ 75 - 2i_1 - i_1 - 5i_3 - i_3 - 3i_3 = 0 \\ i_2 + i_3 = i_1 \end{cases} \rightarrow \begin{cases} 3i_1 + 4i_2 = 75 \\ 3i_1 + 9i_3 = 75 \\ i_1 - i_2 - i_3 = 0 \end{cases} \rightarrow \begin{bmatrix} i_1 \\ i_2 \\ i_3 \end{bmatrix} = \begin{bmatrix} 13 \\ 9 \\ 4 \end{bmatrix}$$

Then $i_1 = 13$ amps, $i_2 = 9$ amps, and $i_3 = 4$ amps are the currents through each part of the circuit.

Kirchhoff's Laws are an efficient way of obtaining the current at distinct locations throughout an electrical network. The mathematical relationship behind the

3.4 Bond Graphs of Mechanical and Electrical Systems

Kirchhoff's Laws apply only to electrical networks. However, we can extend the same concept to mechanical systems and other domains using bond graphs. A bond graph of a thermodynamic system is a way of representing a flow of power between different components in the system. The bonds themselves represent that power transfer and show how that power is distributed amongst the elements. In fact, power is the product of the effort and flow in every domain.

Elements in a bond graph are categorized based on how they affect this power flow in a system. These types of elements are as described in section 3.2, but we now justify Paynter's method of determining which elements in which domains affect power in the same way. *Sources of effort* and *sources of flow*, such as batteries or wall outlets, provide a constant effort or flow to the system from an outside source. *R-elements*, such as dampers and resistors, dissipate energy out of the system. *I-elements*, such as masses or inductors, accumulate flow as a form of energy storage. *C-elements*, such as springs or capacitors, accumulate effort for a similar effect. *Transformers*, such as gears or electrical transformers, change the amount of flow passing through a part of the system without changing the amount of power. *Gyrators*, such as motors or pumps, change the effort-flow ratio between different domains. *Junctions* have no physical

analogue, but they show when groups of elements in a system all have the same flow or effort . These relationships are described mathematically in Table 2 in section 3.2.9.

The question we ask is, given a physical or electrical system with a set of elements, what are the flow and effort on each element at a given time? We can answer that question by finding the flow and effort on every bond in the associated bond graph. Every element in the bond graph is given a constant. However, bond graphs are often used to model dynamical systems, requiring that some part of the modeled mechanical or electrical network is changing over time. Such values – energy variables – are the *generalized momentum* stored by I-elements and *generalized displacement* stored by C-elements. These I- and C-elements are called *states*, a reference to the fact that the energy variables represent the state of the system and can change over time. We can use the effort-flow relationships for each element to construct ordinary differential *state equations* for each energy variable. This state equation represents the value of the energy variable at a given time, and the set of all state equations for a bond graph can be used to compute the effort and flow on any bond. We will demonstrate in the next section how to generate state equations.

3.5 Generation of State Equations

We now perform the following procedure to derive state equations: labeling efforts and flows, establishing initial equations, and deriving final equations. We start with the bond graph in Figure 22, which models a translational damped mass-spring system. Note that none of the bonds have flow or effort labels; we will be filling those in with their final values.

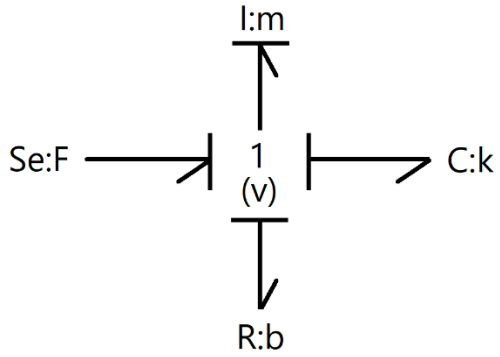


Figure 22: The bond graph we will be using as an example for generating state equations

We first identify the elements with states and their energy variables. We first observe that an I-element has a *generalized momentum* energy variable and that our given I-element is labeled 'I:m'. From this, we deduce that the element exists in the translational domain, since the 'm' indicates a *mass*. We conclude that the energy variable is a *linear momentum* p, and that the effort on the I-element (a force, F_m) then satisfies $F_m = p'$. We then label the effort on the bond incident to the I-element as p'.

Likewise, we recall that each C-element has a *generalized displacement* energy variable and that our given C-element is labeled 'C:k'. We infer that this element also exists in the translational domain, since 'k' indicates a *linear spring constant*. We conclude that this energy variable is a *linear displacement* x, and that the flow on the C-element (a velocity, v_k) satisfies $v_k = x'$. We then label the flow on the bond incident to the C-element as x'. This leaves us with the bond graph in Figure 23.

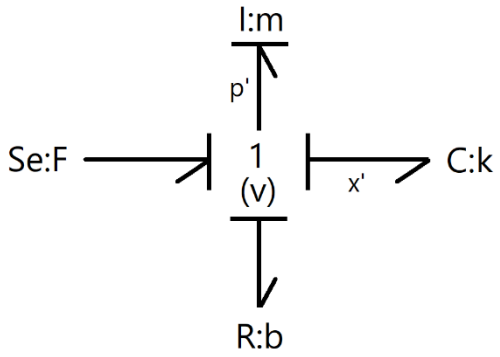


Figure 23: The bond graph with energy variables labeled.

Since we have identified an energy variable for every state, we can now label the rest of the flows and efforts. We first observe that all bonds are incident to a 1-junction with assigned flow 'v'. Then the flows for all these bonds – except for the one we have already labeled x' - can be labeled v. Since a source of effort provides a constant effort to the 1-junction, we label the effort on the bond incident to the source of force in Figure 23 as F. We do the same to the R-element (damper) with F_R and the C-element with F_C. The result is the bond graph in Figure 24.

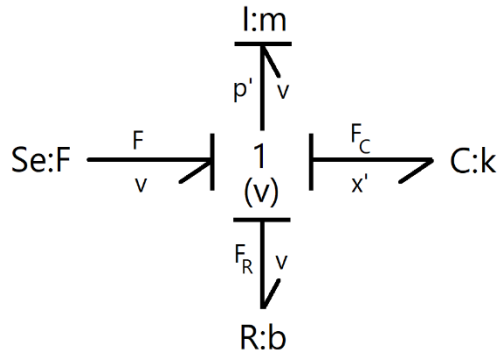


Figure 24: The bond graph with all flows and efforts labeled.

We can now collect a list of initial equations. We can generate these equations by using the effort-flow relationships from Table 2 along with variables in the bond graph in Figure 24. Table 3 has a list of the initial equations for this bond graph.

Table 3: A list of all initial equations, paired with the element from which the relationship was derived.

Element	*Equation(s)*
1-junction	$p' = F - F_C - F_R$ $x' = v$
I-element	$v = p/m$
C-element	$F_C = kx$
R-element	$F_R = bv$

We can then substitute the variables in the 1-junction equations with expressions in the equations from other elements. For p', we substitute F_C and F_R for the expressions on the right side of the C-element and R-element equations, respectively. For x', we substitute v for the expression on the right side of the I-element equation. Our final equations are provided in Table 4. Notice that the final equations are written using only energy variables and the constants associated with each element.

Table 4: The final state equations for the bond graph

Energy Variable	*Differential State Equation*
p	$p' = F - kx - \frac{b}{m}p$
x	$x' = \frac{1}{m}p$

3.6 BoGL MQP

The goal of BoGL Desktop, the previous iteration of this project, was to engineer an application that could generate bond graphs from user-constructed system diagrams. Similar to its predecessor from (Grande & Mancini, 2016), it would allow users to create system diagrams and generate bond graphs. In addition to improving upon the user interface of this older project, the 2020 project group planned to update specific algorithms in the backend for improved performance, including a speed-optimized depth-first search function. The team aimed to add a layout algorithm that would create positions for the elements in a graph to minimize the amount of overlap between components or bonds. Another extension they planned was a classroom feature that would allow homework submissions to be passed from "students" to "teachers" in the system (Courville et al., 2020).

For interface upgrades, the BoGL Desktop team added several usability features to their project. According to their report, their application would allow a user to be able to learn how to add elements from different domains to the canvas. One of the primary goals noted in the BoGL Desktop paper was making the application easier to use. This was done by making improvements such as reducing the number of actions it took to perform a given task. The team also made tasks such as moving elements on the canvas and adding edges between components easier for users. Another feature they wanted to improve was a rudimentary save history, just enough to allow the user to take advantage of an undo-redo system. According to their requirements, the user should also be able to reset the interface and start constructing a bond graph or system diagram from scratch (Courville et al., 2020).

To produce ideas and survey the current market for bond graph simulators, the BoGL team analyzed one competitor program, "20-sim." This application can model bond graphs and

icon diagrams – their system diagram equivalent – for electrical, mechanical, thermal, and hydraulic systems. Its interface consists of a canvas with an alignment grid, and its primary menu is a panel of component options listed on the left side of the screen. Instead of using text to denote menu options, 20-sim opts for icons with images of the different elements (Courville et al., 2020). A 20-sim user must construct bond graphs manually instead of generating them from icon diagrams. The user must then instruct the application to "inspect" the bond graph, at which point it can display initial and final state equations in a separate window. The 20-sim interface is shown in Figure 25.

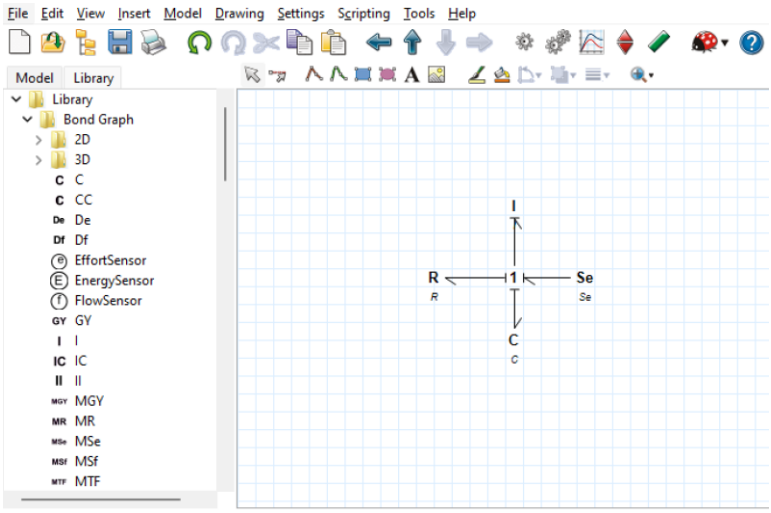


Figure 25: A bond graph as displayed in the 20-sim interface.

The BoGL team completed user tests to determine whether they had successfully improved the interface. The team asked test subjects to replicate a system diagram and then provide some feedback about how easy it was to complete various subtasks. They were asked to indicate the time it took to complete each task and then rank their time spent on the test on a scale of one to five. The team used a one-to-five ranking system to gather how easy tasks were

for their users. To judge the interface, the team asked for users to provide a more general assessment of how complex they felt the process was for adding modifiers and diagram elements. For bond graphs, the team had each test subject load a system diagram, from a .bogl file, into BoGL Desktop and convert into a bond graph. For these evaluations, the team asked only for the generation time in seconds and whether the given bond graph matched one they computed in advance (Courville et al., 2020).

User tests conducted partway through development revealed a series of problems test subjects found with the current interface and functionality. About one-third of users experienced issues when attempting to pan or zoom the graph canvas. While most users were able to successfully operate the visual controls, some respondents needed more time to discover how to use them. Others wished for a canvas that could rotate to display the system diagram at different angles. Some users were unable to distinguish certain graph components as displayed. For example, various kinds of spring icons in mechanical systems appeared too alike to differentiate, as all springs looked like the one in Figure 26. In addition, some users had trouble figuring out how to use velocity elements and modifiers. Other comments indicated that the size of the "Generate" button was too small. Forty-two percent of respondents indicated that they were unsatisfied with the placement of the elements of the generated bond graph in the canvas because they often overlapped, obscuring some of the labels. Respondents wanted an algorithm that would rearrange elements in the canvas to increase readability (Courville et al., 2020).

Figure 26: The system diagram spring icon from BoGL Desktop.

Based on the above user feedback, the BoGL team incorporated several improvements into the final BoGL release. They first implemented a standardized color scheme. Features with similar functionality were grouped together in submenus on the main screen. Within the drag-and-drop interface, they included a visual hierarchy sorted by system type that emphasized commonly used components (see Figure 27). The team programmed the interface to allow the user to manually place new graph elements, simplified tasks such as modifier editing, and optimized procedures overall for system diagram creation. They added a system that provides feedback when a user tries to generate an invalid bond graph, in the form of error messages. Some additional features included updates to the save system; the team added readability and simplified upload and download procedures for BoGL save files (Courville et al., 2020).

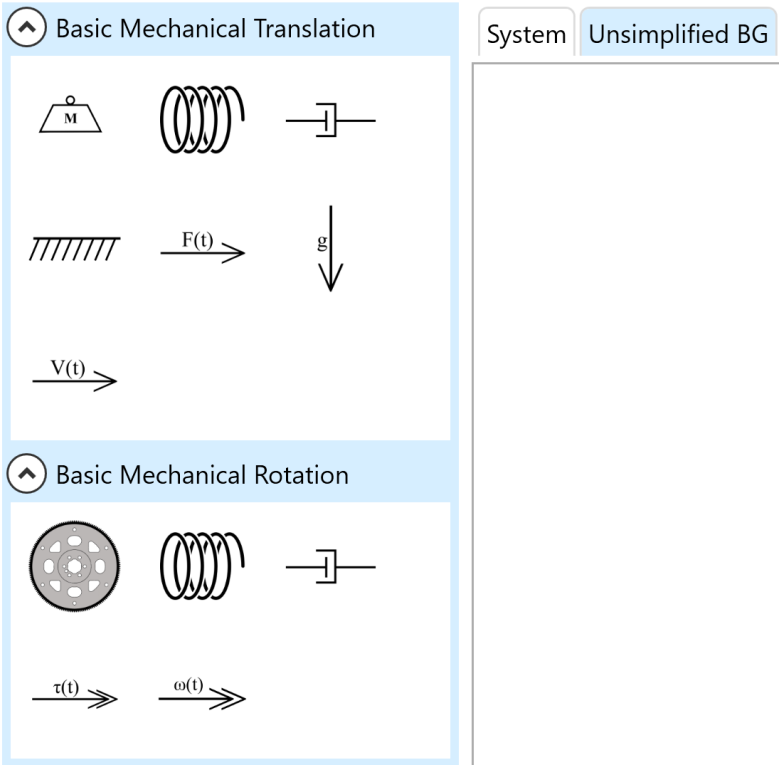


Figure 27: The hierarchical side menu of BoGL Desktop.

Based on results from final user surveys, the previous team outlined a list of requested features and fixes for future projects. Some of these suggestions involved visual enhancement such as font size control. One request was for a background snap grid. This feature would allow the user to align different components to make the system diagram aesthetically pleasing; some responses recommended that an algorithm arrange system diagram elements in this grid automatically. Some users wanted the application to link certain bond graphs with system diagrams, allowing the user to automatically edit both just by changing one. Users also wanted more functionality in bond graph generation, citing limitations when constructing the graphs. One such complaint was about the lack of specific feedback when the user attempts to generate an invalid bond graph. Another user noted that BoGL should, in addition to generating a bond

graph from a system diagram, be able to generate a system diagram from a bond graph (Courville et al, 2020).

4 Project Requirements

In this section, we describe the list of requirements which our team compiled for this project. These requirements include ensuring that the users can complete all the same tasks as in BoGL Desktop, and that BoGL Web matches BoGL Desktop in terms of computational efficiency.

In the initial planning phase for BoGL Web, we developed a list of requirements which would need to be met for BoGL Web to be successful. We start with a ranked list of these requirements and then describe each one in more detail in Table 5. We also include a way to validate the completion of each.

Ranking:

1. The need will be met by the system design.
2. Need that is a promising idea, but not necessary for basic system operation and capability.
3. Need that is recognized but not critical or essential to demonstrate system operation.

Table 5: BoGL Web Requirements.

Need	**Validation**	**Priority**
Users can construct system diagrams	User tests that require drawing diagrams	1
Have a layout algorithm	User tests and unit tests	1
Users can convert system diagrams to bond graphs	User tests that convert system diagrams to bond graphs and compare them with premade results	1
UI matches BoGL Windows application	Visual confirmation	1
Use BoGL Windows' grammar rules	Unit tests that check web bond graph results against Windows results	1
React to changes in screen size	Unit tests and user tests	1
Match BoGL Windows' computational efficiency	Measure the speed with which Windows BoGL converts graphs and test the same graphs on the web version	2
Exceed BoGL Windows' computational efficiency	Measure the speed with which Windows BoGL converts graphs and test the same graphs on the web version	3
Graphs can be exported to BoGL Desktop	Unit tests	2
Graphs can be imported from BoGL Desktop	Unit tests	2
Users can have persistence between sessions	Unit tests and user tests	3
Users can translate bond graphs to state equations	Unit tests	3

The next few paragraphs will expand on all the requirements marked with a priority of one in Table 5. The first of these requirements was that users should be able to construct System Diagrams. Without the ability to construct System Diagrams, no other parts of the application would be able to function. To allow users to construct System Diagrams, several components

needed to be added. The first of these is a Canvas that elements of the System Diagram could be placed on. This is also where users would be able to add connections, or edges, between elements. We would also need to replicate the method that BoGL Desktop used to allow users to create edges. An explanation of this process can be found in section 6.5.4. Finally, we would need to replicate the side menu in BoGL Desktop which allowed users to select elements to add to the Canvas.

The second requirement we had was the users would be able to convert System Diagrams into Bond Graphs. This feature was the main feature of BoGL Desktop, so it was mandatory for BoGL Web. To add this feature, we had to first find a way to load all the rules and rulesets which are used by GraphSynth to convert the System Diagram to a Bond Graph. Since BoGL Desktop ran on the desktop, it was simple to just read files from the file system. Running in the browser requires a different approach because we cannot guarantee that the user has all the rules and rulesets downloaded. Because of this, we must take a different approach, which is discussed in section 8.1.2.4. Being able to convert System Diagrams also requires that we be able to utilize GraphSynth. This could either be in the form it is in the 2020 MQP, or an altered form which can perform all necessary functions. Without some version of GraphSynth, we would not be able to apply the rules and rulesets to the System Diagram to generate a Bond Graph.

The next requirement for BoGL Web was to have a user interface which matched the user interface of BoGL Desktop. The MQP completed in 2020 was completed to improve the user interface of the original application created in 2016. Because of this, we felt that there were few or no areas which could be improved upon. Therefore, copying most user interface elements was a requirement. We would go on to make several small modifications to make the application

better suited to running in a browser. However, the bulk of the UI styling, design, and placement of components would remain the same.

The next requirement was that we match the computational efficiency of BoGL Desktop to a reasonable degree. Running in the browser presents more restrictions than running on the desktop in terms of computational power available. Our goal was to ensure that bond graphs were still being generated within a reasonable amount of time, ideally in under a few seconds. This requirement means that the algorithms we use must be efficient; as the size of the bond graphs increases, our aim is to use an algorithm whose runtime increases as a polynomial function of that size.

The next requirement was that we create a layout algorithm. As mentioned in the Motivation section of this report, BoGL Desktop created layouts of bond graphs such that most elements were placed off screen. We needed to create a layout algorithm which would create a visually appealing layout of the bond graph as well as have some way to ensure that all elements of the bond graph would appear on screen when a tab was first entered.

The next requirement was that System Diagrams are exportable as a URL. This was a new feature in BoGL Web, but one which had been highly requested by our advisors. This would allow users to transfer System Diagrams between each other quickly. To implement this feature, we needed to create an algorithm to encode System Diagrams in some form that could fit within the restrictions of a URL (see section 8.3.1). Additionally, we needed a way to compress the information in the URL to increase the number of elements and edges which could be stored in the URL.

The final requirement for BoGL Web was that System Diagrams created be backwards compatible to BoGL Desktop, and System Diagrams created in BoGL Web be able to be loaded

in BoGL Desktop. This required that BoGL Web be able to match the .bogl file format used in BoGL Desktop.

BoGL Web was able to fulfill all these requirements in addition to making further improvements to BoGL Desktop.

5 Literature Review

Before starting this project, our team made many considerations in terms of choosing the tech stack, UI framework, and the algorithms we would use. To obtain background information on these topics we completed a literature review. We will discuss information for many sources including books, articles, and documentation.

5.1 ASP.NET

We started by researching ASP.NET. When completing this research, we discovered two frameworks: ASP.NET and ASP.NET Core. ASP.NET is the original framework which was Windows only. ASP.NET Core was created to provide a cross platform version of ASP.NET. We will refer to ASP.NET Core as ASP.NET for the remainder of this section as the Windows only framework is not relevant to this project. ASP.NET is an open-source web framework that was created by Microsoft. The framework extends .NET which is a platform made up of many different tools, programming languages, and libraries that can aid in building applications. The .NET platform includes the C#, F#, and Visual Basic programming languages, libraries for working with many diverse types of data, and tools for working in a variety of environments. The ASP.NET extension of .NET incorporates tools necessary for creating web applications. Some additions developers get when using ASP.NET include a framework for handling web requests in C# and F#. ASP.NET also includes a technology called Razor. Razor allows developers to create web pages using C# and HTML. A Razor file has an HTML section and a C# section which gives developers the ability to create dynamic web pages (web pages that can change based on user input) using C#. ASP.NET also includes tools that will allow developers to

implement authentication systems with ease. There are libraries and templates that will help with multi-factor authentication and authenticating users through external websites including Google and Twitter.

ASP.NET backends are written using C#, F#, or Visual Basic and, since ASP.NET extends .NET, developers can use many of the libraries and packages that they would use for their other .NET applications. Since .NET has been around for many years at this point and is widely used, there are many existing tools that developers would be able to use directly in their ASP.NET projects (*.NET and .NET Core Official Support Policy*, n.d.).

There are several reasons why a developer would use ASP.NET over other web technologies. One of these reasons is the many languages that are supposed when developing using ASP.NET. As mentioned before, language support includes C#, F#, and Visual Basic, all of which have their benefits, but the ability for developers to choose is an advantage. ASP.NET is also built on Windows making configuration in a Windows environment simple. Other frameworks are not as easy to configure in Windows environments and must be installed and configured separately. As mentioned earlier, ASP.NET is popular, meaning that there are many libraries, tools, and tutorials easily accessible online. C# is also compiled which leads to performance increases over other, interpreted languages. Interpreted languages must be read every time the code is executed, and include PHP, JavaScript, and Ruby. Compiled languages run simultaneously. Using a compiled language also makes detecting bugs easier. For example, if a method's name is refactored, it must be changed everywhere it appears. In an interpreted language, there are no checks for this prior to users testing the application. In a compiled language, there will be compile time errors which indicate that the necessary changes have not yet been made.

Software cost is also another concern when it comes to web applications. There are often fees associated with frameworks, and the development environment needed to create applications with those frameworks. ASP.NET is both open source and uses Visual Studio. Being open-source means that the software is, by definition, available without fees. Visual Studio is also generally provided for free if the application is not being used commercially.

5.1.1 Angular and ASP.NET

Angular is a web framework developed and maintained by Google. It supports many commonly used technologies and design patterns such as AJAX, HTTP, and dependency injection. It is written in TypeScript and is known for clean code and good error handling. The structure of the framework creates a separation between the UI itself and the logic that controls it (*Angular Vs React*, 2021). One of the main benefits of Angular is its component-based architecture. This approach breaks up the application into individual components that are easy to test and reuse. It also helps developers narrow down where issues are coming from (Gazta, 2021; *The Good and the Bad of Angular Development*, 2022). Angular components are one of the three types of Angular directives, with structural and attribute being the other directive types. Component directives define the appearance and behavior of components, structural directives add or remove DOM elements, and attribute directives apply styles or Cascading Style Sheet (CSS) classes to an element. Angular uses directives to live update the DOM based on variable changes (Kumar, 2022).

Besides its component architecture, Angular offers a wide variety of other features that streamline application development. For example, Angular is platform agnostic and can run on any system. Its framework provides a thorough blueprint for a basic application, with additional

libraries for more complex programs. Developers can easily manage which systems and interfaces control their application through dependency injection (*Angular Vs React*, 2021). Angular's dependency injection defines injectable resources using an Injectable object and allows dependencies to be injected at the application, model, and component levels. If a dependency has already been injected at a given level, the existing dependency will be used instead of creating a new object (*Angular - Understanding Dependency Injection*, n.d.). Angular also supports both one and two-way data binding, which allows a programmer to choose whether UI changes will affect an underlying component. Front-end files are updated by an incremental DOM, which stores the state of the interface in terms of the edits made since a starting state. With the exception of some additional allocated memory, this update method uses less storage space (*Angular Vs React*, 2021).

One of the major disadvantages of Angular is its steep learning curve. Angular documentation can be unclear, and no clear manual exists that describes the entire framework. In general, the scope of components can be hard to debug due to routing. Additionally, third party components and libraries can be difficult to integrate. Finally, Angular can be quite verbose and may require a number of files to define a singular component (*Angular Vs React*, 2021; Gazta, 2021).

5.1.2 React and ASP.NET

React is an open-source JavaScript library which is used and supported by Facebook. The library became open-source in 2013 and popularized component-based architecture.

React is a popular and widely used library for a variety of reasons. The first is that it is very straightforward and easy to learn. There is also less boilerplate code required to start out in

React, making it faster to develop. React utilizes a one-directional binding system that stabilizes the code by preventing UI changes from affecting relevant backend code. React also has a large community, which means there are many forums where questions are answered frequently and a wide variety of public libraries.

React's ability to adapt to program changes stems from its component-based architecture. This architecture divides React into multiple components that control various parts of the library. As a result, react code is modular and reusable for different programs. If developing a mobile version of a computer application, the user can easily switch out the view they are using. Component-based architecture also contains code blocks with similar purposes, so while debugging, the programmer, is less likely to break unrelated algorithms. Overall, this structure reduces unnecessary error and development time (*Angular Vs React*, 2021).

The main downside of React is that it is primarily used for front-end, client-side applications and does not have the required language compatibility to use BoGL Desktop's C# code. This means that we cannot reuse the past MQP's code. It also requires us to use a server which increases the long-term cost of the application.

5.2 OpenSilver

OpenSilver is an "open-source reimplementation of Silverlight that runs on current browsers via WebAssembly," created by Userware (Userware, n.d.-a). This technology was first delivered in the Technical Preview stage with its later 1.0 version being released in October of 2021. OpenSilver was created as a replacement for Microsoft Silverlight, a platform that supported applications for many companies before reaching the end of life. The goal of

developing OpenSilver was to provide an easy alternative to rewriting any SilverLight-based software.

There are a few positives when it comes to using OpenSilver. First, OpenSilver is compatible with all major browsers that support WebAssembly (Userware, n.d.-a). Additionally, OpenSilver supports C#. This means that we would not have to rewrite as much code since BoGL Desktop and GraphSynth are written in C#.

Some elements of OpenSilver make it difficult to use for new projects. Because it is new, OpenSilver offers only a small support community. A software team encountering errors in their code will struggle to find debugging suggestions from forums or manuals. Even though it has been released, OpenSilver still needs development. There are features which have not yet been added, as seen by the roadmap, including support for many JavaScript based libraries (Userware, n.d.-b). This means that we could run into issues where a feature we need does not exist and there are no libraries to add that feature due to limited community support.

5.3 Blazor

Blazor is a web framework that was developed by Microsoft and initially released in September of 2019. The framework is open source, and allows developers to create web applications using C# and HTML for free (*Blazor University - Blazor Hosting Models*, n.d.). Blazor can be run in a client-server architecture, known as Blazor Server, or as a client-side application using Blazor WebAssembly. We will get into the benefits and disadvantages of these two technologies in the next section, but it is important to first understand the major differences between these two frameworks. Blazor WebAssembly uses WebAssembly to carry out client-side computation. This framework runs entirely on the client side, meaning that the only

communication with the server is the initial download of the application. Blazor Server maintains a connection with the server throughout its entire life. This connection is created through SignalR, a communication framework that can be used with ASP.NET applications like Blazor Server (*Real-Time ASP.NET with SignalR | .NET*, n.d.).

5.3.1 Blazor WebAssembly

Blazor WebAssembly is a version of Blazor which runs entirely in the browser. In other words, once the application has been downloaded from the server, no further communication is needed. This gives two distinct advantages over an application which needs ongoing communication between the client and the server. First, since the server is only needed when the user is downloading the webpage, load on the server is significantly reduced and does not scale significantly with more users. Second, the lack of server processing allows the app to run offline if the user loses their internet connection. The user just needs to download the application and they can continue using BoGL Web exactly as they would with the internet.

Making the website a Progressive Web App (PWA) is the key to enabling user downloads. A PWA is a web application that handles loss of internet gracefully, for example, by notifying the user when requests cannot be sent or caching HTTP requests and sending them when connection is reestablished (*Introduction to Progressive Web Apps - Progressive Web Apps (PWAs) | MDN*, n.d.). While this functionality can be difficult to implement, the only hard requirements for a browser to recognize an application as a PWA is to serve the site on an HTTPS domain and to have a web manifest file, an icon for the downloaded app, and a service worker registered (which is only required by Chrome for Android) (*How to Make PWAs Installable - Progressive Web Apps (PWAs) | MDN*, n.d.). Once these requirements are met, the

browser will give the user the option to download the website. Since Blazor WebAssembly does all its computation client side, there are no server interactions to handle and thus meeting these requirements makes the application a fully functional PWA.

Although Blazor WebAssembly has many benefits, it also presents some difficulties. First, since the app will run entirely client side, all necessary libraries and files need to be downloaded. This means that startup times can sometimes be long, and there will be a large footprint on the client machine. To counter the slow startup, clients can cache data and experience faster startup times after their first visit to the website if the cache is not cleared. Blazor WebAssembly is also slower than Blazor Server for certain types of computations, due to working with the limited resources of a browser instead of the larger resources of a server. Another downside is that Blazor Web assembly currently only supports single threads. This means that algorithms which utilize concurrency to speed up computation are of no benefit in Blazor WebAssembly. Also, while Blazor WebAssembly is supported by most modern browsers like Chrome, Edge, Firefox, and Safari, older versions of these browsers do not support WebAssembly, creating potential for some users to have trouble using the app. Despite this, multithreading is a feature which is currently being added to experimental builds of Blazor WebAssembly. Finally, Blazor WebAssembly is not pre-rendered, meaning that search engines cannot view the content of the website, which prevents it from appearing reliably in search engines (*Blazor University - Blazor Hosting Models*, n.d.).

5.3.2 Blazor Server

Blazor Server is a version of Blazor that runs with a client and a server. There are several benefits to this paradigm. One benefit is that the server can pre-render HTML content and send it

to the client, allowing for search engines to index the website and decreasing start-up time. Start-up time will also be decreased since the client-side code is much smaller than Blazor WebAssembly's client, which includes server functionality. Additionally, Blazor Server will work with older browsers since it does not use WebAssembly here, with its only requirement being the ability to run JavaScript (*Blazor University - Blazor Hosting Models*, n.d.). Since we have access to a server in this paradigm, there is also the possibility for performance gains for larger computations due to the increased computational power of the server.

There are however several disadvantages to running Blazor Server. First, Blazor Server sets up sessions in memory for each client. This means that resources need to be allocated for every concurrent client and there is no possibility to perform load-balancing since clients are tied to a single server (*Blazor University - Blazor Hosting Models*, n.d.). Second, Blazor Server cannot run fully client side like Blazor WebAssembly can, meaning that any offline experience with Blazor Server will be missing functionality.

5.4 UI Component Libraries

UI component libraries are collections of pre-made UI elements. These components represent self-contained UI units such as buttons, inputs, or banners. They are made with pure HTML, CSS, and JavaScript, and if applicable, they encapsulate all functionality that an element should have. This could include expanding and collapsing, displaying user input, changing color on interaction, or many other things. These components can be used as they are or may be modified to better match a project's desired UI requirements. Most UI component libraries are tied to a particular web application framework like React or Angular. Many aspects of a project should be taken into consideration when deciding whether to use a UI component library,

including deadline, amount and type of front-end work, project size, developer experience, and budget.

UI component libraries have many benefits. First, they help design a user interface that is easy to use and robust, since library components are carefully designed and tested. The libraries themselves are well-organized and have helpful documentation. UI component libraries also improve collaboration between UI designers and other developers. Since the components are premade, developers can have a higher-level conversation about UI design without getting into the specifics of how particular elements should be programmed. When multiple projects want to use the same UI components, component libraries help reduce redundancy across different sections of an application. They also lead to more consistency as all people on a team or project draw from the same component library.

UI component libraries are well suited for certain types of teams and projects. When a team lacks sufficient financial resources, time, or skilled designers, they should rely on a UI component library instead of building their own. Teams that are building a proof of concept also benefit from using component libraries, since the focus in a proof of concept should be functionality, not precise UI design. Using a component library lets a developer piece together a UI together quickly and easily, regardless of design skill.

Despite their benefits, UI component libraries are not suitable for every project. Plenty of companies, such as Uber, Amazon, and Google, create their own UI component libraries. Doing so gives their websites and apps a unique feel and allows them to build their own recognizable visual brands. One downside of using a UI component library is that it produces a generic look, since many applications and websites use the same UI component libraries. UI component libraries also lock the developer into a specific paradigm, making momentous changes difficult

and potentially requiring a switch to a different UI component library. There is also a space concern; UI component libraries often take up more space than custom component libraries since the library includes many components, especially if a product does not use most of the components in the library. Given these issues, projects with a large scope, such as a plan to create multiple branded applications, and a sizable budget, have incentive to design their own UI component libraries instead of using prebuilt ones.

5.4.1 UI Component Library Options

As previously mentioned, most UI component libraries can only be used with certain tech stacks, so the availability of UI component libraries may affect the choice of tech stack and vice versa. Some UI component libraries, like Ant Design, are popular enough to have been implemented in multiple tech stacks, with each tech stack having varying levels of functionality. Other UI component libraries, like BlazorStrap, React Bootstrap, and UI Bootstrap below, share the same CSS framework, in this case Bootstrap, but do not have identical UI components. Finally, some UI component libraries, like MatBlazor, Material UI, and Angular Material, all follow the same UI design principle, such as Material Design, but do not have the same components. When we evaluated tech stacks to determine which suited the needs of BoGL Web best, we considered which UI components were used in the BoGL Desktop design and what functionality each component should have (see section 6.2). We used UI component recommendation articles for Blazor (Malavasi, n.d.), React (De Moor, 2022; Hughes, 2021; Varkki, n.d.), and Angular (Shah, n.d.; Sharma, 2022) to determine which frameworks are most widely liked and used. All the UI component libraries we discuss in the following sections are free for non-commercial use and open source.

5.4.1.1 All Tech Stacks

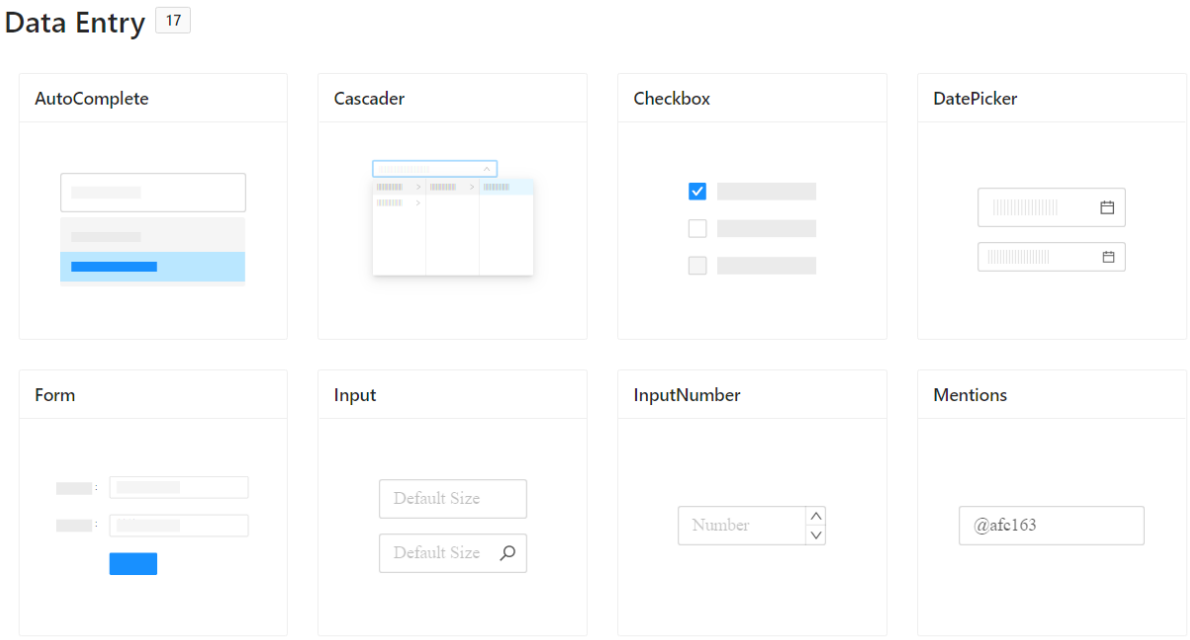


Figure 28: Ant Design components.

The only UI component library that we looked at which is supported by all tech stacks is Ant Design. Ant Design, shown in Figure 28, was originally created by Alibaba and is associated with their affiliate company Ant Financial. The UI library's main design values are to be Natural, Certain, Meaningful, and Growing (*Introduction - Ant Design*, n.d.). The free, open-source library was originally made for React, but versions for Angular, Vue, and Blazor have been created that support most of React's Ant Design functionality.

5.4.1.2 Blazor and OpenSilver

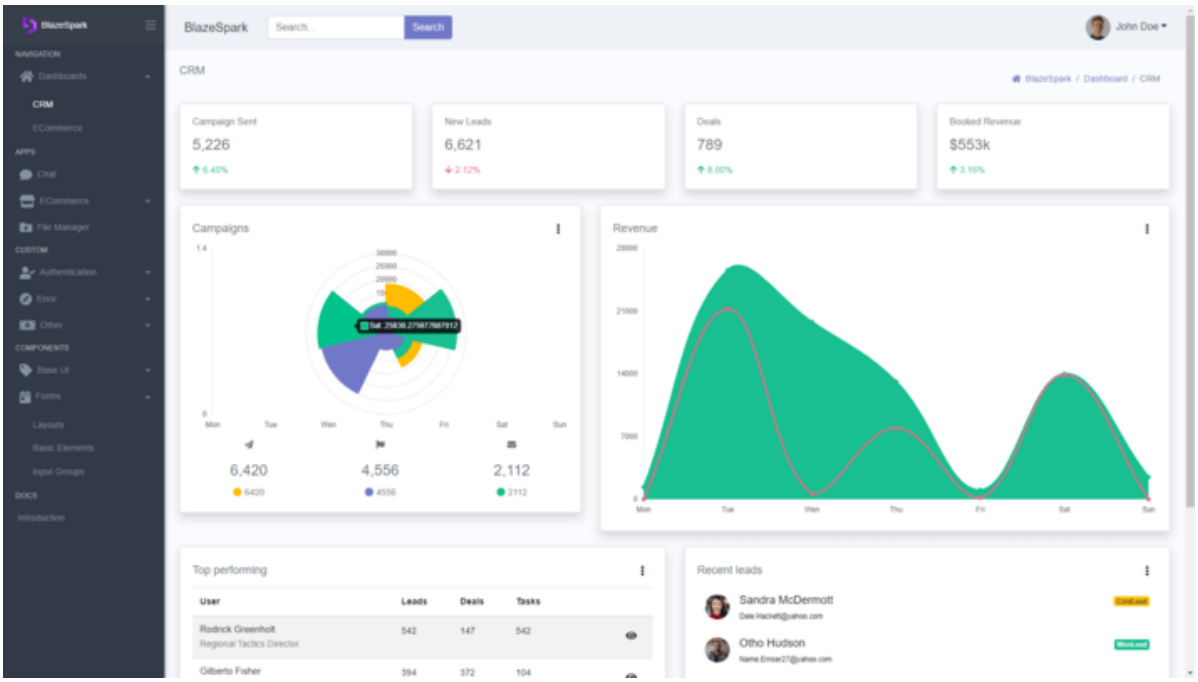


Figure 29: Blazorise example page (ComponentSource - Blazorise, n.d.).

Blazor and OpenSilver support the same list of UI component libraries, since OpenSilver is built on Blazor. These libraries tend to be less developed than the React and Angular options since Blazor is a much newer tech stack. One well-known Blazor UI component library is Blazorise, shown in Figure 29. Blazorise supports CSS styling from Bootstrap 4, Bootstrap 5, Material UI, Ant Design, and Bulma, but does not exactly replicate the appearance of the elements in their respective libraries (*Blazorise Documentation*, n.d.).

MatBlazor (*MatBlazor - Material Design Components for Blazor*, n.d.) is another Blazor-supported UI component library that brings Material Design (*Material Design*, n.d.-a) components to Blazor. Material Design is a system of guidelines, components, and tools that is inspired by the textures of the physical world, takes on principles of print design, and uses

motion to assign meaning. The principles and components were originally designed by Google but are used by many projects through several UI component libraries. MatBlazor is developed and maintained by an independent team with no company ties.

BlazorStrap is a Blazor UI component library that implements BootStrap elements for Blazor. The library supports Bootstrap 4 and 5 (*BlazorStrap 5*, n.d.). Bootstrap, originally designed by Twitter, is the most widely used CSS framework. It is known for enabling responsive and mobile-first layouts, making images responsive, and having high customizability (Ouellette, n.d.). BlazorStrap is supported by a community of volunteer developers and ensures compatibility between Bootstrap and Blazor. It supports a variety of CSS style packages (*BlazorStrap 5*, n.d.).

5.4.1.3 React

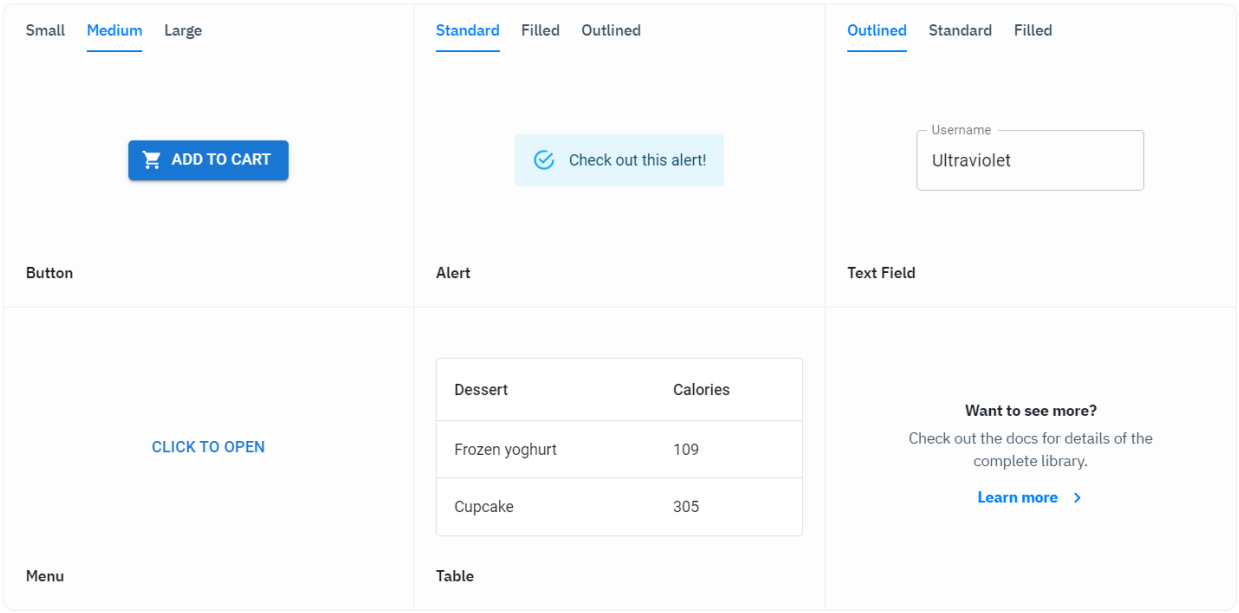


Figure 30: Material UI components (Overview - MUI Base, *n.d.).*

Material UI (MUI) (*Overview - MUI Base*, n.d.), shown in Figure 30, is a popular React UI component library that, like MatBlazor, follows Material Design principles. There are multiple free versions in MUI Core, including Material UI, which comes packaged with default styles, Joy UI, a sister library that focuses on beauty and does not adhere to Material Design (*Overview - Joy UI*, n.d.), MUI Base, which is Material UI without the pre-packaged styles, and MUI System, which helps easily customize the other three libraries. MUI is open-source and used by huge companies like Amazon, Spotify, and Netflix (*Overview - Material UI*, n.d.).

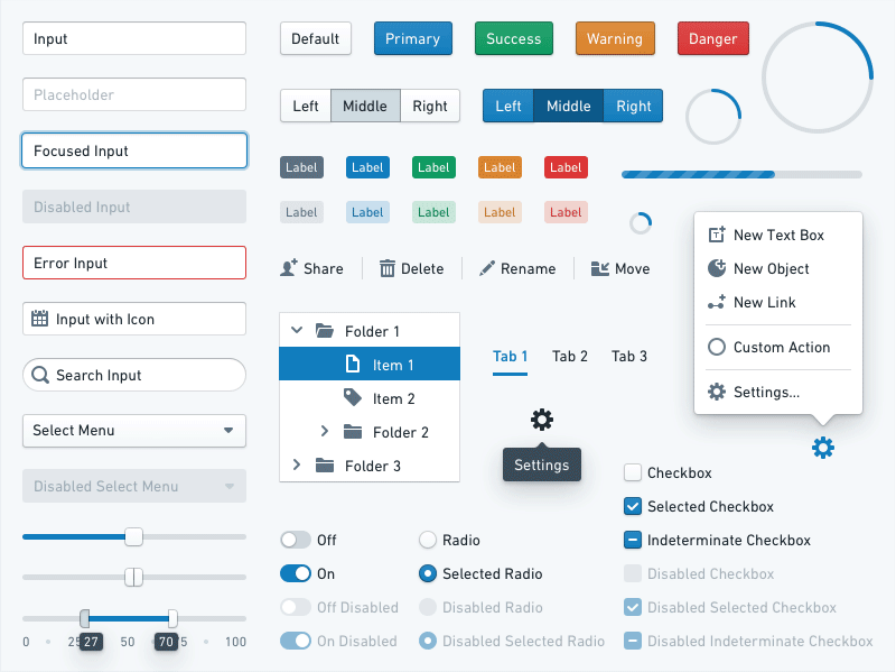


*Figure 31: Blueprint UI components (*Palantir / Projects / Blueprint, *n.d.).*

Blueprint UI (Palantir, 2020), shown in Figure 31, is a React UI toolkit that specializes in displaying complicated, data-dense interfaces optimized for desktop applications. The UI component library was created by Palantir, a big data analytics company, for use in their own products. Its guiding principles are composition, accessibility, and developer experience. The

library uses TypeScript to describe its components with a static typing system and components leverage inheritance and interfaces.

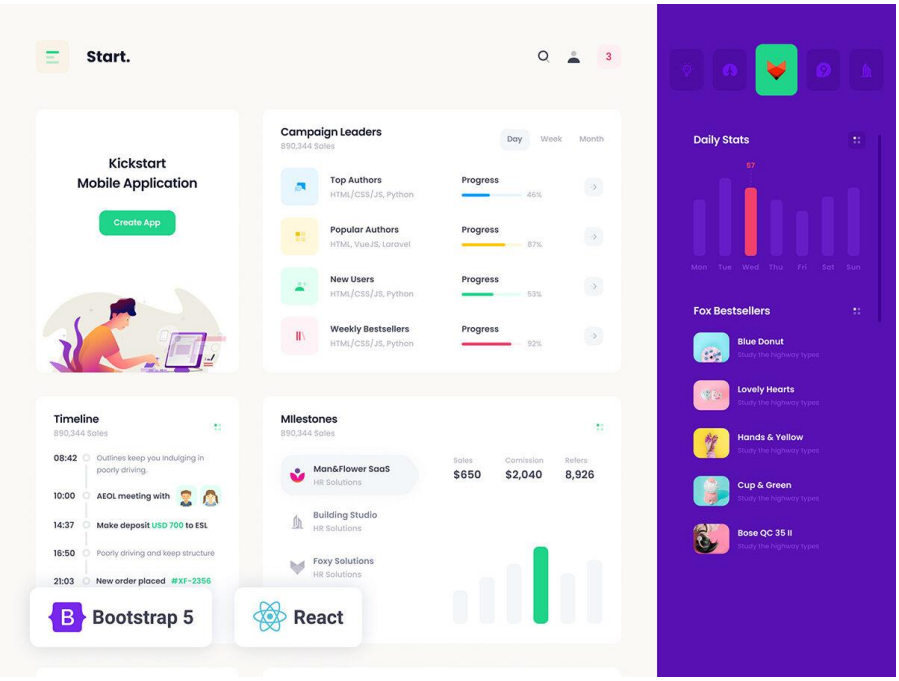


*Figure 32: React Bootstrap components (*Start – React Bootstrap 5 Admin Dashboard Theme, *n.d.).*

Like BlazorStrap, React Bootstrap (*React-Bootstrap*, n.d.) is a UI component library that uses Bootstrap styling. This library, pictured in Figure 32, replaces Bootstrap JavaScript and rebuilds each element from scratch in React to avoid JavaScript dependencies such as jQuery. The library highlights built-in accessibility, which was not possible to the same extent as earlier Bootstrap libraries. It is maintained by an independent developer team and keeps up to date with the latest React updates.

5.4.1.4 Angular

UI Bootstrap (*Angular Directives for Bootstrap*, n.d.) is an Angular UI component library that implements Bootstrap CSS. It is written in AngularJS by the AngularUI team, who support a variety of Angular UI library projects. UI Bootstrap is written to only use native AngularJS directives, avoiding the dependency of Bootstrap's JavaScript code and jQuery. The library comes in normal (unminified) and compressed (minified) forms, both available with and without UI templates.

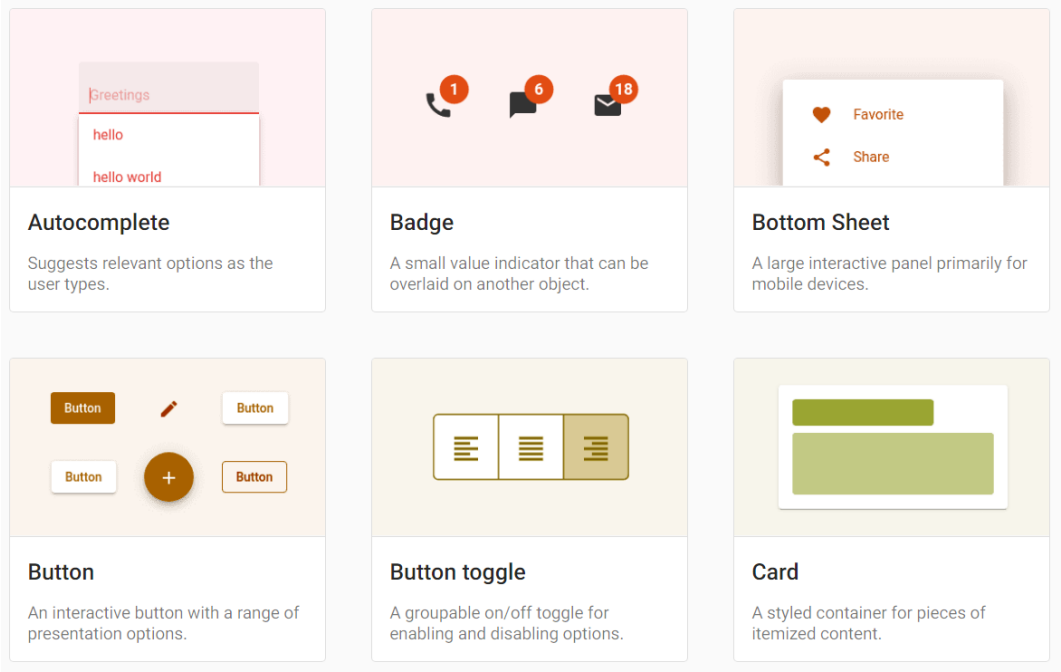


Figure 33: Angular Material UI components.

Another Angular UI component library is Angular Material UI (Team, n.d.), shown in Figure 33. This library, like MatBlazor and MUI, has components that follow Material Design principles. Angular Material UI is built by the Angular team and integrates well with Angular.

Components are well tested, internationalized, accessible, and documents with clear APIs. Angular Material UI also comes with tools to help developers build and customize components.

5.5 Graph Drawing

We researched strategies for creating visual representations of bond graphs. Our goal is to show system diagrams and bond graphs while restricting the size of the display and minimizing the number of edge crossings. We examined a set of characteristics that bond graphs share to understand how bond crossings could occur in a display. We then looked at researched display methods given the set of characteristics we compiled.

5.5.1 Characteristics of Bond Graphs

All bond graphs exhibit two characteristics that directly impact display: connectivity and planarity. Connectivity in a graph is the number of subgraphs containing vertices that can all reach each other traveling only along paths of vertices and edges defined in the graph. Bond graphs are connected, which means that every pair of vertices is bridged by at least one path. Bond graphs are also planar, which means they can be displayed on a two-dimensional surface flat screen with no edge crossings.

5.5.2 Drawing Algorithms: Force-Directed Optimization

One algorithm we researched and tested represents the graph as a physical system. This program models how graph elements interact when they apply forces on each other like physical objects do. To create an optimal embedding of the graph, the algorithm employs two types of forces: attractive and repulsive. The algorithm employs these by pretending that an edge between

two vertices behaves like a spring. Attractive forces act on these vertices that share an edge but are further than a specified distance apart. This kind of force will pull neighboring vertices closer together, as though the spring is contracting. However, if the neighbors are closer than this distance, repulsive forces will cause the spring to decompress and the vertices to repel each other. Repulsion also pushes apart vertices that do not share an edge. One drawback to this algorithm is that it does not automatically uncross crossed edges, which could result in unnecessarily cluttered drawings (Koren, 2005).

5.5.3 Drawing Algorithms: Spectral Graph Optimization

The Laplacian matrix is derived from connections between the vertices and edges in a graph, and it can reveal information about the graph's structure. This matrix is indexed on the rows and columns by the vertices of the graph. The diagonal elements of the Laplacian are equal to the degree of each vertex, or the number of edges with an endpoint at that vertex. When off the diagonal, an element is set to minus one if the edge between the two index vertices exists in the graph; otherwise, it is set to zero. In the Laplacian matrix, the number of eigenvectors associated with the zero eigenvalue reveals the number of connected subgraphs. For connected graphs, this means there will be exactly one such eigenvector.

The second option for visually representing graphs is known as the spectral graph and is derived from the Laplacian matrix. The location of each vertex in an image or plane is represented as a coordinate pair (x, y). The algorithm places starting estimates for the points in the form of two vectors, with each x value an element in one vector and each y value an element in the other. The program then repeatedly applies transformations to them until they settle into specific positions on the graph. There are multiple ways to iteratively change the coordinates so

that they converge to some pair of solution vectors. Each coordinate pair can be changed element-by-element based on the coordinates of the neighbors of a given vertex. The vectors can also undergo repeated linear transformations based on the adjacency matrix of the graph.

One problem with this algorithm is that it allows solutions that are mathematically optimal but not necessarily visually appealing. When graph elements can be placed at any location within the interface, the program tries to pull them together as closely as possible. Without constraints that set a minimum distance, the algorithm will simply condense the graph into a single point. The computed energy is then zero, the minimum possible. While successfully constraining all graph elements to a bounded region, it also makes the vertices and labels in the graph impossible to decipher (Koren, 2005).

One potential solution is to fix the locations of some nodes. This prevents collapse into a single point by maintaining a specific distance between some pairs of nodes. After setting these default locations, the computer will then place the remaining nodes one by one. These leftover graph elements will be put close to or on the weighted centroid, or computed center of the nodes already in the canvas. This solution generates its own set of problems. For one, the program must find a way to determine which nodes to fix at the beginning. Depending on how they are connected, certain starting sets of graph elements could force edges to cross, resulting in a cluttered graph image. The other determination is where each vertex in the initial set is placed. Because of these problems, the final product may have only a locally minimal energy state. Similarly, to the force-directed algorithm, the spectral graph may also display with unnecessary edge crossings (Koren, 2005).

6 Methodology

In this section, we detail our methodology for completing this project. First, we discuss the methodology used when first compiling a list of topics for research. We then discussed the process used to determine which UI component library we would use. We then discussed the process for deciding which technology stack to use. We then discussed the process we used to determine if the technology stack we had chosen would be able reuse the existing code for generating bond graphs from system diagrams. We then discussed the process we used for creating prototypes of elements of the user interface. Last, we discussed our team's methodology for developing the project in terms of tools we would use and the order in which we would add features.

6.1 Initial Research Topics

In the preliminary stages of our project, we outlined several research areas for building a background on necessary topics. The research process started with reading the MQP report that was completed by the previous team (Courville et al., 2020). This would give us background on the previous motivations for the project. Additionally, we would gain a sufficient background in the rationale behind the choices made in areas such as UI. We were also able to learn what the previous team felt should have been done differently.

We started by getting background on bond graphs. Bond graphs are the motivation behind our project, so this was a critical area to build a background on. We also researched how system diagrams are processed into bond graphs.

Next, we looked at GraphSynth. GraphSynth is the library which we used to recognize grammar rules and generate bond graphs. We examined the documentation for this library, as well as how it was used in the previous MQP.

The second research area that we investigated was in tech stacks, software that provides a complete framework for creating an application. Within this category we looked at many different resources to get information on the performance and software strengths of different tech stacks. We started by looking at many different tech stacks before settling on Blazor, React, Angular, OpenSilver, and Adobe Air as options to study further.

Related to tech stacks, we also looked at different UI libraries to understand which one would best suit our project. These libraries included Ant Design, Blazorize, MatBlazor, BlazorStrap, Material UI, Blueprint UI, React Bootstrap, UI Bootstrap, and Angular Material. Our team was looking at what specific components each of these UI libraries had.

By completing this research, we were able to build a background on all topics necessary to fully understand the depth of our project and be able to make important decisions like deciding what tech stack we would use.

6.2 Choosing a UI Component Library

In this section, we lay out our process for choosing a UI Component Library. We discussed in Section 5.4 why a project might use or avoid UI component libraries. Since BoGL Web is not a large commercial project that would benefit from components with a unique brand and since the interface includes many generic components, we did not see any significant advantages to making our own UI components. Using a UI component library would allow us to spend less time creating and testing the UI, and it would instead let us focus more on the difficult aspects of

the graph display area. Additionally, using a UI component library would also allow members of our team less familiar with creating UI to easily edit and add to our interface. Finally, we decided to only use a single UI component library instead of combining multiple to simplify the project and make sure all elements have a similar visual style.

6.2.1 Judging Criteria

We constructed a decision matrix (see 3.2.1 Decision Matrix) to rank UI component libraries. Given this, we first needed to determine what UI components BoGL Web needs from a library and what attributes those components should have. To do this, we first broke down BoGL Desktop's UI into sections and components. We determined that every part of the UI except for the system diagram construction and the bond graph display area should be replicated using a UI component library. The graph and system diagram areas would be rendered with JavaScript. As we identified necessary components, we ranked them on a scale of one to three, indicating the difficulty of replicating the component's appearance and functionality without using a UI library, with one being easy and three being difficult. We identified the slider, dropdown menu, and tabs as the most important components to be in a UI library, the dropdown menu and checkbox as second most important to be in a UI library, and the icon button, collapsible, and button as not especially important to be in a UI library. In future sections, we refer to these UI features as "three-point components."

6.2.1.1 Three Point Components

The first three-point component is the slider. In BoGL Desktop, there was a slider in the zoom menu at the bottom right-hand corner of the graph area (see Figure 34) which needed to be replicated. Key features of the slider are the ability to be vertical and show a tooltip indicating the slider's current value. Being able to replicate the shape that marks the slider's current value is also useful but was not considered as vital to maintaining BoGL Desktop's UI. Despite being a native HTML input tag, the tooltip and vertical orientation of the slider would be difficult to implement with our own code, earning the slider three points.

Figure 34: BoGL Zoom Slider

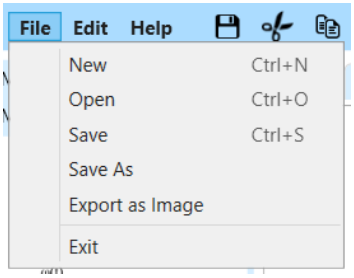


Figure 35: BoGL File dropdown menu.

The second three-point component is the dropdown menu, which is used in the top menu bar in BoGL Desktop's UI (see Figure 35). This component has a base button that, when clicked, shows a dropdown menu. These components enable the File, Edit, and Help menus to contain multiple options in a compact format. Dropdown menus are not native HTML tag and would need to be created using custom HTML, CSS, and JavaScript.

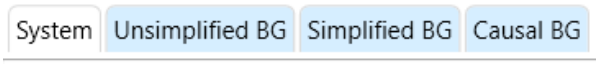


Figure 36: BoGL tabs.

The third three-point component is the tabs. Tabs are used to toggle between the System Diagram, Unsimplified Bond Graph, Simplified Bond Graph, and Causal Bond Graph displays (see Figure 36). Key features for tabs are the ability to set tab color and replicate the rounded tab style from BoGL Desktop. Tabs are not a native HTML tag and are usually constructed using buttons for tabs, “div” tags for tab areas, and JavaScript code for switching between tabs. Using a library hides the underlying code that makes tabs work, reducing the amount of code we write and leading to a more robust implementation.

6.2.1.2 Two Point Components

Figure 37: BoGL dropdown selector.

The first two-point component is the dropdown selector. This component is used on the Causal Bond Graph tab of BoGL Desktop, allowing users to pick from the bond graphs that were generated if multiple bond graphs can be generated from the given system diagram (see Figure 37). As with the slider, the dropdown selector is also a native HTML tag, but styling the component to match the rest of the UI would be difficult without using a library, which helps enforce consistent styling for all UI components.

Figure 38: BoGL icon button array.

The second two-point component is the icon button. This component is used in the top menu strip to represent the save, cut, copy, paste, undo, redo, and delete functions, respectively (see Figure 38). Each icon button has an icon, hover text, and a square highlight on hover and click. Icon buttons are not part of native HTML and can be constructed with a div or button and an image tag. The icon button should be able to be disabled. Making an icon button manually is difficult, since the component has a state, being enabled, or disabled, and preferably has highlight and click effects. Unfortunately, UI component libraries are limited when it comes to replicating these icon buttons faithfully because some only allow you to pick from a set list of icons and many use a circular highlight instead of a square one.

6.2.1.3 One Point Components

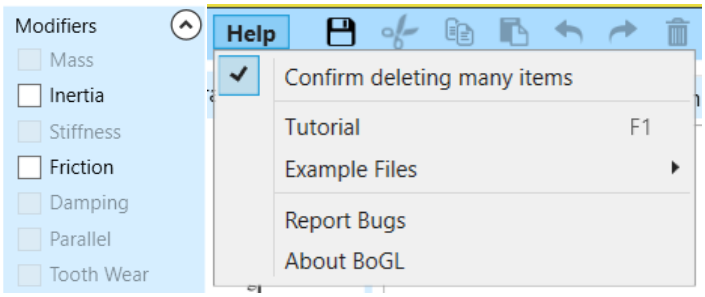


Figure 39: BoGL modifier menu and Help dropdown menu.

The first one-point component is the checkbox. Checkboxes are used in the modifier menu that appears on the system diagram tab and in the Help menu (see Figure 39). One key

aspect of this component is that it must be able to be disabled since different bond graph elements support different modifiers. Checkboxes are part of native HTML and can be disabled with JavaScript, so they have the lowest priority ranking relative to other components. Even so, they are difficult to style, so using a library is helpful for matching their style to the rest of the UI. Checkbox elements in libraries need to have an easy method for disabling the checkbox.

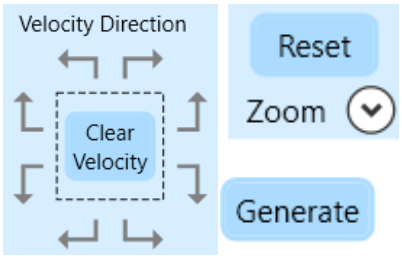


Figure 40: BoGL velocity direction menu, reset button, and generate button.

The second one-point component is the button. Buttons are used in three places: the Velocity Direction section of the Modifiers menu, the Reset button in the Zoom menu, and the Generate button below the zoom menu (see Figure 40). The Generate button can be disabled, so the button component must have this capability. Buttons are a native HTML tag and are easy to style, so the main appeal of using a library button is consistent styling with the rest of the UI and handling of slightly more complex styling like hover and click highlight animations.

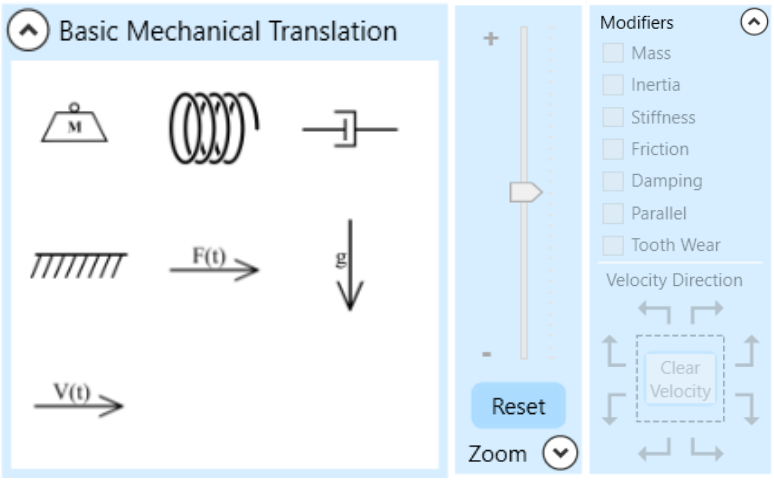


Figure 41: BoGL Basic Mechanical Translation collapsible, zoom menu, and modifiers menu.

The third one-point element is the collapsible component. Collapsibles are used in three places: in the left-hand side menu that holds bond graph elements, in the Zoom menu, and in the Modifiers menu (see Figure 41). Collapsibles must be able to toggle between a collapsed and open state. They must also be able to hold an icon before or after their header text and allow a large degree of freedom for how the header and body areas of the collapsible are styled. This freedom is needed to replicate BoGL Desktop's collapsible with fidelity. However, many collapsible components from UI libraries give the collapsible a set style that is not easy to modify, so the ability to modify the component factored heavily into our scoring. Collapsibles are not a part of native HTML, but they can be constructed easily with a header tag and a body tag, where the header has a simple click action to show and hide the body. Given that we need a significant amount of control over the collapsible's styling, we would prefer this component to be included in a UI library, but we would be able to make this component from scratch if necessary.

6.2.2 Choosing Ant Design

In this section, we discuss the rationale behind why we chose to use Ant Design as our UI component library.

6.2.2.1 Decision Matrix

Decision matrices are a method for evaluating and prioritizing a list of options. Each option is evaluated in several categories, where each category has a weight representing its relative importance. Category scores represent how well an option fulfills the category requirement. When choosing a UI component library, the options are UI component libraries, and the categories are necessary UI components. All weights were assigned on a scale of 1-3 and all scores on a scale of 0-3. Scores were assigned according to how well a library could replicate the functionality and appearance of a UI component. A score of three means the UI component fulfills the requirements well, a score of two means the component has minor issues that can be overcome relatively easily, a score of one means the component exists but has major issues in being able to replicate a given UI component, and a score of zero means the UI component does not exist in the library. None of the libraries were completely missing a necessary UI element, so the table has no zero scores, but it was possible for libraries to score a zero for an element. The category totals take the highest weighted total for each tech stack, since that score represents the best option for that tech stack.

Table 6: UI component library decision matrix.

		All	Blazor and OpenSilver			React			Angular	
UI Element	**Weight**	**Ant Design**	**Blazorize**	**MatBlazor**	**BlazorStrap**	**Material UI**	**Blueprint UI**	**React Bootstrap**	**UI Bootstrap**	**Angular Material**
Button	1	3	3	3	3	3	3	3	3	3
Slider	3	3	1	1	1	3	3	1	2	3
Dropdown selector	2	3	2	3	3	3	3	3	3	3
Dropdown menu	3	3	3	1	3	3	3	3	3	3
Checkbox	1	3	1	3	3	3	3	3	3	3
Icon button	2	3	1	2	1	2	3	1	3	2
Tabs	3	2	2	2	2	2	2	3	2	2
Collapsible	1	2	2	3	2	3	2	2	2	3
	Weighted Total	2.8	1.4	2.1	1.9	2.4	2.0	1.7	2.2	2.0
	Category Total	2.8	2.1			2.4			2.2	

6.2.2.2 Conclusion

As Table 6 shows, Ant Design scored 2.8, Blazor and OpenSilver scored 2.1, React scored 2.4, and Angular scored 2.2, with all scores being out of three. Otherwise, Blazor, OpenSilver, React, and Angular received similar overall category scores for tech-stack-specific UI component libraries. Ant Design, however, is compatible with all tech stack options and scored significantly higher than any other UI component library. The implementations of Ant Design in each tech stack are identical, with the only differences being components BoGL Desktop does not use. Given these results, we concluded two things: one, that regardless of tech

stack we would choose Ant Design for our UI component library, and two, as a result, quality of UI component library options has no effect on our choice for tech stack.

6.3 Tech Stack Methodology

In this section, we discuss the methodology used to determine which tech stack would best fit the needs of BoGL Web. This was done using a decision matrix. We broke this decision matrix down into several categories including Usability, Speed, Space, and Programmability.

6.3.1 Categories

We started the process of choosing a tech stack by first considering four categories that we thought would be important qualities for our chosen tech stack to have. One category is *usability*, which represents how good the user experience is with the application. Another category, *speed*, includes both the speed of UI interactions and computation, both of which affect the user. We also considered *space*, which is the footprint that our application would take up on both a client and a server. Finally, we considered *programmability*, or how easy it would be for our team, as developers, to develop.

Within these categories, we created several subcategories which further broke down the major features that we thought would be beneficial to have in a tech stack. Within the usability category, we had the subcategories *offline usability* and *ability to replicate* BoGL Desktop's UI. For speed, we had the subcategories *client load speed*, *UI interaction speed*, and *computation speed*. Client load speed is the speed of the website loading, from the time the client presses enter on the URL to the time the website has fully loaded. UI interaction speed was used to score the responsiveness of the UI when completing various tasks. Websites which had visible lag had

a lower score than those which felt responsive. Computation speed represents each tech stack's ability to handle larger computations. Tech stacks which did not have visible lag when performing significant computations scored better than those which responded slowly.

The space category had two subcategories, *space taken up on the client side* and the *space taken up on the server side*. Finally, we had four subcategories for programmability: *code reusability*, *ease of writing code*, *community support*, and *server interaction*. Code reusability refers to the amount of existing BoGL Desktop code that we would be able to reuse with the given tech stack. Ease of writing code represents the general learning curve for each tech stack. Community support considers the size of online communities using a given tech stack, as estimated through GitHub topics and StackOverflow posts. Finally, server interaction is the amount of times information will need to be sent back and forth within tech stack, for example, a request and response message for generating bond graphs.

6.3.2 Scoring of Categories

Scoring was done on a scale of one to five. In this scoring system, one means that the tech stack did a poor job at exhibiting the desired quality, while five indicates that the tech stack is optimal for the desired quality. Scores in the middle indicate that the tech stack was sufficient for that category but not preferred. Weights were assigned by first doing research on each tech stack for a specific feature. Following this, we assigned weights to each feature for each tech stack based on how they performed compared to each other. We started by researching each tech stack and creating Table 7 to visualize important features.

Table 7: The logic we used to determine scores given to each tech stack in various categories.

Category	Subcategory	Blazor Client side (Wasm)	Blazor Client-server	React + ASP.NET server	Angular + ASP.NET server	OpenSilver
Usability	Offline Usability	Can easily support offline use as a PWA	Can support some basic functionality offline as a PWA, but not bond graph computation	Can support some basic functionality offline as a PWA, but not bond graph computation	Can support some basic functionality offline as a PWA, but not bond graph computation	Can support some basic functionality offline as a PWA, but not bond graph computation
	Ability to Match UI	Ant Design	Ant Design	Ant Design	Ant Design	Ant Design
Speed	Client Load Speed	Not Cached: Seconds, Cached: Seconds	Not Cached: Milliseconds, Cached: Milliseconds	Milliseconds	Milliseconds	Not Cached: ~ Ten Seconds, Cached: Seconds
	UI Interaction Speed	UI is smooth and reacts quickly to input	UI is smooth and reacts quickly to input	UI is smooth and reacts quickly to input (same test site as above)	UI is smooth and reacts quickly to input some slight lag (same test site as above)	Tab switching has visible lag (same site as above)
	Computation Speed	Somewhat slower computation that client side Blazor with large datasets, but not significant with small data sets. Was able to handle a reasonable amount of computation with no visible lag	Somewhat faster computation that client side Blazor with large datasets, but not significant with small data sets. Was able to handle a reasonable amount of computation with no visible lag	Same as Blazor client-server, since the server computation is similar (only UI interaction changes)	Same as Blazor client-server, since the server computation is similar (only UI interaction changes)	Very laggy when doing intense computations
Space	Client-side	All site resources must be downloaded to the client. On first load, 10,600 kB are downloaded, which becomes 24,400 kB when uncompressed. When cached, 5-7% of this is downloaded and stored when the site loads	Most of the website is stored on the server, so not much client space is needed. Regardless of caching, about 1100 kB are downloaded and stored when the site loads	Most of the website is stored on the server, so not much client space is needed. React 16.2.0 + React DOM, a relatively recent version of React, is 31.8Kb. Ant Design, a possible UI framework, would add 356.8 kB to that, minified.	Most of the website is stored on the server, so not much client space is needed. Angular 2 is 111 Kb. Ant Design, a UI framework, would add 356.8 kB to that, minified.	Clients are required to download a large amount of information. For a light web application, the client needed to download a total of 20.1 MB of data. Data is still compressed however.
	Server-side	Just need to store the client-side application, does not scale with number of users	Sessions are setup in memory for each user and users are tied to one server, so load cannot be balanced at all	With a smart implementation, the server should only need the code for our app, GraphSynth and the Grammar rules. We would be able to implement load balancing to reduce the effect from having many users.	With a smart implementation, the server should only need the code for our app, GraphSynth and the Grammar rules. We would be able to implement load balancing to reduce the effect from having many users.	Just needs to store the client, application is executed on the server-side

Table 7: The logic we used to determine scores given to each tech stack in various categories.

Programmability	Code Reusability	Can fully reuse existing code	Can fully reuse existing code	Can reuse code depending on server architecture	Can reuse code depending on server architecture	Can fully reuse existing code
	Ease of Writing Code	There are some sources for learning the technology, but most only cover surface level topics	There are some sources for learning the technology, but most only cover surface level topics	Low learning curve	High learning curve	Exceedingly difficult because it is very new and there is currently no community for it. Not really designed for building new applications.
	Community Support	StackOverflow has 9395 questions in blazor tag, 2477 in blazor-webassembly, and 719 in blazor-client-side. GitHub Topics has 2,621 Blazor repos, 907 blazor-webassembly repos, and 163 blazor-client repos	StackOverflow has 9395 questions in blazor tag, 2477 in blazor-webassembly, and 3303 in blazor-server-side. GitHub Topics has 2,621 Blazor repos and 590 blazor-server repos	StackOverflow has 411,822 questions in reactjs tag and 121,859 in react-native. GitHub Topics has 235,510 React repos	StackOverflow has 283,523 questions in angular tag and 262,549 in angularjs. GitHub Topics has 41,538 Angular repos	StackOverflow has eight questions that mention OpenSilver and no GitHub Topics
	Server Interaction	No server interaction	SignalR communicates between Blazor server and Blazor client, only twenty simultaneous connections and 20,000 messages / day are free	Implementation dependent. Most interaction would occur when graphs need to be processed.	Implementation dependent. Most interaction would occur when graphs need to be processed.	Execution is done client side

6.3.3 Usability

The usability category received a weight of five because giving clients a good user experience is our priority for BoGL Web. If our users are unable to complete the tasks that they wanted to, there would be no reason to create the application.

6.3.3.1 Offline Usability

Offline usability received a high rating when a tech stack had the ability to continue working if a client loses their internet connection or starts using the tool without an internet connection. This could be achieved by making the website a Progressive Web App (PWA), which would allow the client to download BoGL Web and run it like a regular desktop application. While any of our tech stacks could be converted to a PWA, Blazor WebAssembly received a higher score of five in this category since it does all computation client-side and can therefore run with full functionality offline. Other tech stacks converted to PWAs would have some benefits for user experience, such as letting the user construct a system diagram and use export/import functionality. This, however, would not allow users to generate bond graphs from system diagrams, since this functionality would occur on a server. While this feature benefits the user experience, it is not required if the user has an internet connection, so we rated it two out of five.

6.3.3.2 Ability to Replicate UI

We considered a tech stack's ability to replicate the UI from BoGL Desktop an important feature, earning the category a score of four. One of the design requirements for the application was that the look and feel of the web application would closely resemble that of BoGL Desktop.

While each tech stack had the potential for different scores in this category, we ended up giving each tech stack a score of five. We concluded that the tech stacks had equal ability to replicate BoGL Desktop's UI based on the results of the UI component library decision matrix (see Choosing a UI Component Library), since Ant Design was the best UI component library for replicating the UI and it was compatible with all the tech stacks.

6.3.4 Speed

The speed category was given a weight of four. This is because speed is a key factor in the client having a smooth experience when using the application. This category was not given a five due to the usability of the application being seen as more important than the speed and our understanding that generating bond graphs will result in some delay regardless of tech stack (see Table 9).

6.3.4.1 Client Load Speed

The table below lists the websites we used to test the speed of various aspects of the Blazor Framework. For most of these websites, we include links, and for the Blazor Example Website is automatically generated when creating a Blazor project.

Table 8: Websites used to test various elements of the decision matrix.

Tech Stack	**Website**	**Test Performed**
Blazor Wasm	Blazor Example Website	Client Load Speed
Blazor Client-Server	Blazor Example Website	Client Load Speed
React + ASP.NET	https://ahfarmer.github.io/calculator/	Client Load Speed
Angular + ASP.NET	https://angular.io/guide/example-apps-list	Client Load Speed
OpenSilver	http://opensilvershowcase.azurewebsites.net/?20211012#/Welcome	Client Load Speed
Blazor Wasm	https://aesalazar.github.io/AsteroidsWasm/	UI Interaction Speed
Blazor Client-Server	https://wengier.com/Trains.NET/	UI Interaction Speed
React + ASP.NET	https://ahfarmer.github.io/calculator/	UI Interaction Speed
Angular + ASP.NET	https://angular.io/guide/example-apps-list	UI Interaction Speed
OpenSilver	http://opensilvershowcase.azurewebsites.net/?20211012#/Welcome	UI Interaction Speed
OpenSilver	http://opensilvershowcase.azurewebsites.net/?20211012#/Performance	Computation Speed

The client load speed represents the speed at which a client will fully load onto the website. This includes downloading all the necessary content for the website to function properly. This was measured using several different test websites that all had a similar amount of content to be loaded. The websites used can be found in Table 8. From this, we were able to rate the tech stacks, giving websites which loaded within milliseconds a five, websites which loaded within seconds a three, and websites which loaded in around ten seconds a one. This feature was given a weight of three since users will need to expect to wait some amount to load into the website, but we do not want it to take an exorbitant amount of time.

6.3.4.2 UI Interaction Speed

UI interaction speed was measured in an analogous manner to the client load speed. A test website was used for each tech stack and the smoothness of performing different tasks was measured. The test websites are listed in Table 1 as well as what part of the website was used for the test. Blazor WebAssembly, Blazor Client-Server, and React with an ASP.NET backend all received ratings of five. Angular with an ASP.NET backend received a four due to some lag, and OpenSilver received a three due to having visible lag when switching tabs. This feature was given a weight of five because users will get frustrated if the UI has a lot of lag.

6.3.4.3 Computation Speed

The computation speed was also measured with example websites, all of which are listed in Table 1. All the tech stacks received scores of five except for OpenSilver, which was scored as a two due to it getting very laggy when a lot of computation was being done. This feature was given a weight of four since our application will be doing a lot of computation in the backend, and we want our tech stack to be able to handle this amount of computation.

6.3.5 Space

The space category was created to track the client and server-side space that the application takes up. This category was given a weight of three since we do not want the application to have a huge footprint on the client or server.

6.3.5.1 Client-Side

The client-side space category represents the amount of space the client would need to load and store data when the website is visited. This was measured by researching the amount of data that each tech stack needs to transfer to run. OpenSilver was the worst in this category and received a rating of two. Blazor WebAssembly was next since it still needed to download the entire website but was able to cache some data to reduce the amount that needed to be downloaded on subsequent visits. This got Blazor WebAssembly a rating of three. Blazor Client-Server and Angular with ASP.NET both received scores of four since the clients did not need to score much data, as the majority was stored on the server side. Reacting with an ASP.NET backend was the only tech stack to receive a five since it downloaded the least data. This category was weighted as a three since users may have to download a lot of data due to the many rules and images that will be required for the website to run. We do not want to overburden users with information so it is important to ensure that the tech stack we choose will not require users to download gigabytes of information.

6.3.5.2 Server-Side

The server-side space subcategory represents the amount of space the application would take up on a server. Information for this section was also gathered through research done in the literature review process. Tests would be inconsistent here due to the variety in the structure of

test websites. In this subcategory, Blazor Client-Server received the worst score of one due to the way that each client is handled. React and Angular both received fours since, if implemented in a smart way, the amount of data that would need to be on the server would be minimal. Finally, OpenSilver and Blazor WebAssembly received fives since the server would only need to store the website. This subcategory received a weight of two since the amount of space that is being taken up on the server is a concern when it comes to the scalability of the website, but it is not as important to the user experience.

6.3.6 Programmability

The final category we looked at was programmability. This category considered the ease of developing a website in the given tech stack. The category received a weight of one since we, as developers, felt that we would be able to develop using any tech stack, even if it were more difficult than another. Also, we wanted to prioritize all other aspects of the application before our development comfort.

6.3.6.1 Code Reusability

Code reusability is our team's ability to reuse the code that previous MQPs wrote. This category received a weight of five since being able to reuse code would reduce the difficulty of development significantly. Of the tech stacks we looked at, both Blazor stacks as well as OpenSilver received fives since they use C# by default. This is significant since GraphSynth, the library used to recognize and apply grammar rules, is written in C# and would require interoperability with that language to run. React and Angular can also use C# when used with ASP.NET, but this locks us into a specific implementation server side, leading to a score of four.

6.3.6.2 Ease of Writing Code

Ease of writing code related to the general learning curve and ease of development for a tech stack. We weighed this category as one since we felt that we would be able to write code in any tech stack even if it were more difficult. Blazor stacks received a score of three in this category due to the limited number of tutorials online. React received a five due to its low learning curve. Angular received a two due to its high learning curve. Finally, OpenSilver received one since it was very new and there were almost no resources for it online.

6.3.6.3 Community Support

Community support related to the number of forum posts and public projects which have been developed in each tech stack. This category was weighted a three since having a large community would make debugging quite a bit easier, and lead to many more libraries having been developed for the tech stack. This category was ranked by looking at the number of StackOverflow posts and GitHub repositories that had been created for each tech stack. Blazor stacks were given threes here since they had a reasonable amount of information in these locations (see Table 9). React was given a five since it is an extremely popular tech stack. Angular was given a four since it too is quite popular, but not to the same extent as React. Finally, OpenSilver was given a one as it only had eight posts and no GitHub repositories, meaning there is almost no community support for it.

6.3.6.4 Server Interaction

Server interaction was the final category in this section. This category was used to evaluate the amount of communication that would need to occur between the client and the server. Higher scores here means that there was less interaction since that was more desirable for

our application. This category received a weight of four since more server interaction requires more work on the developer side in learning how to send requests and in handling bad requests, which makes it desirable to interact with the server as little as possible. Blazor WebAssembly and OpenSilver received fives in this category since the only interaction they require is the initial download of the website. The other tech stacks received threes since a lot of communication can be avoided with smart implementation, but there will still be significantly more interaction than the two tech stacks which received fives.

6.3.7 Final Scores

The final scores can be seen in Table 9. Each number in the totals are averages of the weighted scores for that column. From this we can see that Blazor WebAssembly is the best tech stack for our application as it received the highest average score.

Table 9: The final decision matrix for tech stacks.

Category	Category Weight (1-5)	Subcategory	Subcategory Weight (1-5)	Blazor WebAssembly	Blazor Client-server	React + ASP.NET	Angular + ASP.NET	OpenSilver
Usability	5	Offline Usability	2	5	2	2	2	2
		Ability to replicate UI	4	5	5	5	5	5
Speed	4	Client Load Speed	3	3	5	5	5	1
		UI Interaction Speed	5	5	5	5	4	3
		Computation Speed	4	5	5	5	5	1
Space	3	Client-side	3	3	4	5	4	2
		Server-side	2	5	1	4	4	5
Programmability	1	Code Reusability	5	5	5	4	4	5
		Ease of Writing Code	1	3	3	5	2	1
		Community Support	3	3	3	5	4	1
		Server Interaction	4	5	3	3	3	5
			Usability Average	5.0	4.0	4.0	4.0	4.0
			Speed Average	4.5	5.0	5.0	4.6	1.8
			Space Average	3.8	2.8	4.6	4.0	3.2
			Programmability Average	4.4	3.8	4.0	3.5	3.8
			Final Average	4.2	3.7	4.1	3.9	2.8

6.4 BoGL Desktop Port Methodology

In this section, we will discuss a test we conducted in the preliminary stages of our project. One concern the team had when asked to move the existing BoGL Desktop application to

something web based was the issue of processing speed. Bond graphs are generated with a computationally expensive algorithm which we were concerned may not be able to run in a web browser. We also wanted to ensure that we would be able to use GraphSynth in Blazor WebAssembly. To verify the viability of the project and ensure that we would in fact be able to process bond graphs in a web browser, we completed the following test.

The goal of running the existing BoGL Desktop code in Blazor WebAssembly was to test if we could generate bond graphs from system diagrams within browser memory constraints. We also wanted to ensure that the browser would not time out and that the users would not be waiting an unreasonable amount of time after they had pressed the generate button. Through the test we determined that BoGL code running in Blazor WebAssembly can process bond graphs in a reasonable amount of time: between one and three seconds. This means that the computer would complete the request before the user lost patience with any program execution delays. This is important from a human computer interactions perspective as extensive delays will create frustration with the users and make them think that something went wrong. These tests also allowed us to verify that the correct bond graph was generated when graphs were processed in the converted system.

We needed to make several modifications to enable BoGL Desktop to run in a browser with Blazor WebAssembly. Our first adaptation changed the way files were loaded. BoGL Desktop computer file locations using StreamResourceInfo and loaded files using StreamReader, which would not work in Blazor WebAssembly since it does not have direct access to the file system. Instead, an HTTP client needed to be created to make HTTP requests to files stored in the local “wwwroot” folder. The program then receives an HTTP response containing the file's content as a string. This string allows us to perform the same processing as BoGL Desktop. This

process is done asynchronously, which prevents the UI from freezing while the system loads files.

The second change we made was not a direct result of the transition to Blazor. To reduce the number of lines of code and variables needed to store all the rules and increase readability, we created a dictionary to map rule name strings to their respective rules. These strings were generated using a substring of the rule file path. This allowed us to load the rules in a cleaner manner. Instead of loading each rule one by one in its own block of code, a loop was created to iterate through all the rules, which reduced the amount of code needed to load the rules. The change made it so that we could access rules though the dictionary that had been created instead of referencing a variable; a string was passed to the dictionary and the dictionary would return the requested rule.

The third change we made was removing all the UI elements from BoGL Desktop. Since we wanted our test to output data purely to the console, they were not needed and would not need to be added anywhere else. This was also done because Blazor WebAssembly does not have any of the UI components which were present in BoGL Desktop. The dependency on some UI components meant that they would need to be removed for the code to compile. This was a problem when it came to the intermediate graph that was used to communicate between the UI and GraphSynth in BoGL Desktop. This graph was stored as a UI element, directly tying it to a component of the application that would not be carried over to Blazor WebAssembly. To solve this, we created a variable in the GraphProcessor class to represent the current system diagram the user had loaded. This variable would be passed to the function which translated system diagrams stored by BoGL into something that could be understood by GraphSynth. The second issue to be overcome was figuring out what to do with a system diagram that stored information

pertaining to the UI which no longer existed. In all cases, this information could just be removed. This made the translation of this part of the application simple as it just involved deleting a lot of existing code.

The UI that we built to complete these tests was quite simple (see Figure 42). It consisted of three buttons: one to load all the rules/rulesets, one to load the example system diagram, and one to generate a bond graph from the system diagram. The output appeared in the console, which was accessed with Google Chrome's web developer tool panel. The main output from the code, which could be seen in the developer tools console, consisted of reports indicating the completion of tasks necessary for loading the rulesets and rules, loading an example graph, and generating the bond graph. This was sufficient for our tests because we were primarily concerned with the time it would take to process the system diagram into a Bond Graph.

Even though it was not the main goal of these tests, we did check the resulting bond graph for correctness. This was done using Visual Studio's debugger. To complete the verification, the same system diagram was processed in both BoGL Desktop and the converted system. Using the debugger (see Figure 43) we were able to see how many bond graph arcs and nodes were being produced by the converted system which was compared to the number of arcs and nodes which could be visually seen in BoGL Desktop (see Figure 44).

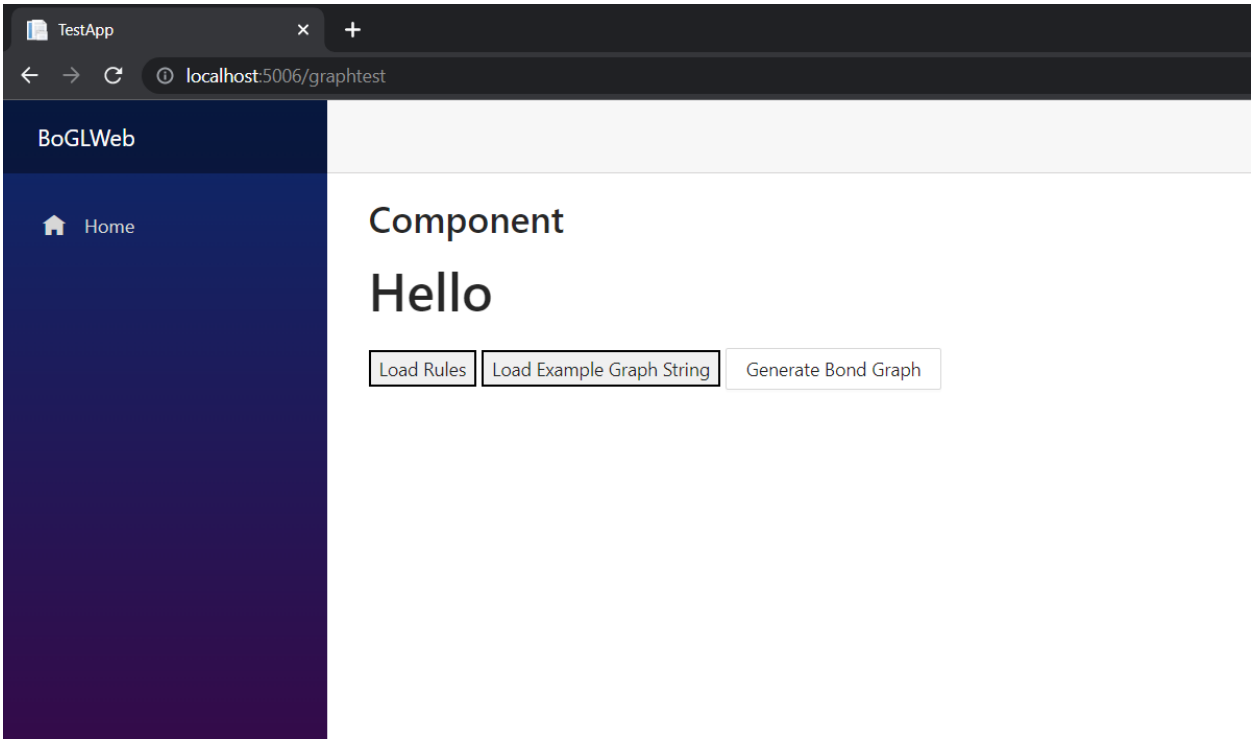


Figure 42: The user interface used for testing Bond Graph generation.

systemGraph	GraphSynth.Representation.designGraph	object
finalresult	Count = 1	object
Items	GraphSynth.Representation.designGraph[1]	GraphSynth.Represe...
0	GraphSynth.Representation.designGraph	GraphSynth.Represe...
_arcs	Count = 9	System.Collections....
_globalLabels	Count = 0	System.Collections....
_globalVariables	Count = 0	System.Collections....
_hyperarcs	Count = 0	System.Collections....
_nodes	Count = 10	System.Collections....
arcs	ƒ arcs ()	function
comment	null	object
DegreeSequence	ƒ DegreeSequence ()	function
globalLabels	ƒ globalLabels ()	function
globalVariables	ƒ globalVariables ()	function
HyperArcDegreeSequence	ƒ HyperArcDegreeSequence ()	function
hyperarcs	ƒ hyperarcs ()	function
name	'system_graph'	string
nodes	ƒ nodes ()	function
length	1	number
Add item to watch		

Figure 43: The Bond Graph processed in the converted system.

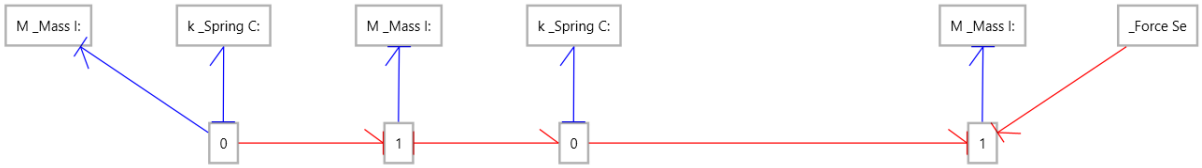


Figure 44: The expected Bond Graph.

One of the setbacks we faced in the conversion was loading example system diagrams. There was an issue where system diagrams were being loaded, but the designGraph, a GraphSynth class representing the system diagram, did not have any arcs or nodes. Since our system diagram had no data, we encountered issues in trying to apply grammar rules. We initially had two theories as to what was causing this. One possibility was that the XML was formed incorrectly, and the other was that the XML serializer class provided with Blazor was not working properly. While we did pinpoint bad XML formation as the root cause, it is worth noting that there were a lot of problems reported online stating that, when pushed to a production environment, Blazor's XML serializer did not always function properly. While this is not currently affecting the project, it is worth noting since it may affect us in the future when we attempt to deploy the application.

As stated previously, the cause of this issue was the system diagram XML having information which did not need to be passed to the deserializer. This led to the data not being parsed correctly. When initially created, the XML text has links which are supposed to help a human reader understand how the XML is written. These links prevent the XML serializer from deserializing the XML text correctly. The links were still present in XML due to a method which was supposed to remove those links that were not working correctly. We fixed this method, and the issue was resolved.

The other main setback that we faced when rewriting BoGL Desktop into Blazor WebAssembly was with two variables, sys_Graphs and sysGraphs, having similar names. The variable sys_Graphs is supposed to be a linked list and sysGraphs is a stack. In the initial conversion to Blazor WebAssembly, we replaced instances of the variable name sys_Graphs with sysGraphs which caused an infinite loop because the stack sysGraph was never emptying. This issue was easily fixed by reintroducing the variable sys_Graphs.

We gained two main insights from completing this test. First, we learned that bond graphs can be generated from system diagrams in a reasonable amount of time in Blazor WebAssembly. This confirmed the viability of the project and ensured that it would in fact be possible to process system diagrams in a browser. We were also able to confirm that the correct bond graph would be produced when a system diagram was processed in the browser.

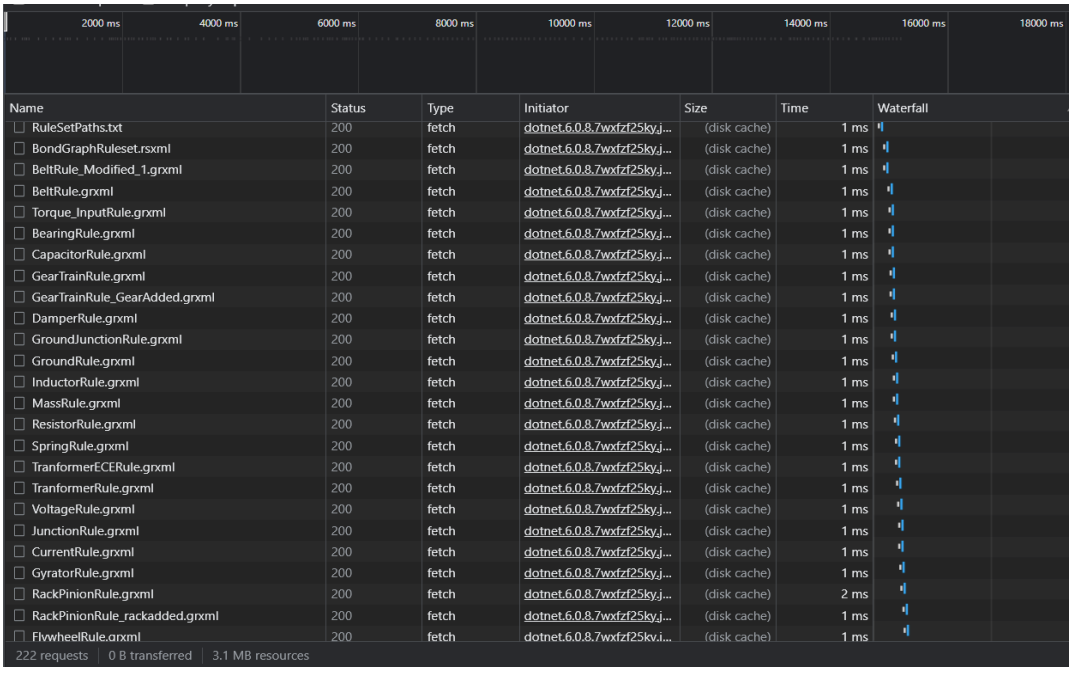


Figure 45: The time it took to load all rulesets and rules.

Through these tests, we learned that rules and rulesets take a long time to load. Loading the rules and rulesets takes over 16 seconds (see Figure 45). While this only needs to be done once, it may cause issues when the user wants to process the system diagram right after entering the website, such as when the user loads a system diagram from a URL (see Section 8.3.1).

6.5 UI Prototyping Methodology

We constructed UI prototypes using Blazor WebAssembly as our web development framework and Ant Design as our UI component library. The goal of making these prototypes was to determine what our final UI code would look like and to confirm that Ant Design and Blazor WebAssembly could support the UI functionality we needed. The prototypes also helped us identify parts of the UI that were challenging to implement and thus required extra design attention.

To make UI prototypes, we needed a Blazor WebAssembly project. Visual Studio has built-in templates for making Blazor WebAssembly and Blazor Server applications, so we generated a default Blazor WebAssembly app in Visual Studio. Helpfully, this template includes an option to make the app a PWA, adding all necessary configuration files for the user, so our default application proved just how easy it is to make a website a PWA and allowed us to immediately check that the website was downloadable. After making the initial application, we imported Ant Design through the _Imports.razor file and successfully added an Ant Design button to the project.

6.5.1 Top Bar Dropdown Menus

The first UI prototype we implemented was an incomplete version of BoGL Desktop's top menu bar. In the desktop application, this bar includes the BoGL logo, the File, Edit, and Help dropdown menus, and several icon buttons. In this first prototype, we replicated only the File, Edit, and Help dropdown menus. The purpose of creating this prototype was to test Ant Design components and determine how customizable they are. Its development also allowed us to become familiar with Ant Design while confirming that it would be able to faithfully recreate the top menu bar.

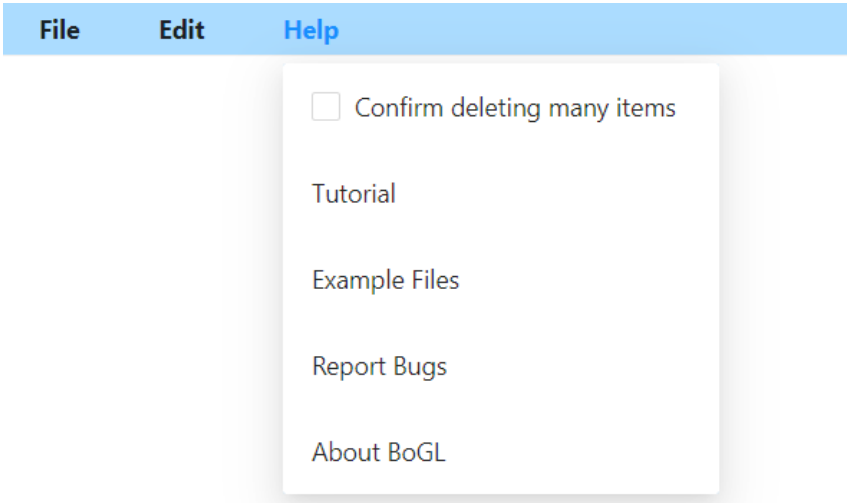


Figure 46: Top menu dropdown prototype.

We used the Blazor Ant Design Menu documentation page (*Menu - Ant Design*, n.d.) to put an example horizontal menu along the top of the application. After verifying that this example worked, we removed submenu icons and changed the submenu labels to reflect the menu names in BoGL Desktop: "File," "Edit," and "Help." We then added menu items to each

submenu to match those in the desktop application. The main issue we encountered in doing this was changing the style of the menu bar to match BoGL Desktop. The default Ant Design submenu highlights open submenus with an underline. Since BoGL Desktop does not use an underline here, we implemented custom CSS for the menu object to remove the underline and change the colors of the menu bar to match BoGL Desktop. This prototype also let us confirm that a checkbox can be put in a submenu, which is necessary for the Help menu (see Figure 46).

6.5.2 System Diagram Editor

After completing a prototype of Ant Design components, we focused on prototyping the system diagram editor. This is the area in BoGL Desktop where elements are dragged in and connected to make a graph. We expected this part of the application to be more complicated and require a custom solution. To start, we considered our options for this type of interactive display. Our most reasonable choices were HTML without canvas, HTML with canvas, and SVG (SXA vector graphics). The first option, HTML without canvas, would require constructing the graph through HTML tags such as “div.” HTML without canvas would have the benefit of being tag based, which allows event handlers to be applied to individual tags. This option also scales naturally with zoom, since most HTML tags scale to the resolution of the screen, so the resulting display would not appear grainy. While this method could illustrate the elements of a BoGL graph easily enough with div tags, it would be difficult to represent edges, since HTML does not have a line object with arbitrary positioning.

Another option is to draw the system diagram editor on an HTML Canvas using JavaScript. Through JavaScript, shapes like rectangles and lines can be positioned on the canvas to form the elements and edges of a system diagram. While this solution supports line drawing

better than HTML without canvas, each element and edge is part of one Canvas tag, which complicates handling mouse events. These events are recorded on the HTML Canvas tag, and the clicked element must be calculated by our code based on knowledge of element and edge positions. With most HTML elements, no calculation would be required as the event handling is automatically assigned to a single element rather than the whole display. HTML Canvas also often has issues with resolution and blurriness, especially when the canvas can be dynamically zoomed. While there are methods for combating this, blurriness will continue to be a possibility wherever HTML Canvas is used. HTML Canvas also offers third-party libraries that simplify the process of making displays and recognizing mouse clicks, but these introduce extra complexity into the project by accessing Canvas behind the scenes and present a high learning curve for the team.

Our final option, constructing a Scalable Vector Graphic, or SVG, solves the issues we had with HTML. SVG is a markup language based in XML used to encode vector graphics (*SVG*, n.d.). Like HTML, it uses tags to represent its elements and supports modification through JavaScript and CSS. Vector graphics are a type of graphic that, unlike most image types, do not lose quality and become grainy when their size is increased. This is because vector graphics store the shapes within an image and their attributes instead of the pixels that make up an image. As such, SVGs (SXA vector graphics) avoid the blurriness issue that plagues HTML Canvas. Like HTML without canvas, SVGs can properly capture events at an element level instead of a canvas level. Finally, SVGs are designed to handle rendering lines and more complex shapes because their primary purpose is rendering graphics, whereas HTML is more focused on creating an informative document. SVGs can be constructed either directly with JavaScript or with the help

of a library. We decided to prototype with a popular library, D3.js, that the team had experience with.

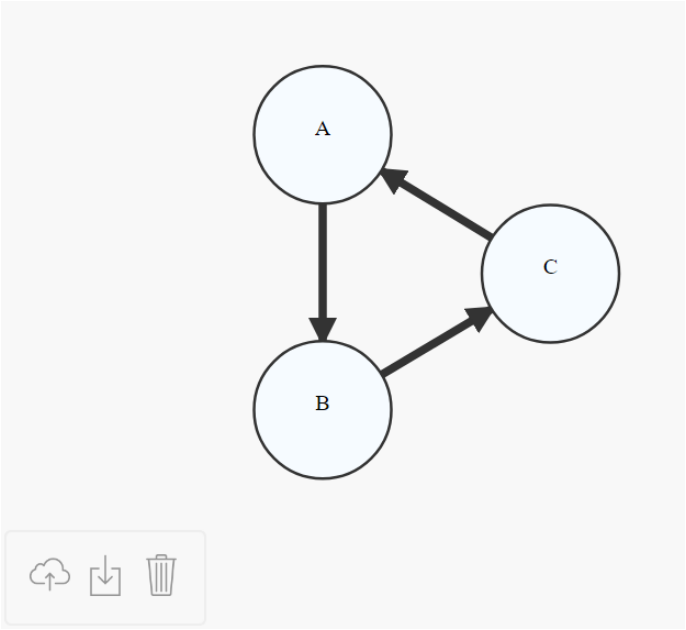


Figure 47: Example SVG graph editor (Interactive Tool for Creating Directed Graphs Using D3.Js., n.d.).

In order to quickly generate a D3.js graph editor prototype, we started with an example project that had similar features to the BoGL Desktop canvas area, pictured in Figure 47 (*Interactive Tool for Creating Directed Graphs Using D3.Js.*, n.d.). This example allowed the user to create and name circular nodes, draw edges between the nodes using shift plus click, pan, and zoom the graph area, and import or export graphs. We started by removing functionality irrelevant to the prototype, such as naming a node and importing and exporting graphs. While graph imports and exports would take place later in the development process, they had no bearing on the UI functionality of an SVG. Since we were prototyping system diagrams, we removed directionality from all edges.

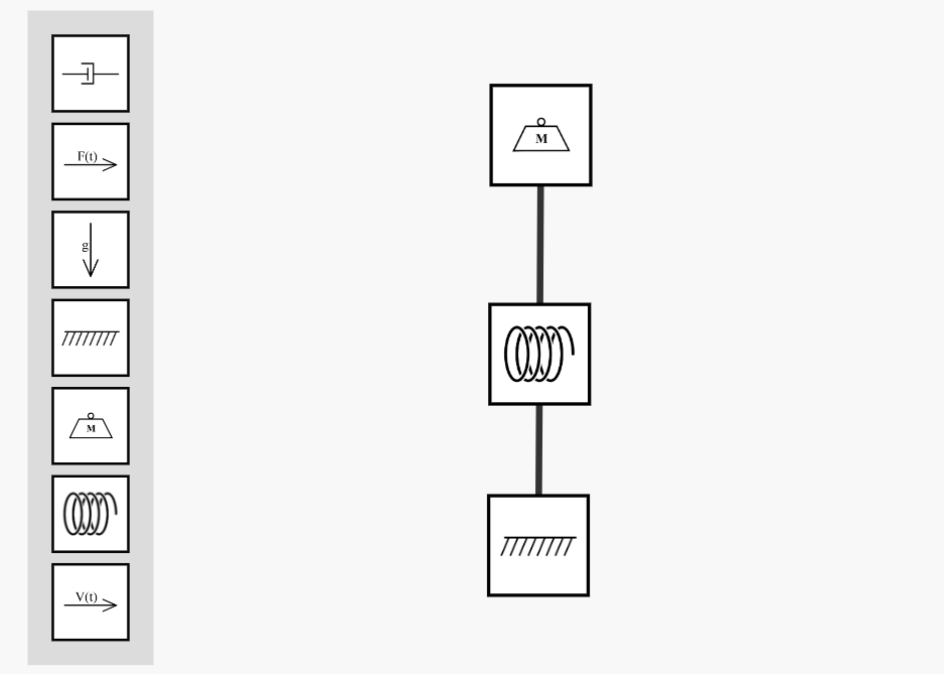


Figure 48: System diagram editor prototype.

Next, we made a mock version of the Ant Design element menu, shown in Figure 48. For this prototype, we created the menu within the SVG, putting the seven basic mechanical translation BoGL elements in squares and placing the squares in a rectangle off to the left. Once this was complete, we started transforming the circle nodes from the example code to look like parts of a system diagram, adding an image link to the underlying node data object to let different nodes represent different BoGL elements.

Afterward, we needed to connect the menu to these node elements by moving the node creation code triggered on shift click in the example to a mouse event on menu elements. To do this, we needed to determine what we wanted the transition from menu to graph area to look like. BoGL desktop uses a cursor change to indicate that an element is being dragged from the menu to the graph area. However, we wanted to experiment with a visible ghost image of the element being dragged to the graph area. We thought this might give the user a better idea that they were

adding an element and show them which element they were adding. Unfortunately, this idea proved difficult in the SVG-only system diagram editor and would have become even more difficult once an Ant Design menu held the elements. One issue was that drag events on menu items were being handled by the graph editor area behind the menu instead of the menu elements, triggering an item creation but then panning the graph editor on drag instead of moving the new element.

A more significant issue was having the dragged element appear above the element menu while being dragged, but below the element menu when the graph was panned around. In HTML, a CSS attribute called the z-index affects the stack level of a given HTML element, disrupting the default stack order. The stack order determines which element is visible when two elements overlap, where an element closer to the top of the stack appears above elements lower in the stack order (Lazaris, 2009). Z-index is already complex and can create strange results in HTML, but SVG does not support z-index, relying instead on element order to determine stack order, further complicating the issue of dragging a menu element. As such, after some experimentation we decided to drag elements out of the menu in the same way as BoGL Desktop, by changing the mouse cursor. The menu item being dragged was stored as a global variable and, on ending the drag in the graph area, a new element was created using this global variable.

While creating and trying out this prototype, we found that using D3.js made development easy and resulted in a demo that could replicate most of the features of the graph area. The D3.js example code we worked with was simple and already written, so it was easy to modify. We also observed no reaction delay in dragging elements, connecting elements, panning, or zooming. Additionally, we converted the BoGL Desktop PNG files to SVGs to ensure that

they would not become blurry when the user zooms in. Neither these element images nor the system diagram overall showed any issues with blurriness, regardless of zoom level.

6.5.3 Integrating the System Diagram Editor with the Element Menu

Once we prototyped the system diagram editor, we wanted to ensure that it could receive BoGL elements from outside the SVG graph area. In BoGL Desktop, the element side menu uses collapsibles and matches the styling of the rest of the UI, so we wanted to build the side menu with Ant Design. We started by building the Ant Design menu, which consisted of five collapsible sections: “Basic Mechanical Translation,” “Basic Mechanical Translation,” “Transmission Elements,” “Electrical,” and “Actuators.” Each collapsible section was made with an Ant Design Collapse component with custom styling to match BoGL Desktop’s collapsible sections. Since the menu contains many elements and their locations could change in the latest version of the menu, for this prototype we generated the element menu using JavaScript instead of hard coding each element to avoid writing repetitive HTML and to make it easy to edit the tags composing a menu element or the locations of their images. We started by giving each collapsible section a unique ID that matched a local image folder holding that section’s element images. Then, we got a reference to the collapsible object through the DOM using its ID and filled it with menu elements generated from the images in its corresponding folder. The completed menu prototype can be seen in Figure 49.

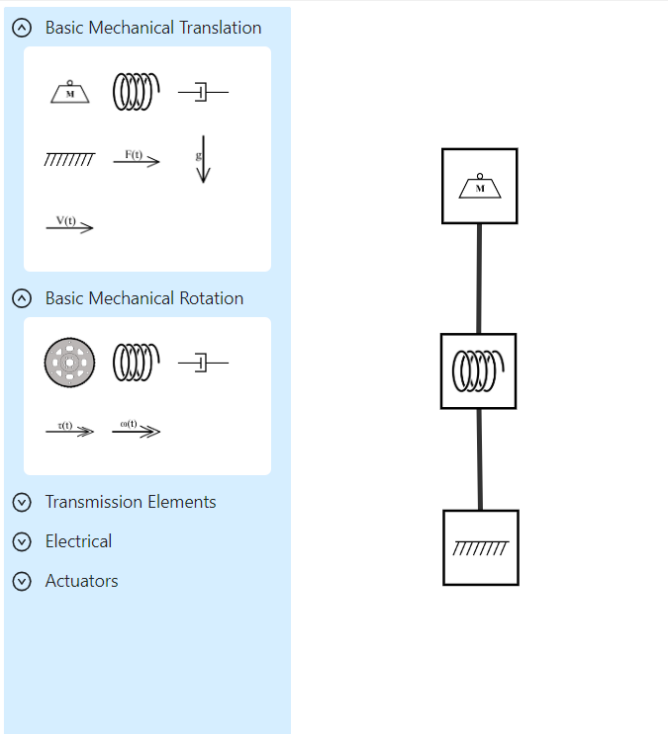


Figure 49: System diagram editor prototype with Ant Design menu.

Since we chose to represent a menu item being dragged as a global variable in the previous prototype, it was quite easy to connect this Ant Design Menu to the graph area SVG (SXA vector graphics) despite the menu elements being in HTML and not within the SVG graph area. Also, the menu item representation did not change the way dragging from the menu to the graph area worked, so the mouse event handler just needed to be added to each Ant Design menu element. Then, they could be dragged into the graph area just like their counterparts in the SVG menu were in the last prototype. This combination of Ant Design components and SVG proved to be quite easy to implement and confirmed that UI elements outside of the SVG can interact with the SVG graph area.

6.5.4 Dragging Edges Prototype

Our next UI prototype was replicating BoGL Desktop's edge creation in BoGL Web. This required having a green dot follow the mouse around the edge of each element in the graph editor and triggering edge creation on clicking or dragging out of the edge of an element. Figure 5 shows a system diagram element with red boxes, which do not appear in the UI, representing click regions. When the cursor is within the area between the dotted red lines in Figure 50, a green dot should appear. Clicking or dragging within this area should trigger an edge to follow the user's mouse, ready to be placed in another element or dropped by clicking anywhere that is not another element. If a user clicks inside the smaller red box the element should be selected, and a drag starting in this area should move the element.

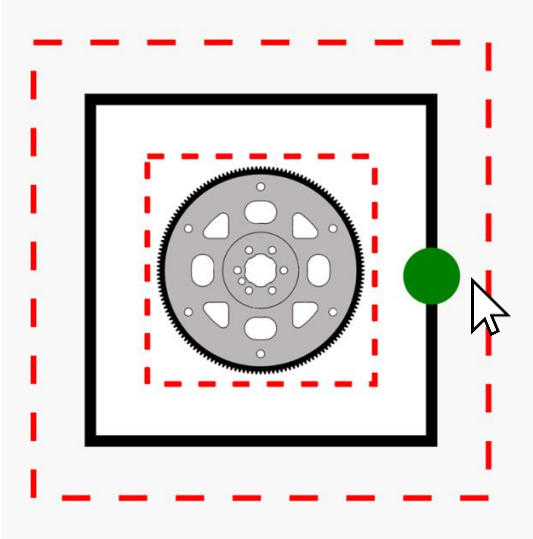


Figure 50: Green circle following mouse tip on movement plus outline of mouse interaction area.

We started by implementing the green circle on hover, which was challenging for two reasons. The first challenge was extending the element object beyond its black border to the

outer red border to detect the user hovering on the edge. Mouse events for the inner region, such as selection on click and movement on drag, were applied to the image tag. All other mouse events for the element, like edge creation on click or drag and showing the green dot on hover, were applied to both squares, making the squares behave like one region. The second challenge of this process was making the green dot appear at the closest point to the mouse along the edge of the element. To do this, we diagrammed the square and determined a formula for each quadrant of the square to convert mouse coordinates to the nearest point on the edge of the square. Once we found these equations, we were able to implement them in the front end so the green dot would follow the mouse.

$$(x, y) = \begin{cases} (s, s\tan\theta) & 0 \le \theta < \frac{\pi}{4}, \frac{7\pi}{4} \le \theta < 2\pi \\ (s\cot\theta, -s) & \frac{\pi}{4} \le \theta < \frac{3\pi}{4} \\ (-s, s\tan\theta) & \frac{3\pi}{4} \le \theta < \frac{5\pi}{4} \\ (-s\cot\theta, s) & \frac{5\pi}{4} \le \theta < \frac{7\pi}{4} \end{cases}$$

The next step of this prototype was enabling edge creation. BoGL Desktop has two methods for creating edges. One method is dragging out from the border of an element and ending the drag in another element and the other is clicking within the border of one element then clicking on another element. If the first action of either of these sequences is completed, the user should see an edge line following their cursor, and the edge should be added to the graph if the second action is completed. If only the first action in the edge creation sequence is completed but the user does not click or drag into another element, the edge following the user's cursor should disappear. We attempted to add edge creation to the previous prototype after enabling

green circle hover and succeeded in enabling edge creation through dragging but ran into several state errors when we attempted to add edge creation through clicking to this. This difficulty let us know that edge creation is a complex enough action that we should make an individual state diagram for it to properly implement the feature in BoGL Web (see Figure 82 in Section 8.2.3.6).

6.5.5 Tab Prototype

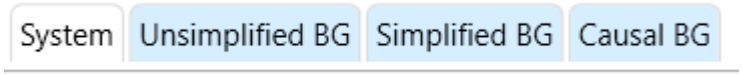


Figure 51: BoGL Desktop tabs.

The last UI prototype we created was to test Ant Design tabs to make sure that they do not cause errors in panning or zooming, that they keep the state of elements inside them, and that they can be styled to match BoGL Desktop's tabs. Figure 51 shows the appearance of BoGL Desktop's tabs. The Ant Design card style tabs shown in Figure 52 were close enough to the BoGL Desktop tab style that it was easy to match BoGL Desktop's tab style with some custom CSS, as seen in Figure 53.

Figure 52: Ant Design Blazor card style tab.

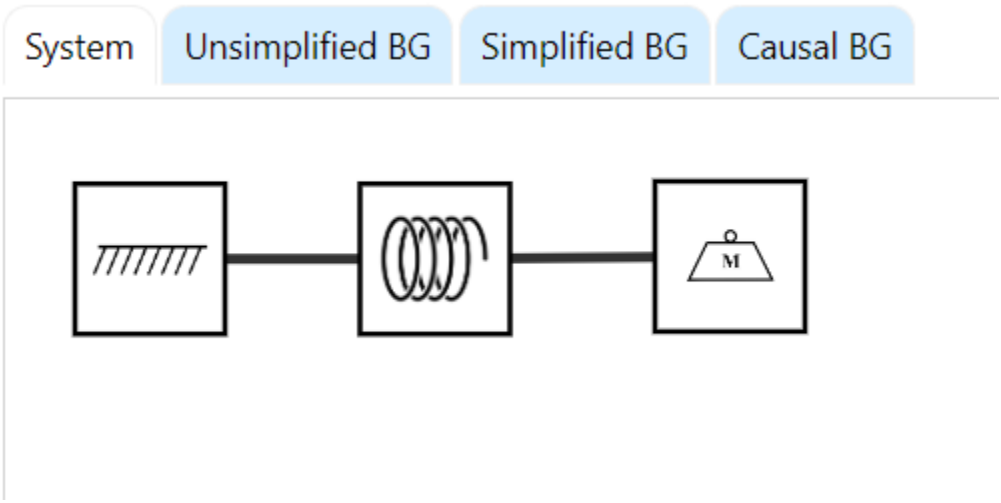


Figure 53: BoGL Web prototyped tabs.

After making the tab prototype, we were also able to confirm that the tabs held state between switches. For example, if a user constructed a system diagram in the system diagram tab, switched to one of the bond graph tabs, and switched back to the system diagram tab, by default its content would remain unchanged. We also confirmed that putting the system graph editor in a tab did not change zoom, pan functionality, or cause errors.

6.6 Development Methodology

In this section, we will discuss our methodology for developing BoGL Web. We discussed the tools that we used as well as the order in which we added features.

6.6.1 Tools Used

In this section, we will discuss the software tools that we used to create BoGL Web. This includes tools like Jira, which we used to organize tasks, GitHub, and Git, which we used as a version control system, Visual Studio and Rider, which we used as our Integrated Development Environments, and Azure and GitHub Pages, which we used for deployment.

6.6.1.1 Organization Tools

We chose to use Jira for tracking tasks and issues throughout our project (Atlassian, n.d.-a). This is a tool which is widely used in creating commercial software development projects. We chose this tool because it can integrate with GitHub (see Section 6.6.1.2) and can also track deadlines for each task. We were able to assign tasks to each team member and manage the progress we had made on each task. Jira works by creating tickets for each task and organizing them into distinct categories. We had several categories for tickets including a category for tasks that need to be done, tasks in progress, and completed tasks. This setup can be seen in Figure 54. We were able to automate moving tickets between in-progress and done by closing pull requests using Jira's integration with GitHub.

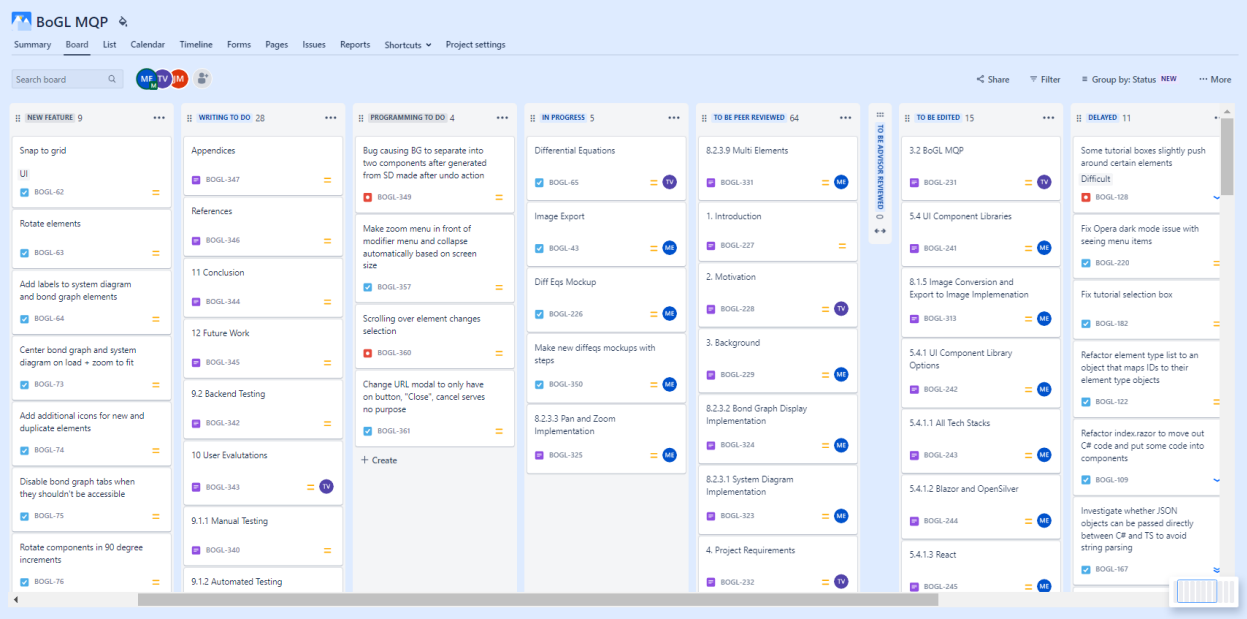


Figure 54: Jira Board for BoGL Web.

6.6.1.2 Version Control

Version control is software used to keep track of changes made to a project. We used Git for version control and GitHub to host our repository (*Build Software Better, Together*, n.d.; *Git*, n.d.). These tools allowed our team to collaborate on the project and ensure that changes did not cause conflicts within the project. We were able to implement features independently and then bring them together while checking for issues along the way.

6.6.1.3 Integrated Development Environments

We used two different integrated development environments (IDE) when developing BoGL Web. The main IDE used by our team was Visual Studio. This IDE is made by Microsoft and Blazor was built to work with it. We used this environment for the initial setup of the

project, and throughout the rest of the development process. It includes a debugger as well as Git integration.

Our team also used Rider, which is a product created by JetBrains (*JetBrains*, n.d.). Members of our team used this IDE during the development process due its numerous available plugins, including IdeaVim and Azure Toolkit for Rider. IdeaVim changes the key bindings within the IDE to match those in the text editor VIM (*Welcome Home : Vim Online*, n.d.). Azure Toolkit for Rider was used in the process for deploying the application to Azure. This made the development process easier since the IDE was more customizable. Both environments worked well and were able to aid in completing the MQP.

6.6.1.4 Debugging

We used two main debugging methods for finding and solving issues in BoGL Web. Most of our debugging was done with Chrome DevTools, a set of developer tools built into the Chrome browser. These tools include an output console, a debugger that can stop on breakpoints in the TypeScript code, and an element panel that shows all DOM elements on the page. The console can receive print commands from the C# code, but breakpoints cannot be set in the C# code since it is compiled. The other debugging tool we used was BrowserStack, a paid tool that allows a website to be tested on Windows and Mac with different browsers. This application, which runs entirely in the browser, lets us find and fix bugs on other browsers and was especially useful for testing Safari and other Mac browsers. BrowserStack allows users to test local builds on other browsers and operating systems, which is incredibly helpful for testing code without having to deploy it. See Section 9.1 for more details about how we tested the UI with BrowserStack.

6.6.1.5 Deployment

In this section, we will discuss the tools used to deploy the website. These tools make the website accessible to the public by providing a server which can serve up the data needed to run the website. We started by using Azure Static Web Sites as a hosting platform (*Create Your Azure Free Account Today | Microsoft Azure*, n.d.).

Azure is a tool created by Microsoft that can be used for many tasks, including hosting websites. We started by using Azure because Blazor is also built by Microsoft, and they provided several tutorials (guardrex, 2023) with steps on how to upload the webpage to the Azure hosting platform. We were able to deploy and access the website through this hosting platform easily; however, we did struggle with deploying rules, rulesets, and examples. To solve this issue, we transitioned to using GitHub Pages, another hosting platform for these files (*GitHub Pages*, n.d.). Once setting this up, we had our website working on Azure. At this point, we were hosting the website using the Azure student license which provides a limited $100 credit. We quickly found that Azure was charging a substantial amount and would use all the credit in a month and a half.

Due to the prohibitive cost of Azure, we determined that it would not be a viable solution in the long term. Since we were already using GitHub Pages to host rules, rulesets, and example files, we investigated hosting the webpage there. This process was simple in the end and involved following a tutorial (Swimberghe, 2020). This platform is free to use and provides us with everything we need to host the website.

We used GitHub Actions to automate deploying the website. This tool takes our repository, builds the code, and uploads the project to the hosting platform we are using. The process for setting this up varied depending on the hosting platform, but both GitHub Pages and Azure required a moderate amount of setup.

6.6.2 Development Order

In this section, we will explain why we developed features in the order that we chose.

6.6.2.1 Backend

The primary goal of the backend was to be able to process system diagrams and generate bond graphs. The order in which we implemented features reflects this goal with tasks that support processing system diagrams coming first, followed by the other features.

We started by developing a method of loading and storing rules and rulesets. We developed this feature first since the entire backend was reliant on an efficient method of accessing these rules and rulesets. We first moved the GraphSynth library for loading rules and rulesets into our project. We then created a class which would be used to store each rule and ruleset, and finally worked on loading the rules and rulesets from a file.

The next feature we added was the ability for the backend to store bond graphs and system diagrams. This feature was also required to be able to be able to process system diagrams into bond graphs as we need a way to store both the input and output to this function.

The next feature needed to achieve our goal was a way to load a system diagram since we would need a way to test our implementation of system diagram to bond graph generation. To be able to do this we would need to be able to load example system diagrams as well as those which had been created on the front end. Naturally, this feature came next.

The last feature we needed to be able to generate bond graphs from a system diagram was a way to translate our system diagram storage implementation to one that could be processed by GraphSynth. This was needed so we could access functions provided by this library, which helps generate the bond graphs.

Now that we had all these features, we were able to work on generating bond graphs from system diagrams. With the completion of this feature, we accomplished the major goal of the backend.

Now that we had a system diagram on the backend, we needed a way to translate it into something accessible to the front end. This was the next step in our development. We also needed a way to create a reasonable layout for the bond graph. The conversion leaves us with no positions for the vertices in the bond graph, so we implemented a method to place these vertices.

At this point, we were able to implement the remaining features from BoGL Desktop. These features included save and save as, error messages, and undo/redo. We also implemented generating URLs for system diagrams and started working on state equations. All these features did not have the same prerequisites as the previous features, so we were able to implement them in any order. The order we chose was driven by which tasks were easier, and what tasks were needed to match BoGL Desktop.

6.6.2.2 Frontend

Development for the front end started with putting the major static elements of the page in place. These elements include the top menu bar, the element menu on the left side of the screen, and the four graph tabs. The static elements were developed first to define the size and placement of the area that would hold the system diagram and bond graph SVGs. These sections were also easy to start with since they were based on Ant Design components and not SVG construction. Some Ant Design components, like the modifier and zoom menus, were created later because they are contained within a tab. The element menu was important to create early on because adding elements was the first graph functionality we added, so we needed the menu to support the feature.

Next, we implemented simple system diagram functionality such as adding and deleting elements, adding, and deleting edges, and displaying a system diagram. We started by adding edges using shift + click because we were building off a tutorial that added edges that way and because it was simple. Afterward we transitioned to adding edges through clicking and dragging, in the style of BoGL Desktop. At this point we transitioned the system diagram code into a parent graph display class and a system diagram class, which allowed us to start developing the bond graph display. Most of the code to display bond graphs was written at this point since they are not as difficult to display as system diagrams. This included creating text nodes, line endings for edges, and refactoring edge ends to end at the edge of a text node.

After establishing a codebase for drawing both types of graphs, we turned our focus to the modifier and zoom menus. The modifier menu was made at this point so we could connect it to elements afterward, allowing a user to edit modifiers. The zoom menu is a similar element to the modifier menu, so it was developed at the same time out of convenience. During this time, we also developed the ability to open BoGL files, display loaded system diagrams nicely, and generate and display bond graphs. This functionality fit naturally after we finished the basic ability to display system diagrams and bond graphs, since we could display what we uploaded and use the upload ability to continue developing the graph displays.

After this we added the ability to edit modifiers and velocities on elements and bonds since the modifier menu was finished. Another major feature, multi-select, required system diagram and bond graph displays to be finished so there could be nodes and edges to select. Given this, multi-select was developed during this period. The development of multi-select enabled us to add more functionality that relied on the current selection, such as cut, copy, paste, and select all.

At this point, the backend for undo/redo was mostly finished, so we developed UI functions to implement undo/redo changes and put in triggers to record changes in undo/redo objects. This required a lot of bugs fixing and testing, which encouraged us to fix some other major bugs, such as the Ant Design bugs mentioned in Section 8.2.2.1. Looking for bugs led us to start working with BrowserStack, which furthered our ability to find and fix bugs. Finally, we developed image export functionality last because we had identified significant challenges with it early in the project and we did not consider it a high priority feature since users can take their own screenshots of graph displays.

7 Design

In this section, we discuss the rationale behind the design of various elements of BoGL Web. These include the undo/redo system, state equation system, and tutorial.

7.1 Undo/Redo Design

One change we made from the BoGL Desktop was implementing a full undo/redo system. In BoGL Desktop, undo/redo recorded only the most recent change and, on recording a change, saved the entire system diagram. Instead of storing each latest version of the contents of the canvas, undo/redo version control systems typically record only individual modifications. We designed the BoGL Web undo/redo system similarly.

For BoGL Web, we decided to maintain an undo/redo stack for each of the four graph tabs, so that undo and redo affect only the current tab. The undo/redo stacks recognize six actions for system diagrams: “Add Selection,” “Delete Selection,” “Change Selection,” “Change Selection Modifier,” “Move Selection,” and “Change Selection Velocity.” Of these, only “Change Selection” and “Move Selection” are also recorded in bond graph tabs, as the user can only select and move elements in these tabs. All six actions inherit from a single change class that stores element IDs and supports the overridable method that executes each change on the system diagram in the backend.

7.1.1 Undo/Redo Action Types

Here we discuss the various actions which will add a change to the undo/redo stack. These actions include adding elements, deleting elements, changing modifiers, moving elements, modifying edges, and modifying velocities.

7.1.1.1 Add Selection

The *add selection* action is recorded whenever elements or edges are added to the canvas. This occurs when a system diagram is loaded from a file, the example menu, or a URL, and when an element is dragged into the canvas, or an edge is created. When this action is recorded, a JSON (JavaScript Object Notation) object representing the action is created and passed to the C# backend. This object contains a list of JSON objects representing the new elements and/or edges, a list of IDs for previously selected elements, and a list of JSON strings representing previously selected edges. The new elements and edges are needed to reconstruct the elements on redo or remove them on undo. The other two lists, which track the selection before the items were added, allow the correct elements to be re-selected on an undo. This is necessary because adding a collection of elements and/or edges to the canvas selects the new elements and edges, so returning to the previous state requires a record of the previous selection. In the backend, the JSON objects are parsed into system diagram element and edge objects.

7.1.1.2 Delete Selection

The *delete selection* action is recorded when edges and/or elements are deleted. The JSON change object stores two lists of JSON strings: one list of deleted elements and edges that were selected before deletion, and one list of edges that were deleted without being selected. This second case only happens when one of the elements at the end of an edge is deleted,

necessitating removal of the edge as well even though it is not selected. When a deletion is undone, all deleted elements and/or edges are returned to the canvas and those from the first list are selected.

7.1.1.3 Change Selection

The *change selection* action is recorded whenever the user changes the selected set of elements and/or edges by clicking on or dragging over a set of elements. The user can change the current selection by clicking an edge or element, by making or modifying a multiselection, or by clicking the canvas to clear the current selection. Notably, this only includes selections done by the user that are not tied to another action, such as the selection changes that occur on adding or deleting elements on the canvas. These automatic selection changes are coupled with the user action that caused them so that the number of undo/redo steps occurring matches the number of actions a user completed. For example, adding elements to the canvas is a singular action of dragging or clicking, so it is recorded as an add selection change without a change selection change.

When the user changes the selection set, the elements and edges being added to the selection and the elements and edges being removed from the selection are recorded. Specifically, four lists are stored: a list of elements being added, edges being added, elements being removed, and edges being removed. The element lists are a list of integer IDs while the edge lists hold JSON objects representing which element IDs are at the end of a given edge. The elements and edges are separated to make storing and rendering the undo/redo changes easier. One other piece of information stored on a selection change is the current tab ID, since elements and edges in bond graphs can be selected to be moved individually or as a group.

7.1.1.4 Change Selection Modifier

The *change selection modifier* action is recorded when a checkbox in the modifier menu is selected, since the modifier menu is the only way of affecting element modifiers. This action records the IDs of affected elements, the modifier ID, the new modifier value, and a list of previous modifier values for the element IDs listed. The element IDs come from the current selection stripped of its edges, since edges cannot receive modifiers other than velocity, which is edited separately. The modifier ID indicates which modifier checkbox was clicked by referencing the modifier's index in a universal list of modifiers. The boolean modifier value refers to whether the checkbox is being selected or unselected. If some of the selected elements have a given modifier but others do not, the checkbox will show a partially filled, intermediate state. Clicking the checkbox in this state will always result in the checkbox being unselected, even if some elements in the selection can have the given modifier but do not have it yet. When the checkbox is clicked while unselected, the modifier value is false and otherwise is true on click. This value indicates whether the given modifier should be removed from or added to selected elements.

Removing modifiers is straightforward but adding them is a bit more complex. Each element in the selection must be checked to ensure it is compatible with the modifier that is being changed because each element type only supports certain modifiers. If it is not compatible, the element is ignored, so that the modifier is only added to elements it is compatible with. The previous modifier value list is a list of booleans that indicates whether each element in the selection had this modifier enabled before the modifier change was applied. This is important because you could have several elements selected that are all compatible with a modifier, but only half of them have the modifier currently selected. If you then remove the modifier from all

of them and undo your action, the system needs to know to which elements to re-apply the modifier.

7.1.1.5 Move Selection

The *move selection* action is recorded when the user drags the current selection to a new location on the canvas. The action stores the IDs of elements in the current selection, an x offset and y offset representing the change in position of the elements in the canvas, and the current tab ID. This action only considers element IDs, excluding edges, because edge positions are entirely determined by the position of their respective elements, so which edges are included in the selection is irrelevant to this action. The x offset and y offset are decimal numbers representing the total x and y distance that the selection traveled on the canvas. When moving a selection, the entire selection moves the same distance. The tab ID is included for the same reason as the change selection action, because this action can occur on bond graph tabs and the system diagram tab.

7.1.1.6 Change Selection Velocity

The *change selection velocity* action occurs when one of the velocity buttons or the Clear Velocities button is clicked. This action stores the element IDs as integers, the edge identifiers as JSON string, the velocity ID, and the previous velocity values for the element and edge lists, ordered as the element list followed by the edge list. Unlike modifiers, velocities do not need a value because a velocity ID of 0 represents no velocity, which velocity buttons return when they are unselected. As with modifier changes, the previous velocity values are used to restore the states of elements and edges in the selection after undoing a velocity change in the system diagram.

7.1.2 Canvas Change

CanvasChange is a parent class that encompasses all six undo/redo action types. We created such a structure to allow more streamlined access to each edit object in the backend. When the user makes changes in the canvas, it is easy enough for the backend code to choose the appropriate action type from the six listed in the previous sections for storing the change. However, when undoing or redoing a change, the program needs to know what kind of object is storing the change.

7.1.3 Undo/Redo Stack Design: “EditionList”

In this section, we discuss the data structure used to track changes in BoGL Web. This undo/redo stack in BoGL Web is an *EditionList*, a modified doubly linked list with a built-in node container. Each node stores a pointer to the *next* node, a pointer to the previous (or *prev*) node, and a nullable data payload. The stack is initialized with one node containing a null object. In this state, an *EditionList* is considered “empty” and has *size* zero. An *EditionList* with size n then has $n + 1$ nodes. In addition to the null node at the head of the stack, the object stores a pointer node that tracks one’s current position in the list and an *index* field that stores the numerical offset of the pointer from the null head. This location can be moved toward the head via the *prev* method and toward the stack top with the *next* method.

When an outside function calls for the pointer to move to the next node, the *EditionList* checks the current pointer to make sure the *next* node is not null. If it is, the pointer is at the end of the list and does not move; otherwise, the pointer will be set to the node indicated by its *next* field and the *index* will increase by one. When a function calls the *prev* method, the program checks that the payload of the pointer node is not null. If it is, the pointer is on the null head node

at the beginning of the list and is prevented from moving forward; otherwise, it is set to its *prev* field, and the *index* decreases by one. The *hasNext* and *hasPrev* methods check separately that calling *next* or *prev* is valid without moving the pointer.

The only methods that change the length of the stack after initialization are the *clear* and *add* methods. The *clear* function sets the *size* to zero, the *index* to -1, and the *next* element of the null head to null to dereference any non-null nodes. It also moves the pointer to the head. The *add* method takes in a single payload object. It locates the pointer in the list and sets its *next* field to a new node with the indicated payload, effectively removing all nodes after the pointer and establishing the new node as the new end of the list. It also increments the index by one, moves the pointer to the new node, and resets the *size* field as $index+1$.

7.2 State Equation Design

One major feature in BoGL Web that was not implemented in BoGL Desktop was the state equation generator. While some backend design had been considered for BoGL Desktop, significant changes were made in the generation algorithm to integrate with the rest of the BoGL Web system. We also made mockups of designs for a state equation user interface. In the following sections, we detail the reason for rebuilding the generator from scratch. We also provide the interface mockups that we considered when designing user interaction with the generator.

7.2.1 Backend Design

While BoGL Desktop included code that handled state equation generation, this code was not complete, nor was an interface implemented in the system. We decided to implement equation generation differently for the following reasons:

1. *The code is relatively undocumented and unintuitive.* Because the code has few comments, it is difficult to determine how some parts of the program work. Tracing the program to auxiliary methods does not help this situation. It appears that many of the methods load in at runtime, which means they are not visible in the editor during development. The description of this algorithm in the AVL report did not provide enough detail for us to decipher the purpose of each section of code, as it only expanded upon the generation process in section 3.5 by claiming to incorporate a depth-first search (Grande & Mancini, 2016).
2. *The code is too long and does not break out functionality well into methods.* The entire generation process occurs in a single method with more than eight hundred lines of code. From what we could tell, the method both generates the equation strings and adds some form of equation display to the general BoGL Desktop interface. These two roles can and should be split into different methods to maintain good coding practice and readability.
3. *Significant changes to the project file structure have already been made that make using the desktop code far more difficult.* At least one file and several global variables existing in BoGL Desktop that are necessary for the generation algorithm do not exist in BoGL Web. In the desktop version, many of these essential variables are modified in other methods, and those changes cannot be replicated easily.

4. *We have designed a more efficient method for assembling a function.* Part of the desktop generation algorithm reconstructs a GRXML rule file line by line. Setting aside the computation complexity of parsing a new file every time equations are generated, constructing a String object adds a lot of unnecessary time to the program and should be avoided when possible.
5. *The old code is hard to test.* The BoGL Desktop code runs all subtasks of the equation generation at once. A programmer would only have two options for verifying the reliability of the algorithm. The first would be to get all components of the problem working simultaneously without verifying each subtask individually. This is impractical and makes it difficult to narrow down the cause of errors. The second would be to comment out sections of code for individual testing, at which point splitting the method into smaller methods would be easier anyway.
6. *The old code generates equations for bond graphs in only the mechanical and electrical domains and would be hard to generalize.* If BoGL is extended in the future to support thermal, hydraulic, or other domains, then the generation algorithm would have to be rewritten to accommodate new kinds of object models. Moreover, the desktop algorithm forces the variables in the equations to conform to a specific format. This conflicts with our plan to allow students to pick custom variable names.
7. *The new code would have the same generation results as the old algorithm.* The new algorithm we designed for equation generation would provide equations and all intermediate steps in the same general format as the old algorithm. We have determined that there is no foreseeable drawback to rewriting the equation generation code using the new design.

The generation algorithm we implemented is described in Section 8.1.7.

7.2.2 State Equation UI Mockups

To determine what BoGL Web's state equation UI should look like, we made mockups of some options for creating and displaying the equations. A mockup is a model which can be used for construction of a full system later. These options were then rated using a decision matrix. We will begin by discussing the requirements for these mockups. We will then present the mockups created for adding labels and values to a system diagram, viewing differential and state equations, and viewing labels. Lastly, we will present a series of decision matrices which we used to analyze these mockups, as well as a set of recommendations detailing the mockups that should be implemented and some slight modifications that should be made. It is important to note that these mockups are not necessarily meant to be distinct, but rather each exist to solve a specific problem and can be combined to create a complete system.

7.2.2.1 Requirements

We started by developing a set of requirements for the state equation mockups. We created these requirements in conjunction with Professor Radhakrishnan who teaches ME/RBE 4322, Modeling and Analysis of Mechatronic Systems. These requirements are as follows:

- The user should be able to:
    - Read the equations easily.
    - Easily distinguish between the set of equations and the bond graph labels.
    - Easily distinguish between equations corresponding to different nodes.
    - See the whole equation, such that the equation is either fully visible immediately or viewable through scrolling.

- Be able to see the rest of the UI without the equations obstructing the view or be able to easily dismiss the equations from view.
- View only the equations for the currently visible causal bond graph.
- Be able to see the steps required to find the final state equation.

These requirements ensured that we were creating mockups that best suited the users' needs and would be able to help students taking ME/RBE 4322 most effectively. In addition to these requirements, we considered the possibility of letting students copy equations from BoGL Web and paste them into calculator software to solve them. The copying could be done manually by students or could be handled automatically by BoGL Web.

7.2.2.2 Mockups for Adding Values and Labels

We will first introduce the mockups our team created to allow users to add labels and values to elements and modifiers. The user will input labels as strings and values as a number with some associated units. All our mockups use text boxes for inputting the label and value, and a drop-down menu for the units. Text boxes make the most sense for labels and values since they are the standard input component for strings and numbers on the web. A drop down is used for units so the user can select from all units which they might want to use but cannot select units which do not fit the current energy domain, which would cause errors.

The first mockup we present can be seen in Figure 55. In this mockup, we create a modal which will allow users to enter a label, a value, and units for the value. This modal is triggered every time an element is added to the system diagram. There are several advantages to this idea. The first is that it mandates that a user adds a label and value to every element or approves the default value. This modal is also triggered while the user is thinking about the element that they just added to the screen. This means that the user will not need to continuously shift their focus

between adding elements and labels, and it prevents the user from needing to learn how to add labels. This is important for inexperienced users since knowledge of the label system is important for state equation generation.

One issue with this mockup is that more experienced users may find the pop up frustrating as it continues to appear. It is reasonable to assume that more experienced users will want to add all their elements to the canvas first, so they can understand the entire system diagram, then move on to adding labels. One way to fix this issue is by creating a check box. This would be like what currently exists for delete multiple elements, which will disable this automatic popup. This mockup also does not allow users to modify the labels they have assigned, since the popup is only triggered when the element is first added to the canvas. To change the label or value assigned to some element, the user would need to delete the element from the canvas, then re-add it and assign a new label or value. Lastly, it is worth noting that this mockup does not address labels and values for modifiers.

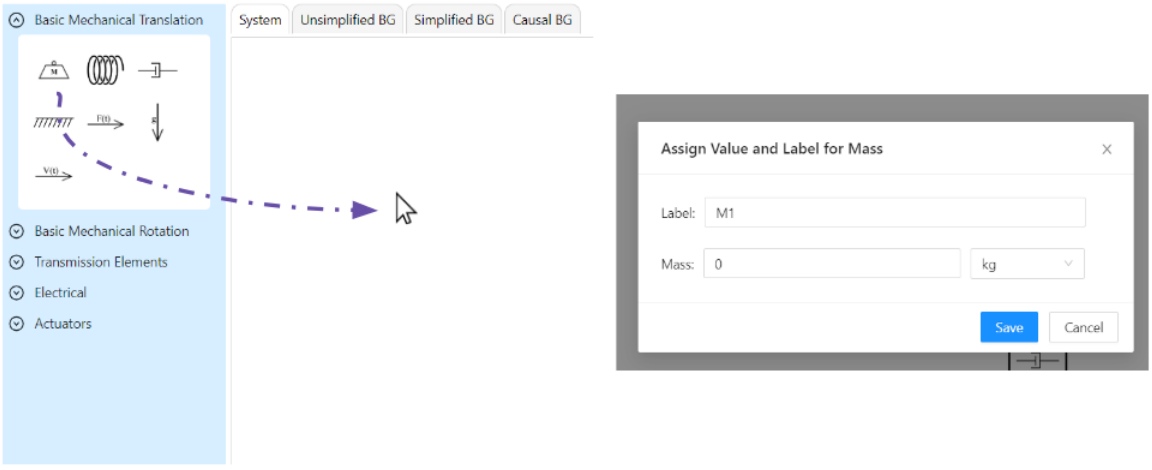


Figure 55: Mockup where label/value modal opens when element is added to canvas.

We will next evaluate a mockup which uses the same modal as in the previous mockups but is triggered with the right click action (see Figure 56). The main benefit to this mockup is that it does not interrupt the user by asking them to add a label and value every time they add an element to the canvas. This will allow users to first focus on creating their system diagram, then focus on adding labels afterward. While this may not be a typical workflow for inexperienced users, the more experienced users will operate like this. This mockup requires the user to know that they must right click on an element to change the label and value for that element. This could be included in the tutorial, but it is still an added step which may not be intuitive to all users. Finally, this mockup does not include a way to subsequently edit labels and values for modifiers.

Figure 56: Mockup where the user right clicks to add a label/value.

As stated previously, several mockups could be combined to create a better experience for the user. For example, the previous two mockups could be combined so users are forced to

add/approve labels and values for all elements they add but can also edit their labels by right clicking on the element.

The next mockup changes the modifier menu by adding buttons which allow the user to open the label/value modal for a modifier (see Figure 57). This mockup uses the same modal as the previous mockups. The user can open this modal by selecting an element in the system diagram then selecting one of the modifier gear icons. The main advantage of this mockup is that it provides a way to add labels and values for modifiers. One disadvantage is that it does not provide a way to change labels/values for an element, which can be remedied by combining this mockup with another approach.

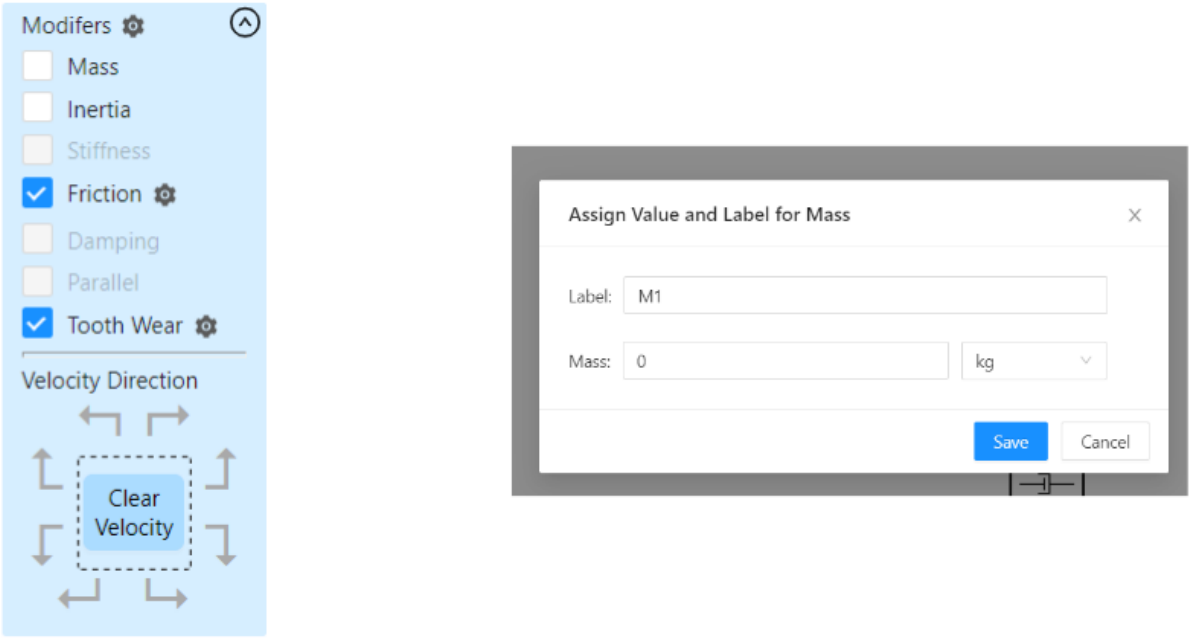


Figure 57: Mockup that allows the user to change labels/values for modifiers.

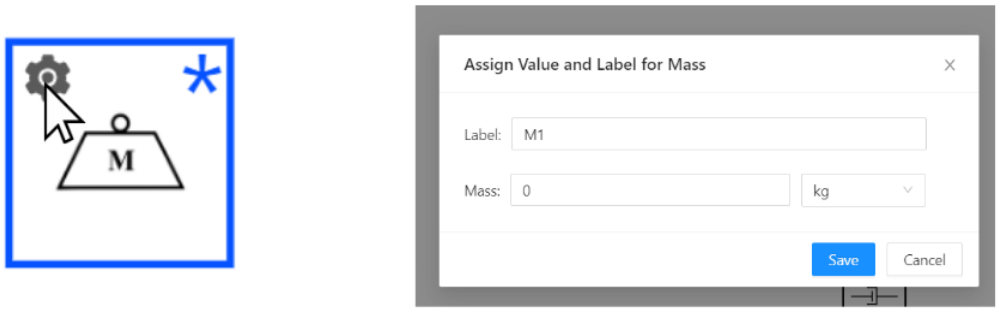


Figure 58: Mockup with settings button for adding label/value on element.

Another option for assigning labels and values to an element would be to place a settings button on the element itself, as shown in Figure 58 with the gear icon. Clicking the settings button will open a modal that lets the user edit the labels and values for the element. The benefit of this approach is that the menu button is visually obvious and easy to access. It also clearly associates the settings button with a particular element as the button is visually attached to the element. One disadvantage of this design is that the settings button would appear on every element, taking up a lot of space in the UI, which distracts the user and makes the system diagram more difficult to interpret. Another downside is that allowing the user to click a button within an element would disrupt the natural click actions of the system diagram display, so the element's click actions would change and become more complex.

The final mockup we present does not use the modal, but rather adds a menu next to the modifier menu that users can use to add labels and values (see Figure 59). This menu will modify the element which is currently selected in the canvas. There are several advantages to this design. The first is that it does not bring up windows which may interrupt the user's workflow. The user can visually see that they are able to add values and labels, but they are not required to until they are ready. This menu will also collapse when no element is selected. This means that it will not

be as intrusive to the UI as if it were constantly open. Since elements are automatically selected, the menu will automatically open, but not obstruct the user, when an element is added to the graph.

While this mockup does not directly allow for a way to add labels and values to modifiers, this feature could easily be added. One way to do this is by adding another instance of the Labels and Values box below the current one each time a new modifier is added. This would allow for the user to see all the labels and values associated with a single element when it is selected. However, this would create a lot of clutter in the UI. One way that this could be resolved is by making this list of Label and Value menus an accordion menu. This would make it so the user would be able to see a list of all the possible labels and values they would be able to add, but only one of the menus would ever be open at a time. Since the list of names is much smaller than a list of the menus, this modification might reduce the clutter added to the UI.

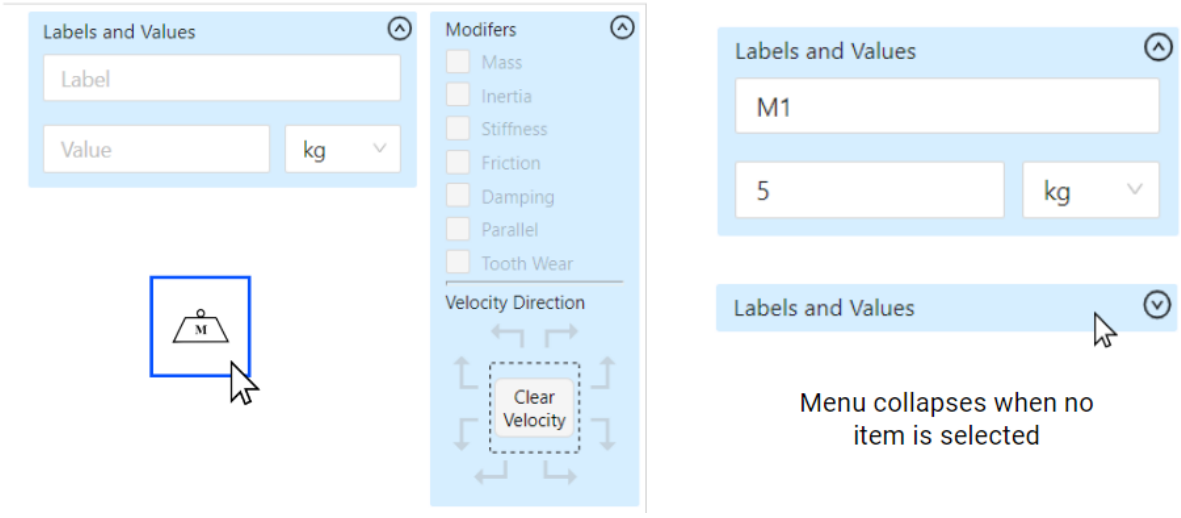


Figure 59: Mockup that allows a user to change labels/values when an element is selected.

Now that elements have labels and values, we can move on to looking at methods of displaying state equations and state equations to the user.

7.2.2.3 Mockups for Viewing State Equations

In this section, we will discuss several mockups which can be used to display state equations to the user. These mockups are required to display equations to the user in a readable manner. Since one of the requirements of the state equation system is to show steps, it must be clear to the user what these steps are and what the final solution to the problem should be. Since the images in this section are just mockups, the equations are just text strings. For a production version of these features, we would recommend that the developer format them so that they are written more clearly. This would include putting fractions on top of one another, including subscripts, etc. This could be done using a typesetting system such as LaTeX and a library that could render the LaTeX equations.

The first mockup we present for displaying state equations can be seen in Figure 60. This mockup uses IntroJS, the library we use for our tutorial, to highlight elements of the bond graph and display the state equations and state equation for that element. One of the main advantages of this mockup is that it is clear to which element of the bond graph the state equations correspond. However, it is not necessarily intuitive to the user how to trigger this view for a given element. This mockup also obscures part of the bond graph by both darkening the screen in addition to the pop-up box covering some elements. In an actual implementation, this popup box would be large due to including the steps for finding the final state equation so it would obscure even more of the screen.

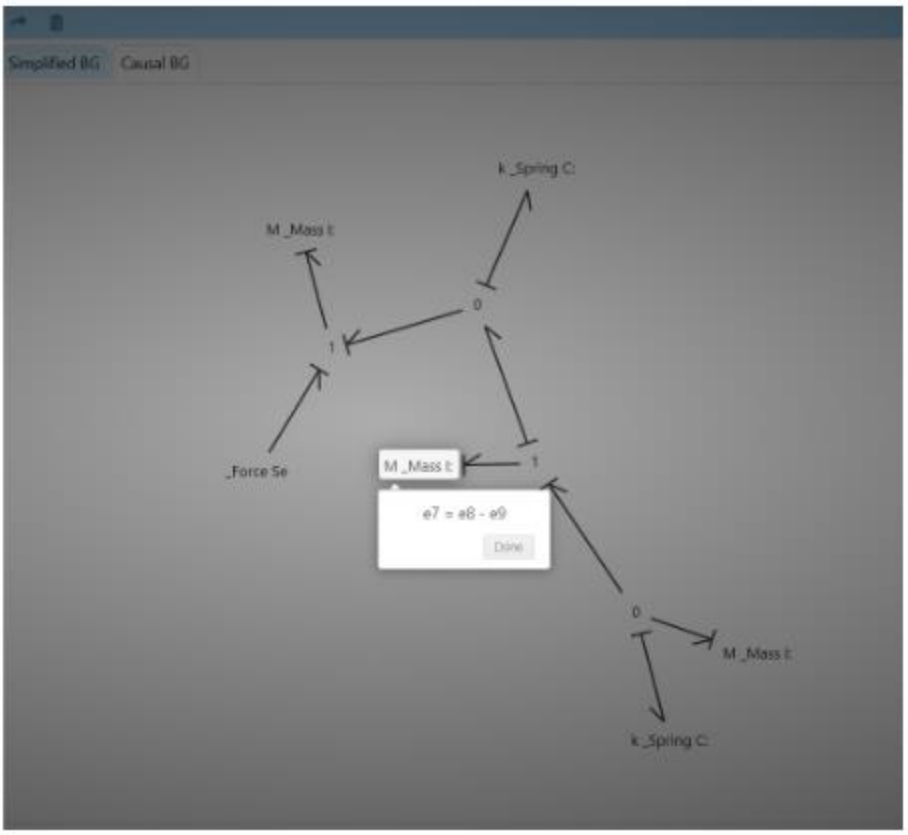


Figure 60: Mockup using IntroJS to show state equations for a given node.

The next mockup we present simply puts the differential equations and state equation on the canvas (see Figure 61). Since the user will be looking at the canvas while analyzing the bond graph anyway, they will be able to easily see and interact with the differential equations. This mockup does have several drawbacks, however. The first is that, for a larger system, many steps would be printed on the canvas alongside the final equation. This could lead to an issue where the state equations collide with parts of the bond graph, reducing the readability of both UI elements.

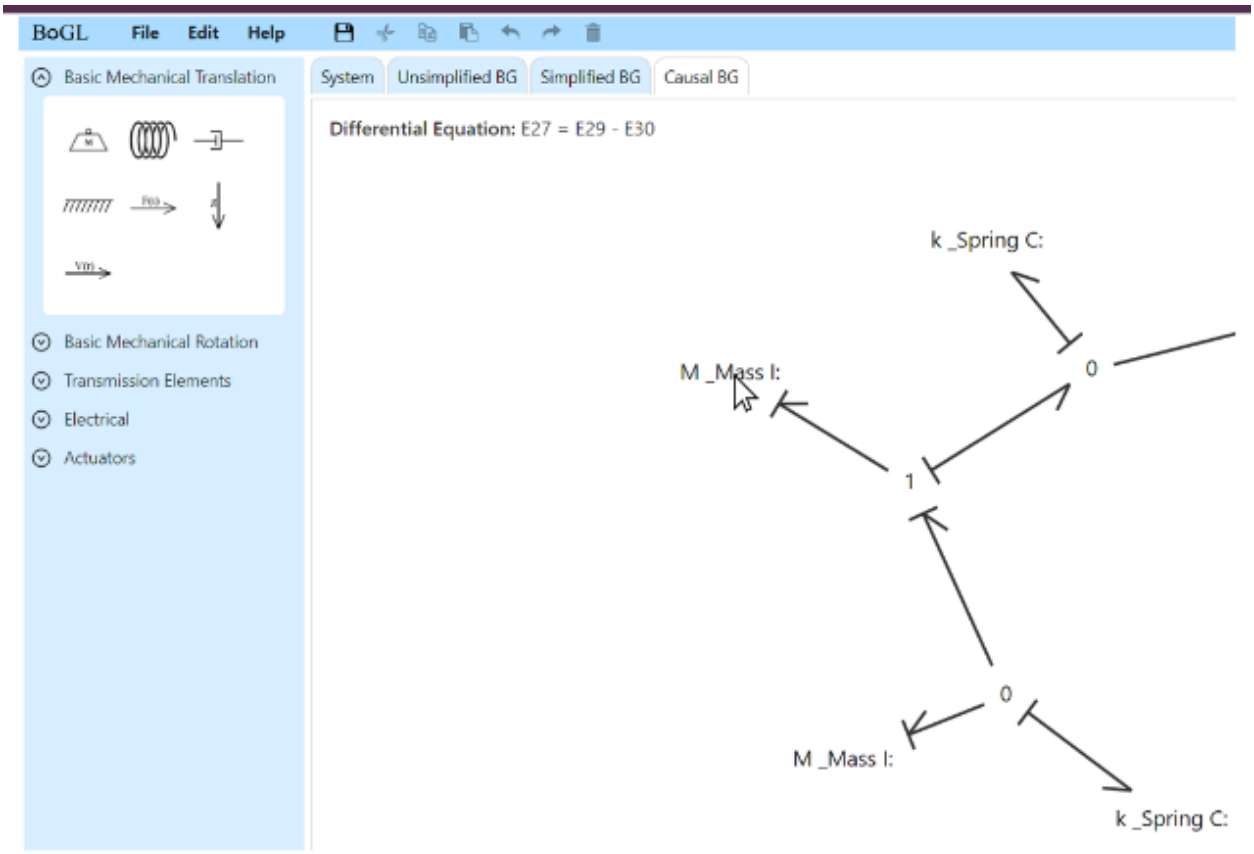


Figure 61: Mockup that prints the state equation on the canvas.

The next mockup we present, shown in Figure 62, is like the mockup which used purely IntroJS (Figure 60). For this mockup, there are two main differences, the first being that the canvas is not darkened. This improves the readability of the entire bond graph, but no longer highlights the element associated with the state equation. The second difference is the addition of colors. These colors are used to highlight elements of the final state equation to make it clear to the user which elements of the bond graph are important. This mockup has many of the same positive aspects as the previous IntroJS mockup, and with the addition of colors, makes up for the lack of a darkened background. However, many of the concerns with the previous mockups also are important to consider here. The popup box is likely to cover elements in the bond graph,

and it is not clear how to display the state equation box. It is also worth noting that this mockup could cause issue for those with colorblindness. Using the Google Chrome color blindness tool, we were able to partly analyze the effectiveness of this mockup and determined that the highlighted text would still be readable (Lyles, 2020). However, depending on the user, the colors may lose meaning and become confusing if they are no longer distinct. Further research would need to be done in this area if this mockup were to be implemented.

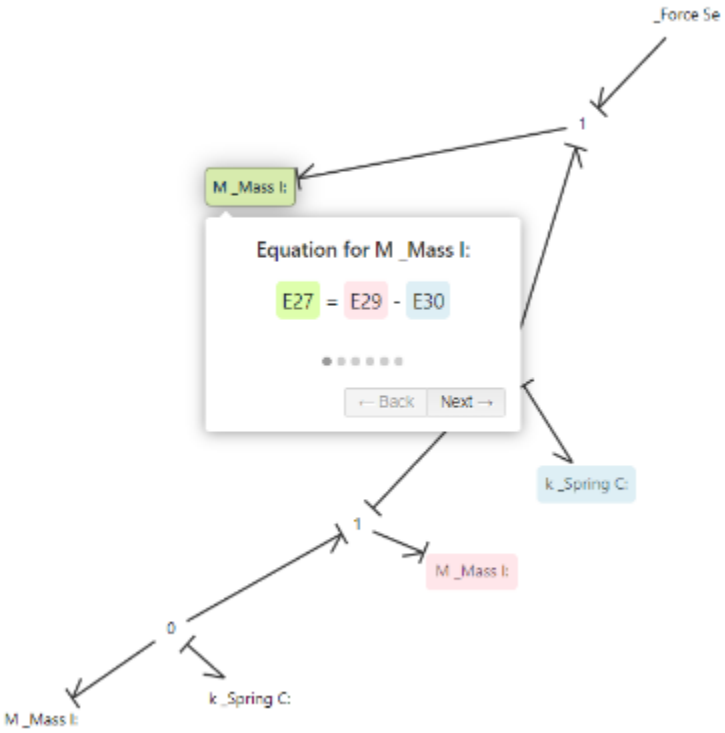


Figure 62: Mockup using IntroJS to display state equations with color highlights.

The next mockup we present is triggered through the file menu and pops up a window which shows all state equations for the system (see Figure 63). One advantage of this mockup is that, since the user would have interacted with the file menu before this point, it will be clear

how to trigger this modal. However, there are many drawbacks to this option. The first is that there is no obvious way to show steps in this mockup. This is requirement for the state equation display, so this mockup would need to be combined with something else to be usable. Another disadvantage is the amount of screen space taken up by the modal. The modal obscures a substantial portion of the bond graph, making it hard for the user to interpret the state equations.

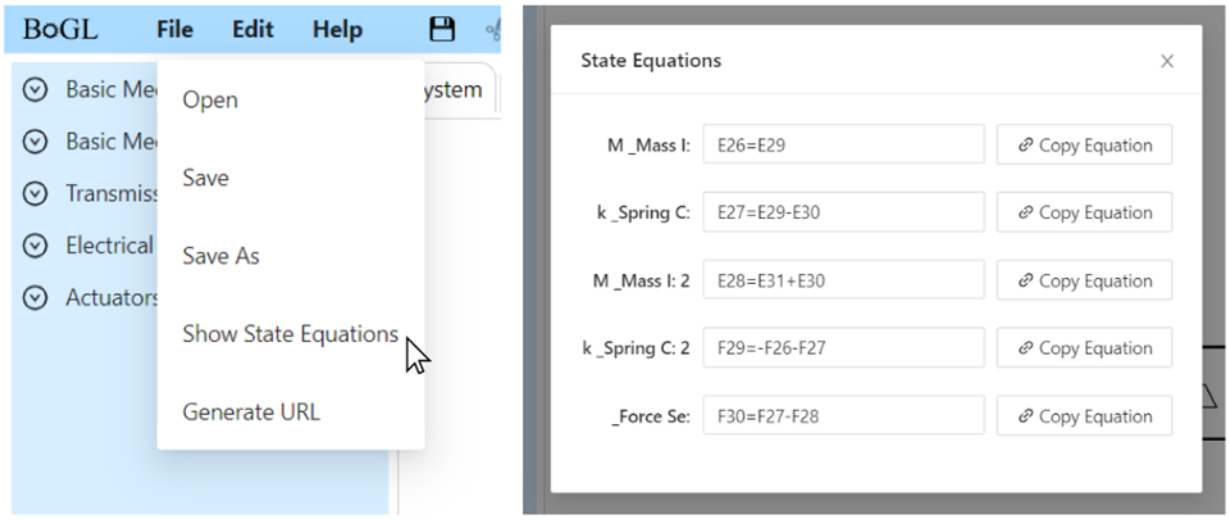


Figure 63: Mockup that displays final state equations in a modal.

The final mockup we present in this category can be seen in Figure 64. This mockup replaces the menu on the left side of the screen which would normally be used to add elements to the canvas. This menu is not usable on the bond graph screen since the user cannot add elements to bond graphs. We would replace this menu with a new menu that indicates which state equation is currently being shown, the total number of state equations for the bond graph, the set of state equation steps plus the final state equation, and buttons to move between the state equations. The final state is also circled to make it clear to the user that this is a final solution. This mockup clearly includes the steps for finding the final state equation and makes them clear

to the user. One concern with this system is that long equations may get cut off or the text may need to be small to fit them in this portion of the UI. To fix this issue, we propose that the left menu be made resizable. This will allow users to allocate more space to the state equation portion of the interface so they can more clearly view the equations. One of the main disadvantages of this system is the need for users to move back and forth between state equations using buttons. This can be slow for systems with many state equations and would be cumbersome if the user wanted to swap between several state equations quickly. One way to fix this would be to replace these buttons with a dropdown menu, like the causal options menu, which would allow the user to select the state equation that they wanted to be shown.

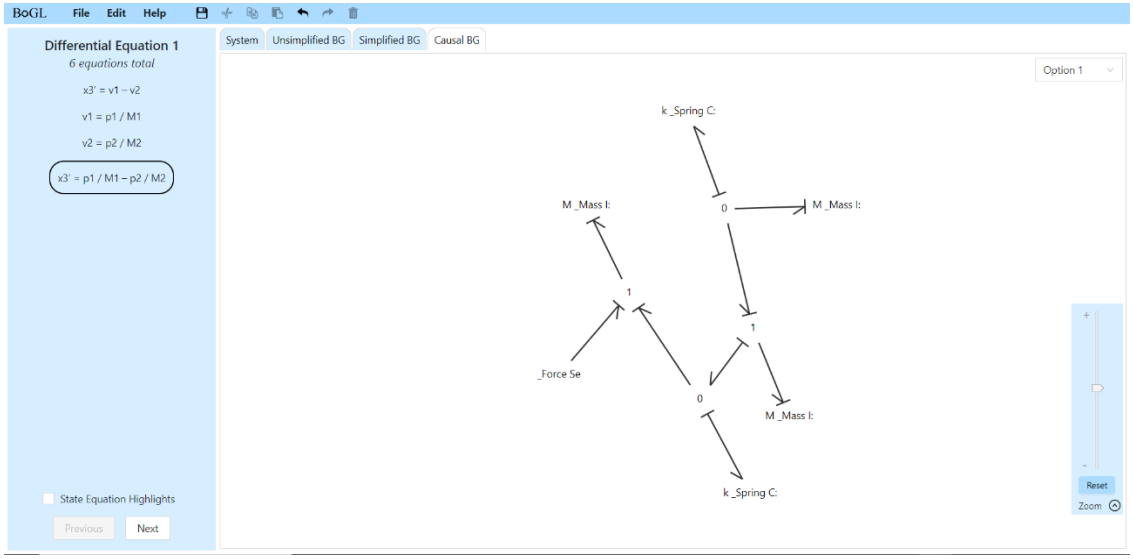


Figure 64: Mockup that displays state equations in the left side menu.

The last set of mockups we will present show labels to the user. These labels are important for understanding which elements are used in each state equation.

7.2.2.4 Mockups for Viewing Labels

For users to fully understand the state equations, they need to be able to see which elements of the bond graph each term of these equations correspond to. To do this, there must be some area of the UI which displays these labels, values to the user, and correlates them with the bond graph. These values must be easily readable, and users must be able to easily tell what part of the bond graph corresponds to each label.

The first mockup we present places the labels directly on the canvas (see Figure 65). These labels will be placed at the middle of the bond and follow the bond as it is moved around the screen. This clearly associates each label with a bond. As with element names in the bond graph, it would be possible for a user to overlap the labels with other elements of the bond graph. However, there is not a clear solution for this, and the user would have the option to re-arrange the bond graph to make these labels readable.

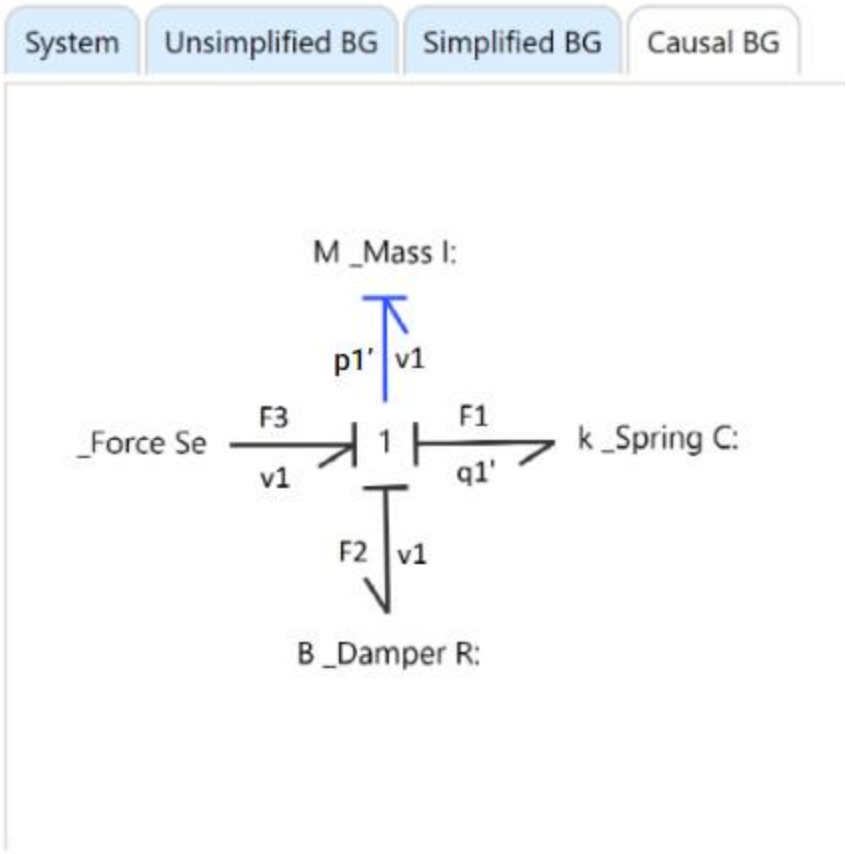


Figure 65: Mockup that attaches state equation labels to bonds.

The next mockup we present adds a menu which lists the state variables as well as other variables in the bond graph (see Figure 66). This menu will appear when the user is in the causal bond graph tab. These variables correspond to labels on the bond graph. The user can select labels in this menu and have them highlighted within the bond graph. The location of this menu is not clear as it would depend on which state equation display option was chosen. Locations could be where the modifier menu is normally, as it is not present in the bond graph tab, or as a replacement to the left menu if it were not used by another mockup. This mockup also includes gear icons, which would allow users to edit the names and values associated with these labels. One advantage of this mockup is that it makes a clear distinction between state variables and

other variables. This could be important to a student as they would be able to check if their state variables are correct. Providing a list of all labels also prevents any labels from getting lost if they are overlapping with another element of the UI. One disadvantage is that this mockup does add another menu to the interface and would increase the amount of information being presented to the user at one time. While this is not a huge issue, it is worth considering if the user would be overwhelmed by this additional menu. The menu also duplicates the labels by showing them in two locations instead of handling editing and highlighting the labels all in the same place.

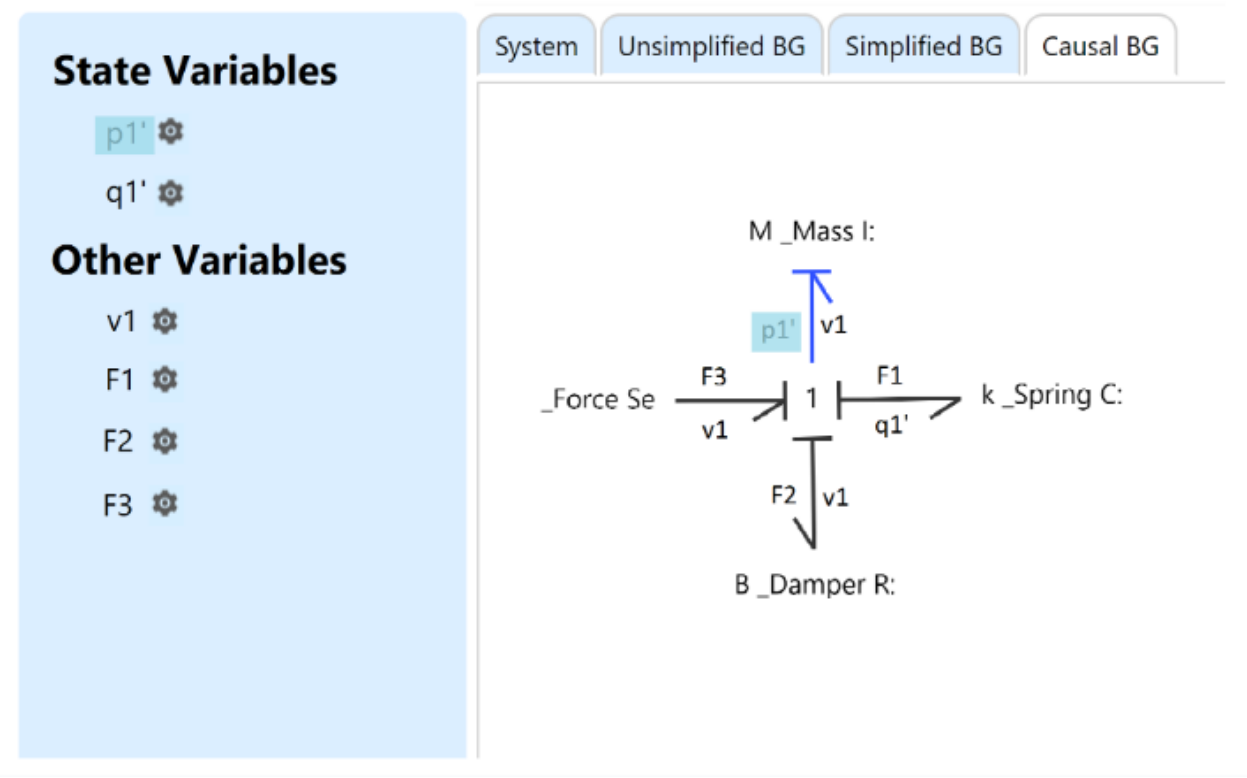


Figure 66: Mockup that highlights state equation variables in a separate menu.

The next mockup we present is dependent on which state equation mockup is selected (see Figure 67). This mockup assigns a unique color to every label. These colors are then used to

highlight the label whenever it appears. An important thing to mention here is that the same color blindness concerns need to be considered as with the mockup that used colors to highlight parts of state equations (see Figure 62). By adding colors, the user may be able to identify labels more clearly in the bond graph. However, as the number of labels in the bond graph grows, the number of distinct colors on the screen may begin to confuse the user. Additionally, from a programming standpoint, dynamically generating these colors would be a challenge. The colors need to be distinct to best help the user interpret the diagram, so the algorithm used to generate these colors would need to support that constraint. This design does require that the color scheme of parts of the website be changed to make the highlighting more visible.

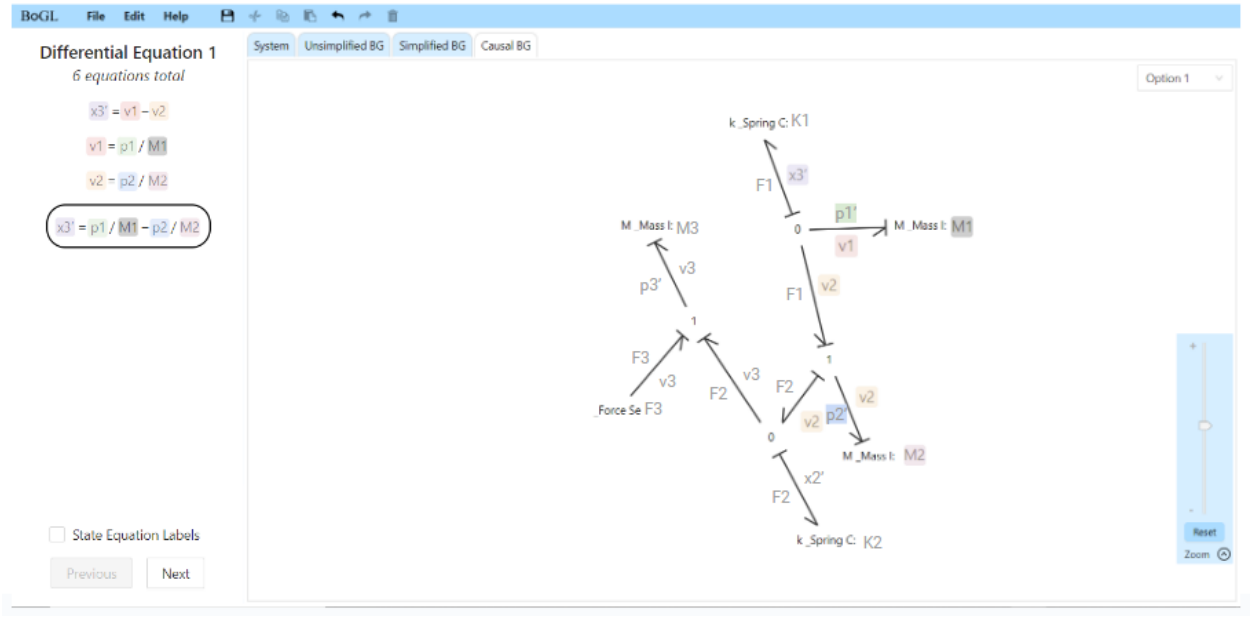


Figure 67: Mockup that highlights all state equation variables in the bond graph.

The next mockup is like the previous except that only a single equation is assigned colors at a time (see Figure 68). An equation is selected using left click. This reduces the number of

colors on the screen at once which both reduces the clutter and the programming challenge. One should note that the color blindness concerns still exist here.

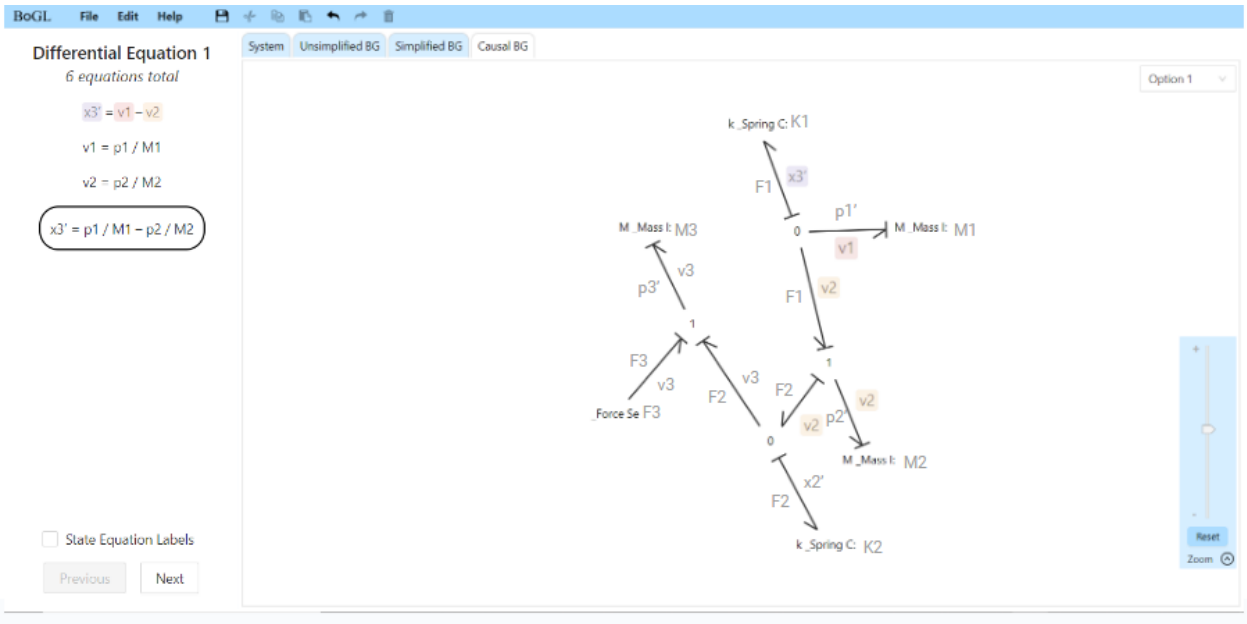


Figure 68: Mockup that highlights the state equation variables for one equation at a time.

At this point we have presented mockups for all the feature's key to implementing a state equation system. We will now move on to a discussion of which of these mockups we would recommend for implementation.

7.2.2.5 Decision Matrices

To evaluate the various mockups, we created several decision matrices to guide our recommendation. We used these matrices to compare individual elements of each UI mockup with each other. Since we plan to combine certain mockups to create a more complete system, the score here may not accurately reflect the entire system, and only compares one mockup with the others in that category. Evaluating all combinations of the mockups would have been a

complex task, so we present this information for simplicity, and will provide rational for a final decision in Section 7.2.2.6.

The decision matrices are all scored on a scale of zero to four. A score of a zero means that the mockup does not meet the criteria in the matrix at all, and a four means that the mockup fully meets the criteria. We chose to weigh each category equally. This was because we felt that each section of the matrix had equal importance.

Table 10: Table comparing methods of changing labels/values.

	Ease of Adding Values/Labels	Ease of Modifying Values/Labels	Ease of Seeing What Elements Have Values/Labels	Ease of Implementation	Effect on User Interaction with other UI Elements	Score
Auto Pop Up	3	0	0	3	1	7
Right Click	4	4	1	1	4	14
Button On Modifier Menu	3	3	1	3	3	13
Button On Element	4	4	2	1	2	13
Menu Near Modifier Menu	4	4	4	2	2	16

The first decision matrix we will discuss is for adding labels and values in the system diagram tab. The first category we have for this matrix is “ease of adding values and labels.” This category refers to how easily a user can add a label to the element when the element is first added to the screen. In other words, this refers to the first time the user would want to edit these values and not any subsequent times. All the mockups were able to meet this criterion, so they all scored a three or four. The difference here was with how easily the user would be able to find the

button to trigger the input fields. For the mockups which received a three, the location of the button was slightly more difficult to find.

The next section is “ease of modifying values and labels.” This refers to any time the user would want to edit the values and labels after they have first assigned them. For most mockups, this value will be the same as in the previous category, however, we make this distinction to account for the mockup which opens automatically when an element is added to the canvas. In this section, all mockups received a four except for button on modifier menu and auto pop-up menu. For the button on modifier menu, the score was lower due to the difficulty to find the button again. For the auto pop up menu, this mockup receives a zero since it can only be triggered when the user first adds the element to the screen. This means that it cannot be modified.

The next category is “ease of seeing what elements have labels and values.” This category refers to how easily a user can tell if an element has a value and label assigned. All mockups scored low here except for the menu near the modifier menu. This is because none of the low-scoring mockups can show if an element has a label or value, while the menu near the modifier menu shows the element’s label whenever the element is selected.

The next category is “ease of implementation.” This category refers to the estimated difficulty of implementing the feature. To determine scores for this category, we analyzed how the feature would change the current structure of BoGL Web, what UI elements would need to be added, and what would need to be done on the backend. We also looked at what UI elements would come from a UI Component Library and what elements we would need to construct from scratch. We felt that the auto pop-up and button on modifier would be the easiest to implement. This is because these ideas do not require modifications to click actions and we are able to use

the built in modal from Ant Design. The other mockups would be more difficult to implement due to needing more complex click actions. Right click and button on element are scored the worst because they both require editing click actions. This would require more development time and introduce more effort to avoid conflicts with existing click actions.

The last category is “effect on user interaction with other UI elements.” This category was used to track how the new feature would change the current user experience in BoGL Web. In this category, a score of a four means that the mockup does not have much effect on existing interactions with the UI, and a score of zero would mean that the mockup completely changes interaction with the UI. For example, if the new feature would entirely change how click actions worked in BoGL Web, requiring experienced users to relearn how to interact with the system, it would score poorly in this category. The right click action scored best here. This is because it does not conflict with any existing actions that the user would do. The button on the modifier menu also falls into the category, however, it scored slightly lower due to adding icons to an existing menu. The other mockups will interrupt the user experience more by covering screen space, creating automatic popups, and by disrupting pre-existing click actions, so they scored lower.

Table 11: Table comparing methods of displaying state equations.

	Ease of Seeing Final State Equation	Ease of Seeing Steps to get to State Equations	Ease of Seeing Elements Involved in State Equations	Ease of Switching Between State Equations	Ease of Implementation	Effect on User Interactions with other UI Elements	Score
IntroJS	2	1	2	3	1	1	10
On Canvas	2	1	1	3	3	0	10
Colors	2	1	4	3	1	1	12
Modal Popup	3	0	1	4	4	1	13
Left Menu	4	4	4	4	3	4	23

The decision matrix in Table 11 was used to compare methods of displaying state equations. The first column of this decision matrix relates to the ease of seeing the final state equation. To score high in this category, the final equation in the mockup would need to be easily readable, and the user would need to be able to easily locate this equation. We ranked the mockups with IntroJS, the equations on the canvas, and the colors, a two since, while the equations were easily visible, there was nothing which highlighted that this was the final equation. The modal pop up did better in this category since it only shows final equations. However, there is still nothing which highlights that these are the final equations. The left menu mockup scored a four because the equations can all easily be ready, and the final equation is highlighted with a circle.

The next category ranked the ease for a user to see the steps to get to the final equation. For a mockup to score well in this category, a mockup would need to be able to clearly show all the steps needed to reach the final state equation. All the mockups except to the left menu scored poorly since they do not inherently show the steps to find the final state equation. The left menu scored a four since it clearly shows the steps to reach the final state equation.

The next category ranked the ease with which a user can find an element involved in the state equation. To rank high in this category, a mockup would need to make it clear what parts of the bond graph were present in each equation. This could be done in a variety of ways if it were clear to the user what parts of the bond graph were important. In this category, all mockups scored poorly except for the left menu and the mockup which colors the elements. The other mockups scored poorly because they covered parts of the bond graph by appearing over the display. The left menu does not cover the bond graph area and the mockup with colors helps the user easily see what elements are involved in the bond graph. Because of this, they ranked highly.

The next category is the ease of switching between state equations. To score highly in this category, a mockup would need to make it clear to the user how to switch between state equations. All mockups scored highly here. The distinction between a three and a four was how clear it would be to an inexperienced user how to switch between the state equations.

The next category is ease of implementation. As for the previous matrix, to score well in this category, a mockup would need to be easy for developers to implement, and not require a large restructuring of existing code. The mockups which used IntroJS, and colors scored poorly in this category since they would require modifying the current click actions. This adds a lot of complexity and would require changing how click actions work for other elements of the user interface. The left menu and equations on the canvas scored a three since they are simple to implement. The modal pop up scored the highest because it would be the easiest to implement, as it just re-uses the Ant Design modal component. Since the modal would be triggered by a button, it would not require any modifications to click actions either.

The last category is effect on other UI elements. Like the previous matrix, to score highly in this category, a mockup would need to limit its impact on the existing UI and not require a user already experienced with the current system to relearn many tasks. All mockups scored poorly in this category except for the left menu. This is because these mockups would have a substantial impact on what the user would currently see when viewing state equations and potentially require modifying click actions. The left menu mockup only replaces the element palette, which is not used in the bond graph tab anyway. The mockup earns a four for having a minimal impact on the user experience.

Table 12: Table comparing methods of displaying state equation labels.

	Ease of Seeing Labels	Ease of Modifying Labels	Ease of Implementation	Effect on User Interaction with other UI Elements	Score
On Canvas	3	3	2	2	11
Menu	3	4	2	0	9
Constant Labels Highlighted	2	2	1	1	8
State Equation Select Highlighted	3	2	1	2	10

The decision matrix in Table 12 was used to evaluate mockups which would display state equation labels to the users. The first category in this matrix is the ease of seeing the labels. To score well in this category a mockup would need to make it clear to the user what labels were associated with what elements. All mockups scored well in this category except for constant label highlights. This scored worse because it added a lot of clutter to the screen. No mockups received a perfect score because it is possible to make all labels difficult to read if the bond graph is poorly laid out.

The next category is the ease of modifying labels. Ideally, the user would be able to change the labels to their liking. To score highly in this category, it needs to be clear to the user how to modify a label, and the menu to modify the label would need to be easy to use. Constant label highlighting and selected state equation highlighted scored a two since, they do not inherently provide a method of modifying labels, but this could be added easily. If it were added, it may not be immediately clear how to interact with this feature.

The next category is the ease of implementation. To score well in this category a mockup would need to be simple to implement and limit conflicts with existing elements of the system. All mockups scored poorly in this category since they would all need to interact with the canvas in some way. The two highlighting options scored lower because implementing the highlighting for elements increases difficulty.

The last category is effect on other UI elements. Like the previous matrix, to score highly in this category, a mockup would need to limit its impact on the existing UI and not require a user already experienced with the current system to relearn many tasks. Again, all mockups scored poorly in this category. This is because all mockups heavily interact with the existing user interface due to the nature of this category. Some mockups scored better because they only interacted with the canvas and did not cause a heavy impact on existing UI elements.

Now that we have analyzed all the mockups through decision matrices, we can make a recommendation about how the state equation user interface should be implemented. In the next decision we will take the information from the decision matrices and compile it into a recommended UI.

7.2.2.6 Recommendations

In this section, we will provide a recommended user interface to be implemented. These recommendations are motivated by the decision matrices from the previous sections as well as conversations with our advisors.

We will begin with a recommendation for adding labels. For this we recommend that developers implement the mockup seen in Figure 59, which is the label/value menu near the modifier menu. This mockup scored the best in the decision matrix and our team feels it would provide the best user experience. To account for modifier labels and values, we would recommend that developers add an accordion style collapse menu to this component. This would make it so the user would not have a lengthy list of modifier labels and values covering a sizable portion of their screen. This menu would only allow for one pane of the menu to be open at once, preventing the user from intentionally making the window too large. The element pane would open automatically on selecting an element and a modifier section would automatically open when a modifier is added to the element. When modifiers are removed, they would be removed from this menu.

For displaying state equations, we recommend that developers implement the left menu seen in Figure 64. This mockup scored the best on our decision matrix and was heavily preferred in our meetings with our advisors. We recommend that developers replace the buttons to switch between equations with a drop-down menu, like the dropdown which exists for switching between options of the causal bond graphs, to help users view all the state equations that they can select. Figure 69 shows a mockup of such a dropdown.

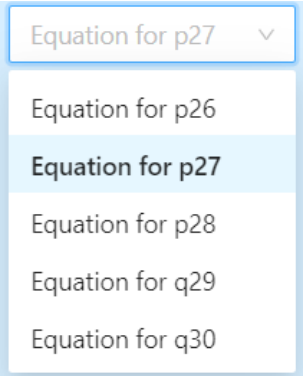


Figure 69: Dropdown menu mockup for selecting state equation.

The last section of the UI that needs to be implemented is a way to display labels. For this we recommend that labels be displayed on the canvas. For now, we do not recommend any additional features as these appear to be unnecessary. If highlighting variables is found to be an important feature, the mockup that highlights the variables from one equation is preferable to the one that highlights all variables simultaneously.

While we originally expected to simply recommend UI options for state equations, we ended up implementing some aspects of these recommendations. These implementation details can be found in Section 8.3.2.

7.3 Tutorial Design

We chose to use IntroJS to run our tutorial. For the rationale behind this decision, see Section 8.3.4. IntroJS works by highlighting a UI element, then displaying text and images or gifs beside the highlight (see Figure 70). We created this tutorial to walk the users through the website so they would not have to use their intuition to learn how to construct a system diagram and generate a bond graph. This tutorial will teach the user how to add elements to the canvas,

create edges, and generate bond graphs. We also teach the user how to add modifiers, use the zoom menu, and introduce them to several other features in the UI.

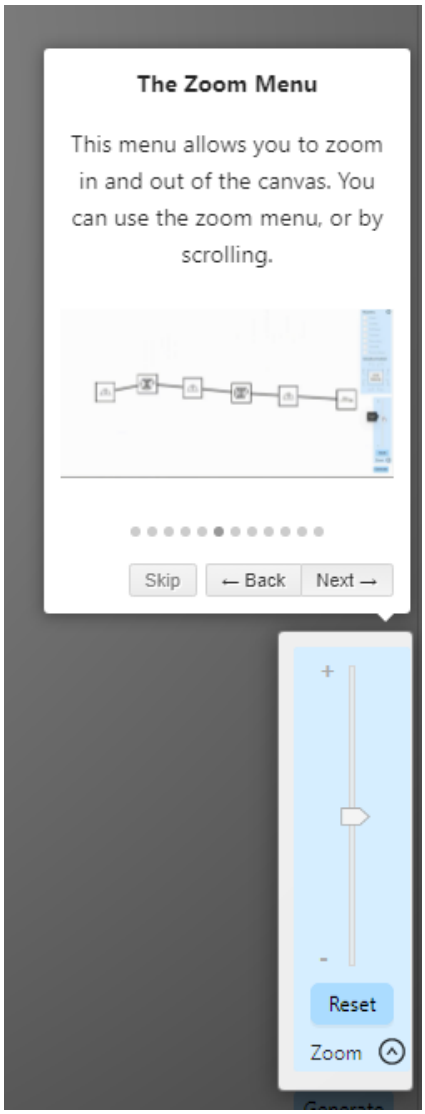


Figure 70*: Zoom menu being explained in the tutorial.*

Once we had chosen IntroJS, we needed to plan what UI elements the tutorial would highlight for demonstration, their order, and what text/images would be displayed next to the highlighted UI element. Users can navigate the tutorial using the next button to move to the next

set, back button to move to the previous step, and the skip button to skip a step. We needed to have each of these steps in our plan. This plan can be seen below. Next to each number in the list is the element that will be selected, and the sub bullets will be the message which is displayed.

Tutorial Steps:

1. No element selected/Whole Screen (Intro Screen)
    a. Welcome to BoGL Web
    b. This application is used to construct system diagrams and generate bond graphs from those diagrams.
2. Canvas
    a. The highlighted space is the Canvas where you can construct, move, and rearrange your system diagrams.
3. Element Palette
    a. This is the element palette. After expanding the menus, you can select and drag elements onto the canvas to construct system diagrams.
4. Constructing a System Diagram (Figure 71 is included as a GIF in this step of the tutorial)
    a. Drag an element from the element menu into the Canvas to add it to the system diagram, and then select near its black border to start creating an edge. You can then select near a second element to finish making the edge. If you see a green circle, your edge is valid, if you see a red X when you try to make an edge, it means the edge you are trying to make is invalid (the two elements do not make sense to be connected).

b. Gif of adding elements to the system diagram and the creating a valid edge and an invalid edge

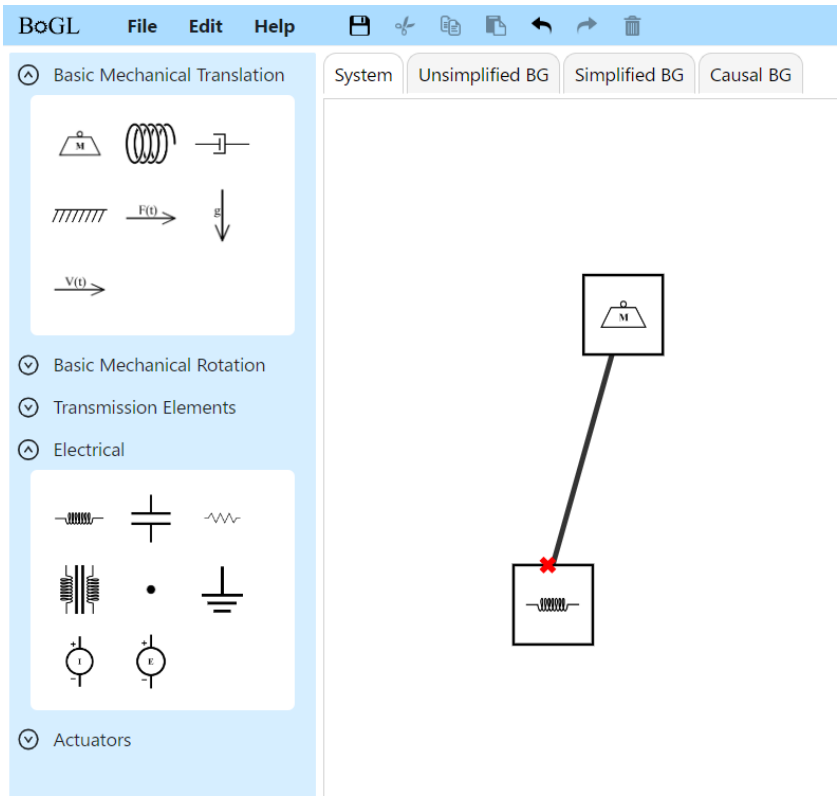


Figure 71: Still frame from the gif of adding an invalid edge.

5. Modifiers

   a. Use this menu to add modifiers to the selected element. Some modifiers require multiple elements to be selected. You can do this by holding down the control key and clicking elements you want to select or dragging the cursor across the canvas with the left mouse button to create a selection region. All elements that are completely or partially inside the region will be selected.

6. Zoom

a. This menu allows you to zoom in and out of the canvas. You can use the zoom slider, or your scroll wheel.
b. Gif that shows zooming

7. Generate Button
    a. The generate button allows you to turn your system diagram into a bond graph. While the bond graph is being generated you will see a loading bar which signifies that BoGL Web is processing your System Diagram. This can take a few seconds.
8. Tabs
    a. These tabs change what stage of the bond graph generation is being displayed. You can look at the unsimplified bond graph, the simplified bond graph, or the causal bond graph.
9. File
    a. This is the file menu button. Selecting it opens a menu which allows you to:
        i. Create a new system diagram.
        ii. Open a previously saved .bogl file from your computer.
        iii. Save a .bogl file representing the system diagram to your computer.
        iv. Generate a URL that can be used to share your system diagram.
10. Edit
    a. This is the edit menu button. Selecting it opens a menu which allows you to:
        i. Copy, cut, and paste elements of the system diagram.
        ii. Undo and redo changes
        iii. Delete elements from the System Diagram

11. Toolbar
    a. You can perform similar features to the edit menu here. By selecting the icons, you can save a system diagram, cut, copy, paste, undo, redo, and delete an element or edge from the system diagram.
12. Help
    a. This is the help menu button. Selecting it opens a menu which allows you to:
        i. Confirm deleting many items at once. Selecting this option will allow you to select multiple items and then delete them all at once.
        ii. Start the tutorial.
        iii. Load example system diagrams.
        iv. Report bugs that you find.
        v. Learn about who created the BoGL Web system.

The goal of the tutorial was to provide a quick onboarding experience for users so they would have an idea of how to use the website without someone needing to walk them through it in person. The IntroJS tutorial met this goal, as shown in Section 10.2.

8 Implementation

In this section, we will discuss the implementation details of all components of BoGL Web. This includes both front end and back-end components. We will also discuss new features which our team has added in addition to.

8.1 Backend

This section discusses the backend of the application which was written in C# using the Blazor WebAssembly framework. We will first discuss the data flow diagram that describes the structure of the backend. We next discuss several key features of the backend. These include loading rules of ruleset, graph processing, loading system diagrams from files, image conversion. undo/redo, and state equations.

8.1.1 BoGL Web Backend Data Flow Diagram

When planning out the backend, we created a data flow diagram to compartmentalize various aspects of the system (see Figure 72). There are two colors of boxes within the diagram. The blue boxes are components of the backend. The gray boxes are places where data can be stored/seen. The arrows show the data flow between components in the backend.

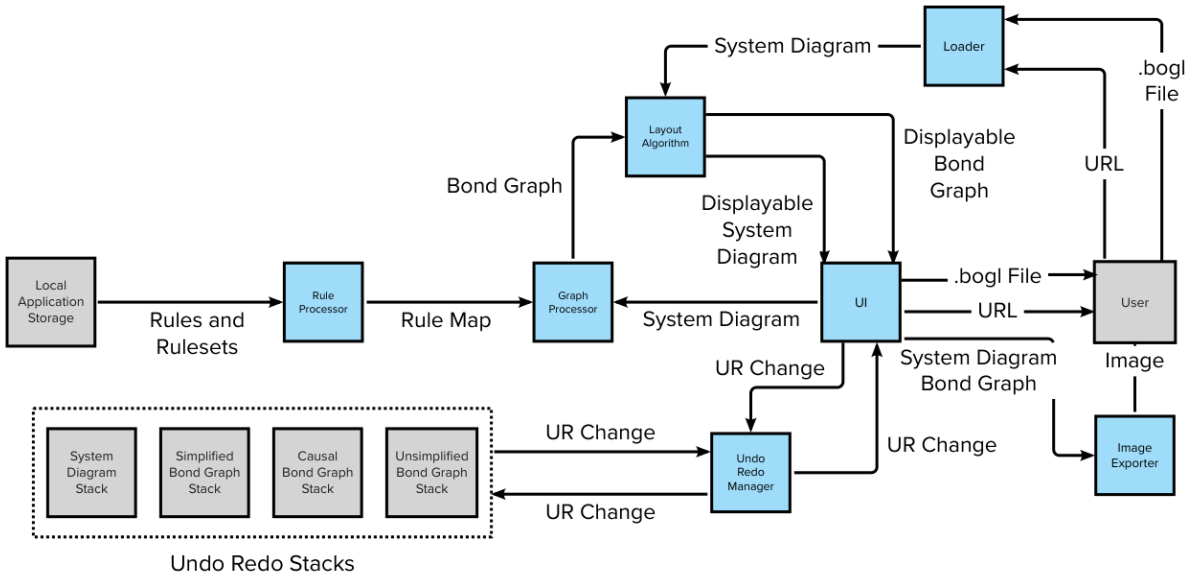


Figure 72: BoGL Web Backend Data Flow Diagram.

We will now explain the different components of the backend. These are the blue boxes seen in Figure 72. We will move through the diagram from left to right to describe each component of the backend.

The first component we see is the rule processor. This component is responsible for loading and storing the rules and rulesets used to process system diagrams and generate bond graphs. More details on this component can be found in Section 8.1.2.

The next component we see is the graph processor. This component is responsible for taking a system diagram and the rules and rulesets and generating all the bond graphs associated with the system diagram. More details on this component can be found in Section 8.1.3.

The next component is the layout algorithm, otherwise known as the graph embedder. This component takes in a bond graph and generates a visually appealing layout that can be displayed to the user. More details on this component can be found in Section 8.3.4.

The next component is the Undo/Redo manager. This component is responsible for tracking changes made by the user so they can be undone or redone. This requires managing all the undo redo stacks which can be seen in the dotted box. More details on this component can be found in Section 8.1.7.

The next component is the UI. This component is responsible for managing all the actions done by the user. It is the intermediate between the frontend and the backend. More details on this component can be found in Section 8.2.

The next component is the loader. This component is responsible for loading system diagrams from files, or URLs. More details on this component can be found in Sections 8.1.4 and 8.1.5.

The final component is the image exporter. This component takes the graph area from the current tab, whether it be the system diagram tab or one of the bond graphs tabs and converts it to an image. The user is prompted for a location in their local file system where the image should be downloaded.

By creating this separation, we were able to create code that was easy to maintain and scale because a developer would be able to quickly find where specific computations were happening.

8.1.2 Rules and Rulesets Implementation

In this section, we discuss our implementation of loading and storing rules and rulesets.

8.1.2.1 Design Patterns

Design patterns are created to solve recurring problems in the design of software with a standard, reusable, solution (Gamma, 1995). There are four elements stored in each design

pattern. The first is the name of the pattern, which allows a programmer to quickly understand the problem and solution which the pattern is solving. The next element is an encoding of the problem that the design pattern will solve. This can include problems relating to algorithm representation or class structure. The third element is the solution. This is the way in which the pattern will rectify the problem which was presented. This is providing a standard for how to layout a class or algorithm. The final element is the consequences or tradeoffs that are made when choosing to implement the design pattern. Common examples of these tradeoffs are the time and space required to run and store the program.

8.1.2.2 Singleton

The Singleton design pattern was created with the intent of ensuring that there is only ever once instance of a class (Meersman et al., 2008) This is implemented using a static or global variable to store a single instance of a class. There are several common ways in which this is implemented. The naive way to create a singleton is eager instantiation, which creates an instance of the object when the program starts or when the class is loaded (Meersman et al., 2008). This means that the instance will be created even if there is never an attempt to access the class. This comes into play when there is a concern about the allocation of limited resources. The most popular is lazy instantiation. This is when a method is created which allows for access to the instance. This method checks whether the instance has been created, and if it has not, it creates one meaning that the object is only created once it is needed, preventing resources from being allocated unnecessarily. This will avoid the issues of eager instantiation but can have issues in multi-threaded environments. There are several other methods in which singletons can be implemented which can be found in (Meersman et al., 2008) however the specifics will not be necessary for our use case.

8.1.2.3 Our Implementation

For our implementation we created a singleton using lazy instantiation. We chose this option because our application runs on the web, which limits the amount of space we can use. Since Blazor only allows for single-threaded execution, we can disregard any potential problems caused by multithreading. Lazy instantiation is also the most used form for a singleton, which provides our team and application with ample support. Additionally, someone reading our code would be able to recognize that we have created a singleton, assuming they have a reasonable understanding of design patterns. This means that our code will be easier to understand for programmers working on the project in the future.

8.1.2.4 Loading Rules

To load rules and rulesets, we first create a list of strings with the names of all the rulesets. We already know the source and destination directories for the rulesets, so we have all the information we need. Additionally, the names of the rules are stored within the ruleset files so this information does not need to be stored in our code. Our next step is to load the rules and rulesets.

In Blazor WebAssembly, loading files is done asynchronously. This is mandated by the language and ensures that the UI will not freeze when files are being loaded. Additionally, files are loaded by making an HTTP connection to a URL which represents the location where the file is stored.

Once the data is loaded, we can pass it to GraphSynth which performs the parsing needed to interpret the loaded text.

8.1.2.5 An Attempt to Speed Up Loading Rules and Rulesets

The implementation of loading rules and rulesets described above takes 14-20 seconds to finish loading depending on internet speed. One idea to speed up this loading time was to reduce the number of HTTP requests we needed to make to retrieve all the rules. We tested this by storing a compressed zip file instead of individual files. This made it so only one HTTP request needed to be made instead of one for each file.

Unfortunately, this did not speed up the loading times for the website. This leads us to the conclusion that the load time is bounded by how fast the system can read the text in the files rather than the response time of the HTTP requests.

The conclusion that we came to is that, to improve the speed for loading the rules and rulesets, we would need to move this computation to compile time. Testing this idea was outside of the scope of this project, but it is something to experiment with in the future.

8.1.3 Graph Processing Implementation

The structure of the graph processing code does not necessarily follow a standard design pattern but is close to a factory (Gamma, 1995). We use a static method to create an instance of the graph processor. The static method takes in a System Diagram, and returns a tuple with the unsimplified bond graph, simplified bond graph, and a list of causal bond graphs. We found that this structure made the most sense for how the graph processing code was called since it did not need to maintain any state after the bond graphs had been generated.

Graph processing relies heavily on a library which was also used in the 2020 MQP called GraphSynth. Since we chose to use Blazor WebAssembly which used C#, and GraphSynth uses C# importing this library was as simple as copying the files from the 2020 MQP's code base and

adding it to ours. Unfortunately, this method does not allow for the library to easily be updated. A more robust solution would be to use Dependency Injection to add the library to the project. However, this did not appear to be an option with GraphSynth since it was not hosted on Dependency Injection platforms. Luckily, the current version of GraphSynth worked well for the previous MQP, so updating should not be too much of a concern.

Now that we had access to GraphSynth, the next step was to take the existing code from the previous MQP for converting System Diagrams into bond graphs and manipulate it to a state which allows our program to use it.

When the existing code was first copied into the BoGL Web codebase, there were many errors. Many of these errors were caused by not having the system diagram and bond graph representations which were used in BoGL Desktop. To remedy this issue, we created a function which was able to take the output from GraphSynth and convert it into our System Diagram and bond graph representations. Due to the lack of documentation around GraphSynth, this was a process of experimentation, where we used the Visual Studio debugger to evaluate what was being generated by GraphSynth. We were then able to correlate the data coming from GraphSynth and map it to our bond graph class.

Once we determined how to extract data from GraphSynth, it was easy to wrap the generate function in a factory to make it easy for our code which handles the generate button click to integrate with the bond graph generation code.

8.1.4 Load from File Implementation

To implement the ability for users to load .bogl files we built a custom parser. We chose this method over reusing the code from BoGL Desktop since BoGL Desktop looked for symbols

in static positions instead of looking for keywords. This makes it more difficult to detect the cause of error, and more likely for the parser to throw errors.

To construct our parser, we first created a context free grammar.

$$P \rightarrow \text{`[Header]'}Q\text{`[Elements]'}R\text{`[Arcs]'}S$$

$$Q \rightarrow \text{``|`name' DOUBLE Q}$$

$$R \rightarrow \text{``|\{`name' STRING `x' DOUBLE `y' DOUBLE `modifiers'\{T\}\}}R$$

$$T \rightarrow \text{``|X T|`velocity' INTEGER T}$$

$$X \rightarrow \text{`MASS'|`INERTIA'|`STIFFNESS'|}$$

$$\text{`FRICTION'|`DAMPING'|`PARALLEL'|`TOOTH WEAR'}$$

$$S \rightarrow \text{``|\{`element1' INTEGER `element2' INTEGER W\}}S$$

$$W \rightarrow \text{``|`velocity' INTEGER}$$

This grammar was constructed by examining .bogl files generated from BoGL Desktop. In This grammar, strings wrapped in ‘’ are literals, DOUBLE is a floating-point number, INTEGER is an integer, and STRING is a string.

Now that we had a grammar which defined the structure for a .bogl file we were able to start creating the parser. Due to the simplicity of the file structure, we did not use an established scheme for a parser and instead just parsed the file piece by piece. We first take the entire string representing the .bogl file and split it into tokens. Each token is an individual word in the .bogl file.

Once the file was tokenized, we started by parsing the header. Since we know that a header starts with the string “[Header]” we were able to use this to find where the header started. We then looked for a name of a modifier and then a double which would be the value. We then

put these headers in a dictionary where the keys were the names and values were the doubles. This dictionary aligned with the header dictionary in the System Diagram object.

We then parsed the elements. We were able to tell when the list of elements started by the presence of the “[Elements]” token. We started by creating a queue with all the tokens which would need to be processed for all elements in the file. We then dequeued each token one by one and were able to process it based on assumptions about prior tokens. If we found a token which we did not expect we were able to throw a detailed error message about what the token was that caused the error, and what token was expected. These error messages did not end up being used in the final system due to their technical nature, however, they are there for developers when debugging issues.

A similar process was completed for arcs. Like headers and elements, we have the token “[Arcs]” to signify the start of the list of arcs. Arcs are processed using a similar queue structure to elements. Each arc contains two element numbers and potentially some modifiers. Since we parse the elements in order, we can use the indexes in the list of elements to get elements corresponding to these numbers.

Our parser also looks for matching opening and closing braces. This further improves the robustness and ensures that we only load valid .bogl files.

8.1.5 Image Conversion and Export to Image Implementation

In BoGL Desktop, the File menu includes an option labeled “Export as Image.” When selected, the application downloads a PNG image to the user’s computer that shows the current tab as the user sees it, with its current zoom and pan. In BoGL Web, we decided to approach this feature a little differently. When loading system diagrams, we ignore the saved file’s zoom and

pan information and instead always center the system in the user's view and zoom in such that the system diagram is as large as possible with a percentage of the view reserved for a margin. On saving BoGL files, we are also no longer saving the user's current zoom and pan. In the same way, images exported by BoGL Web always show the entire system diagram or bond graph with a small margin around it at 100% zoom. This allows users to export images that capture the whole graph without having to align their view and zoom level with the whole graph. We consider this a useful modification because users usually want to capture the entire system diagram or bond graph on export. In the unlikely case where they do want to highlight a particular portion of it, it makes more sense to export the whole thing and let them crop the image themselves.

Another change from BoGL Desktop is the options for image export type. BoGL Desktop always exports images as PNG. For BoGL Web, we also offer JPG and SVG as export type options, although PNG is still the default. We mostly offered these two new file types because it required little extra effort on our part, since JPGs are created the same way PNGs are and our images start as an SVG. Students might want to use an SVG image with large system diagrams or bond graphs so that they can zoom in on portions of the graph without the image being pixelated. SVGs may also be useful when the image will be shown on large displays, such as a projection, or for more professional final formats like a paper.

System diagrams contain references to SVG files in each element to display the element's logo, such as a picture of a mass or spring. This requires them to have a step of preprocessing before being converted to an image. The SVG image generated from the system diagram, whether it is output directly to the user or used as an intermediate step in generating a PNG or JPG, loses access to the SVG images that the system diagram SVG references, resulting in an

exported image that does not display these images. To avoid this, the image export code briefly replaces all these external SVG references with an inline version of the SVG so that all the SVG code is contained within one SVG. The inline SVG conversion is applied to system diagrams directly instead of making a copy because it does not change the graph's appearance to the user and because updating the graph after making these changes reverts them, so it makes no permanent change to the SVGs. The next preprocessing step for converting system diagrams or bond graphs to an image is adding CSS styling to the image. Like the external SVGs, CSS classes applied to SVG elements cannot deliver their normal functionality when they are separated from BoGL Web's CSS, so a few CSS properties are added to elements as inline styles.

Once the system diagram or bond graph has undergone preprocessing, its SVG is saved as a data string. This string can be converted to a Blob to export the file directly as an SVG or can be rendered on an HTML Canvas to export the file as a PNG or JPG. Since an HTML Canvas can render SVG images and export as PNGs and JPGs, it handles the conversion from SVG to PNG and JPG without using any external libraries. When the image file is ready to be downloaded, the user is shown a standard "Save As" file explorer, shown in Figure 73, that lets them choose a file name, image type, and location on their machine before saving the file there.

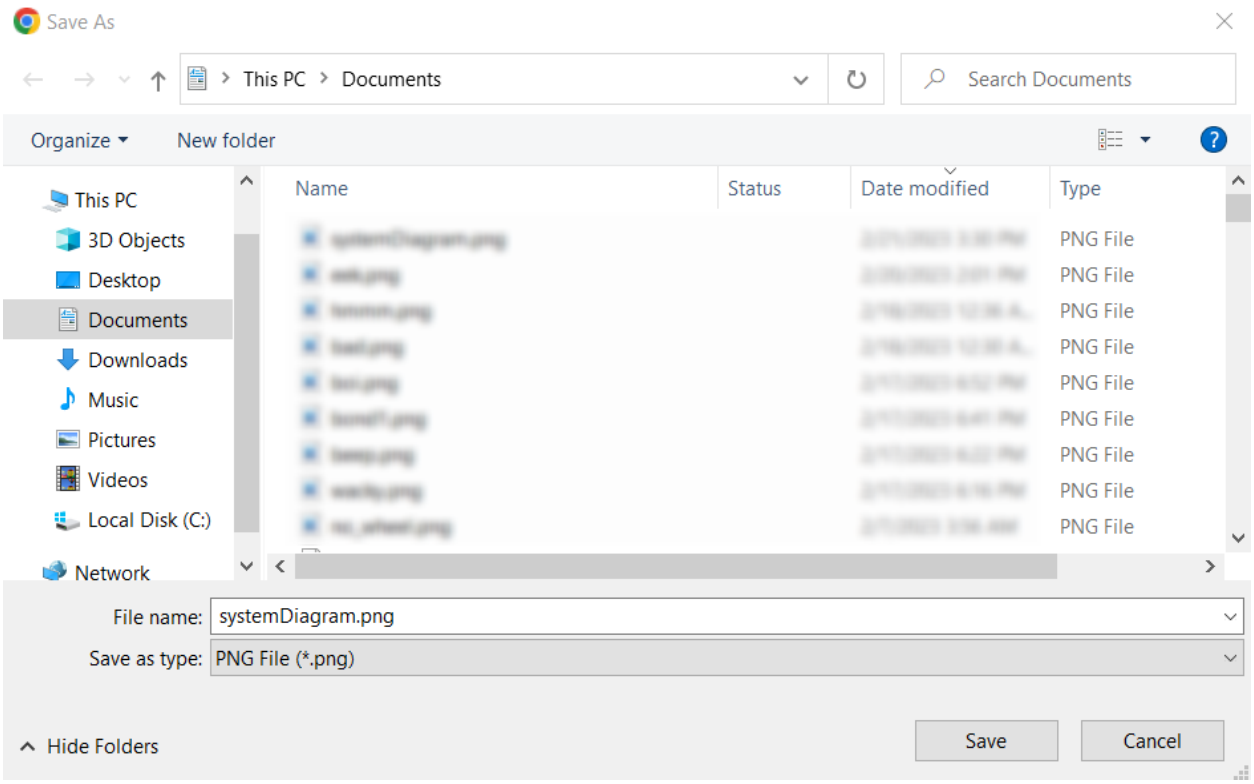


Figure 73: A Windows Save As file explorer saving a BoGL Web system diagram as an SVG.

8.1.6 Undo/Redo Implementation

In the next section, we describe the process of undoing or redoing a change in the canvas. We describe the singleton used to store canvas changes and other information in the backend. We provide all methods of invoking a call to that singleton in the BoGL Web interface. Finally, we describe what happens in the backend when the user makes a change to the undo/redo stack.

8.1.6.1 Edit Stack Handler

We store changes made to the canvas in a single *EditStackHandler* object. The application only ever needs to store one system diagram and a specific set of bond graphs that the user is working on. These are all kept as fields in the *EditStackHandler*. There is an

EditionList field associated with every system diagram and bond graph in the *EditStackHandler*. Any change made by the user in the canvas is represented as one of the child objects of *CanvasChange* and stored in the *EditionList* corresponding to the system diagram or bond graph to which the change was made.

8.1.6.2 Undo/Redo Interface Elements

The BoGL Web interface accesses the *EditStackHandler* in a few separate ways. When the user makes a change in the canvas, the program automatically adds information about that change to *EditStackHandler*. There are also keyboard shortcuts that add or remove elements or change the set of selected elements in the interface; these shortcuts are provided in Section 8.2.1. Each shortcut has a corresponding clickable option in the Edit menu and in the icon menu above the canvas.

8.1.6.3 Undo/Redo Process

When the user makes a change in the canvas, the application determines what action the user took to modify the system diagram or bond graph. This action is stored as a child object of *CanvasChange* and passed to the *EditStackHandler*. With it, the algorithm also passes an int designating the current user tab – ‘0’ for system diagram, ‘1’ for unsimplified bond graph, ‘2’ for simplified bond graph, and ‘3’ for causal bond graph. The *EditStackHandler* locates the respective stack according to the passed int and adds the *CanvasChange* to it. The program then calls *ExecuteUpdate* with the respective bond graph or system diagram and a boolean flag set to *false*.

The *ExecuteUpdate* function edits a bond graph or system diagram. This method is specific to each *CanvasChange* object and can access the fields of any *CanvasChange* child

objects. These fields are described in Section 7.1.1. The function receives the system diagram or bond graph to which changes should be applied. The other parameter received is a boolean flag indicating *true* if *undo* was called and *false* if *redo* was called. If the flag is *true*, *ExecuteUpdate* will undo the changes specified in *CanvasChange* and move the pointer on the respective *EditionList* to the previous edit. Otherwise, the flag is *false*, and *ExecuteUpdate* will apply those changes and move to the next edit in *EditionList*, clearing all next elements if this action adds a new edit to the stack.

8.1.7 State Equation Implementation

After the "Generate" button is clicked in the system diagram tab, the bond graphs for the system shown are generated, and then the state equations are generated. In this section, we describe the generation algorithm for state equations in BoGL Web. We first provide the format for equation storage in the backend. We then present the process by which the program forms partial equations from leaf nodes in the graph. We then describe how those equations are combined to form the final differential form of the state equations.

8.1.7.1 Expression

The backend generates state equations by manipulating *Expression* objects. Each *Expression* encodes a mathematical expression containing any of addition, subtraction, multiplication, division, negation, differentiation, parenthetical expressions, numerical values, and variables. *Expression* obeys the composite design pattern; each object contains a single string representing a mathematical operation and a list of child *Expressions* to which the operation is applied.

8.1.7.2 Depth-First Search

The main body of the state equation generation algorithm requires a series of *depth-first search* (DFS) algorithms to run. This kind of search algorithm starts from a specified source node and traverses nodes adjacent to ones it has already visited until it has accessed every available node in the graph in some order. DFS is named as such because the algorithm is more likely to access nodes a greater distance from the source earlier during runtime.

A depth-first search uses a stack object to determine which elements in the graph to visit next. The algorithm starts by pushing only the source node onto an initially empty stack. Then, the program runs through the following process until the stack is empty:

1. Pop a node off the stack and call it n.
2. If n has a neighboring vertex v that has not been accessed yet, push n and then v onto the stack; otherwise, do nothing.

We perform an example depth-first search on the bond graph in Figure 74. We will step through the DFS procedure as outlined below. Every time an element is pushed onto the stack, we will mark that element as "visited". Keep in mind that this traversal is only one of many valid depth-first searches. At any given step in the search, the algorithm can choose any unvisited neighbor, so we provide an example order below without loss of generality. In the following process, "element X" refers to the element labeled with integer X in blue in Figure 74.

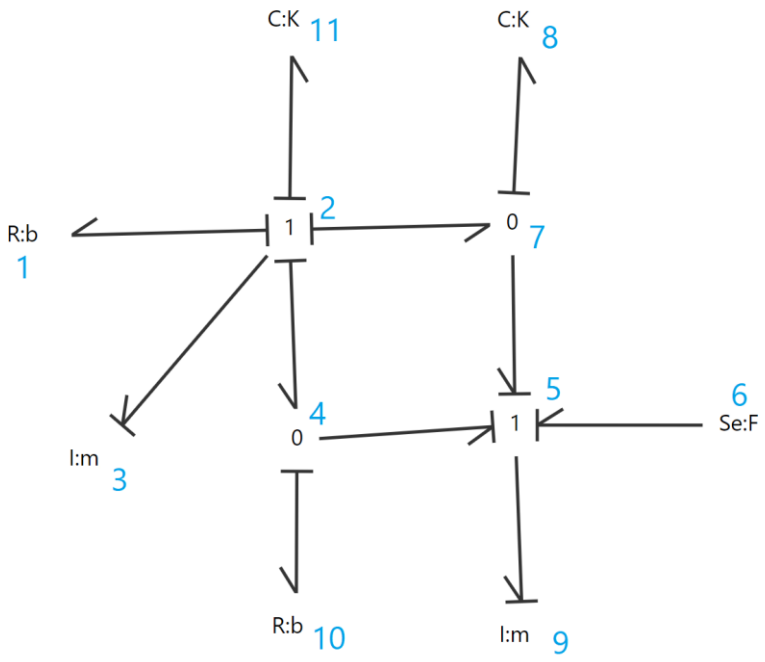


Figure 74: An example bond graph we will use to perform depth-first search and generate state equations. Bond labels have been removed for visibility, and the vertices are assigned numerical labels in blue to better follow the process.

1. Start with an initially empty stack and arbitrarily choose an element to push onto it. In this example, we will push *element 1* (an R-element) onto the stack.
2. Pop *element 1* off the stack and check if it has any unvisited neighbors. Since element 2 (a 1-junction) is an unvisited neighbor, push element 1 and then element 2 onto the stack.
3. Pop element 2 off the stack and check if it has any unvisited neighbors. Since element 3 (an I-element) is an unvisited neighbor, push element 2 and then element 3 onto the stack. Note that elements 4, 7, and 11 are unvisited neighbors of element 2, so they would also be valid choices for the next visited node.

4. Pop element 3 off the stack and check if it has any unvisited neighbors. Since its only neighbor is element 2, which is marked as visited, it has no unvisited neighbors. The algorithm does nothing.
5. Pop element 2 off the stack again and check if it has any unvisited neighbors. Since element 4 (a 0-junction) is an unvisited neighbor, push element 2 and then element 4 onto the stack.
6. At risk of repetition, we leave off explicitly writing out the remaining steps. A valid order of access from this point forward is elements 5, 6, 7, and so on. The depth-first search algorithm terminates here.

BoGL Web implements the modified depth-first search algorithm described in section 8.1.7.4 to form state equations. For this new search algorithm, we store bond graphs as *BondGraphWrappers*, described in the next section.

8.1.7.3 Bond Graph Wrapper

Bond graphs in BoGL Web are stored primarily in *BondGraph* objects, in which all elements are stored in a dictionary with string labels as the keys, and bonds are stored in a single list object. At each step in a depth-first search algorithm, the program needs to access all neighbors of the element it pops off the stack. If each bond is accessed from the *BondGraph* object to perform the depth-first search, finding all bonds incident to an element requires searching through every bond in the bond graph.

To make a more efficient system for getting all bonds incident to an element, we reorganized the *BondGraph* class into a *BondGraphWrapper* to modify how bonds are stored for depth-first search. *BondGraphWrapper* is a class nested within *BondGraph* that uses two dictionaries to store bonds, *bondsBySource* and *bondsByTarget*. Each key-value pair in the

bondsBySource dictionary stores an integer key corresponding to an element ID and a list of all *Bonds* with that element as the head. Similarly, *bondsByTarget* pairs an integer key and all *Bonds* with that element as the tail. With this implementation, a program can access all bonds incident to an element with a single call to *bondsBySource* or *bondsByTarget* instead of searching through the entire list of *Bond* objects. In *BondGraphWrapper*, elements in the bond graph are stored in the *elements* dictionary used by the original *BondGraph* object.

As an example, we will describe the bond graph in *Figure 74* as it would appear in a *BondGraphWrapper*, focusing specifically on bond storage. Take element 1 (an R-element). Notice that this element is the tail of exactly one incident bond (from element 2, a 1-junction). Then there is a mapping in *bondsBySource* from integer key "1", denoting element 1, to the list of all bonds with element 1 as the head. Since there are no such bonds, this list is empty. There is also a mapping in *bondsByTarget* from "1" to a list of length 1. This list contains only the bond from element 2 to element 1. As a second example, we can consider element 4 (a 0-junction). Then *bondsBySource* maps "4" to a list of length 2 containing the bond from element 4 to element 5 and the bond from element 4 to element 10. Likewise, *bondsByTarget* maps "4" to a list of length 1 containing only the bond from element 2 to element 4.

8.1.7.4 Causal Wrapper

The state equation generation algorithm uses a modified depth-first search algorithm, described later in this section. *CausalWrapper* is a wrapper object that tracks the elements in a bond graph visited for a particular execution of this search algorithm. Each *CausalWrapper* stores an element, a list of other *CausalWrappers* containing neighboring elements, and a flag indicating *flow* or *effort*. Each wrapper also stores an *Expression*; the *Expressions* will be combined to form state equations. While the child *CausalWrappers* will be created for all nodes

traversed, the starting element of the modified depth-first search will always be an I-, R-, or C-element.

As an example, we can create a *CausalWrapper* for element 10 (an R-element) in Figure 74. The one bond incident to element 10 has a causal stroke adjacent to element 4, its other endpoint, which means the flag in the *CausalWrapper* is set to *flow*. The program then begins the modified depth-first search on the graph, starting with a stack containing only element 10. We then consider element 10 to be the "starting location" of this search.

The only bond incident to element 10 is the bond to element 4, so that is the only bond considered. Since the *CausalWrapper* effort/flow flag is set to *flow,* the program checks that the causal stroke on the incident bond is *not* adjacent to element 10; since this is the case, element 4 is pushed on top of element 10 on the stack. Whenever an element is pushed onto the stack, a new *CausalWrapper* is created with that element and the same effort/flow flag; this new wrapper object is added as a child object to the previous *CausalWrapper*. The DFS then pops element 4 off the stack and continues.

At element 4, there are three incident bonds: the bond to element 10, the bond to element 5, and the bond from element 2. Of these, the bonds incident to elements 2 and 5 have non-adjacent causal strokes, so the modified depth-first search will access only these. Without loss of generality, we choose element 5 to push onto the stack on top of element 4. Using the same logic at element 5, we can push only element 9 onto the stack. At element 9, there are no incident bonds with non-adjacent causality – or any unvisited neighbors, for that matter – so we work backward to form initial expressions that will be converted into state equations later.

The *Expressions* formed during this stage are each assigned to *CausalWrappers*, which are associated with elements, but the expressions themselves represent the flow or effort on the

bond between the current element and the next element on the stack. At element 9, since the effort/flow flag of the *CausalWrapper* is set to "flow" and there are no child *CausalWrappers*, the initial equation for element 9 is set to "F9". The "F" here indicates flow, and "9" is the ID of the wrapper element. The DFS algorithm then pops element 5 off the stack. This *CausalWrapper* object has child *CausalWrappers* – specifically, the one associated with element 9 – so the program forms the element 5 expression using the child expression in the element 9 wrapper.

Since element 5 is a 1-junction, the flows of all incident bonds are equal. Then the algorithm picks the flow expression from the first child *CausalWrapper* – in this case, the element 9 wrapper – and sets the flow expression to that of the child. In general, causal strokes around a 1-junction are oriented so that there is only one such child for a flow expression at a 1-junction. For this example, the element 5 flow expression is set to "F9" accordingly.

The algorithm follows the same example when traversing from element 2, the other unvisited neighbor of element 4. Using the same traversal logic with respect to causal stroke adjacency, the only element traversable from element 2 is element 3. At this point, there are no unvisited neighbors, so the expression in the element 3 *CausalWrapper* is set to "F3". As element 2 is a 1-junction, all bonds incident to element 2 have equal flow, and the element 2 *CausalWrapper* expression is also set to "F3".

The search algorithm then returns to element 4. Element 4 is a 0-junction, indicating that the flows of all incident bonds sum to zero. The sign of each flow in the sum is determined from the flow direction of the bond to the previous element in the search, which for this example is element 10; this bond has element 4 as a source. If the bond to element X in a child wrapper has the same flow direction as the bond to element 10 – wherein they both have element 4 as a source – then the sign of that bond flow is negative; otherwise, the sign is positive. The bond

between elements 4 and 2 has element 4 as a tail, indicating “opposing” flow directions and thus a positive sign in the element 4 flow expression. The bond between elements 4 and 5 has element 4 as a source, indicating a negative sign. Then the element 4 flow expression is F3–F9”.

The last element to be popped off the stack for this search algorithm is element 10, which we recall is the starting location for the search. The program detects that element 10 is a leaf node with a child wrapper object. This can occur only if that leaf node was the starting element for the depth-first search. In this situation, the flow expression is set to that of the lone child wrapper, and the search terminates. In this example, the element 10 flow expression is set to that of element 4, “F3–F9”, concluding the search.

If the bond incident to the starting element has a causal stroke adjacent to the starting element, an effort expression is generated. In this situation, the program will use “E” as the first character in each variable instead of “F”, and the algorithm will create intermediate expressions in accordance with effort relationships instead of flow relationships. This primarily affects the behavior of the algorithm at 0- and 1-junctions. Specifically, the program now accounts for the fact that all efforts are equal at a 0-junction and that incoming effort equals outgoing effort at a 1-junction.

After any element is popped off the DFS stack, its ID is added to a “used” list. This list is used to ensure a tree structure is present in the depth-first traversal. Every time an element is added to the stack, the program checks whether the ID of the element is in the “used” list. If it is, it means that the element was accessed by two different paths from the initial element, indicating a cycle, which means that the bond graph cannot generate valid state equations. The program then throws an exception to exit equation generation.

All expressions formed from this part of the algorithm are stored in a dictionary. These expressions are stored as the “value” in each key-value pair. The key for that expression is the effort or flow variable from the starting element in the depth-first search. In our example, “F3-F9” is the value in the pair. Since the equation formed is for flow (F) and the search algorithm started from element 10, the key is expression “F10”. The list of all expressions is formed as follows:

- Key: “F10”; value: “F3–F9”.
- Key: “F1”; value: “F3”.
- Key: “F11”; value: “F3”.
- Key: “F8”; value: “F3–F9”.
- Key: “E9”; value: “E6+E8+E10”.
- Key: “E3”; value: “–E1–E11–E8–E10”.

8.1.7.5 Substitution of Intermediate Equations

We again return to our example from above to combine the equations. First, we provide the rest of the effort and flow equations from the leaf nodes in the bond graph. Each I-, R-, and C-element should have both an effort and a flow equation substitution. For each such element, one of these was provided in the depth-first search from section 8.1.7.4. We generate the other equation by remaking the effort-flow relationships given in Table 2 (in section 3.2.9) with the E and F variables we used in this example. Using E and F as effort and flow and X as the ID of the element, these relationships are as follows:

- For an R-element with associated value R, we have either EX=RX*FX or FX=EX/RX.

- For a C-element with associated constant C and energy variable Q, we have either FX=EX'*CX or EX=QX/CX.
- For an I-element with associated constant I and energy variable P, we have either EX=FX'*IX or FX=PX/IX.

For each I-, R-, and C-element in the graph, the dictionary will contain either key "EX" or key "FX", but not both. "EX" is the key for an effort expression; if "EX" is already present in the dictionary, then the effort expression was already provided during the modified depth-first search. Then the algorithm adds the respective flow equation, with key "FX", to the dictionary. Conversely, "FX" is the key for a flow expression; if "FX" is already present, then the flow expression was provided during the search. The algorithm will then add an expression with key "EX", representing effort. In our example, the algorithm makes the following additions to the dictionary in the example:

- For element 1 (an R-element), the dictionary already contains "F1", so a flow relation has already been provided, and the effort relation is needed. Then the program adds a mapping to the dictionary with key "E1" and value "R1*F1".
- For element 11 (a C-element), the dictionary already contains "F11", so a flow relation has already been provided, and the effort relation is needed. Then the program adds a mapping to the dictionary with key "E11" and value "Q11/C11".
- For element 9 (an I-element), the dictionary already contains "E9", so an effort relation has already been provided, and the flow relation is needed. Then the program adds a mapping to the dictionary with key "F9" and value "P9/I9".

We truncate the rest of the substitutions to avoid repetition.

- For element 10 (an R-element), the dictionary already contains “F10”, so the program adds the key-value pair (“E10”, “R10*F10”).
- For element 8 (a C-element), the dictionary already contains “F8”, so the program adds the key-value pair (“E8”, “Q8/C8”).
- For element 3 (an I-element), the dictionary already contains “E3”, so the program adds the key-value pair (“F3”, “P3/I3”).

The complete list of key-value pairs is then:

- (“F10”, “F3–F9”)
- (“F1”, “F3”)
- (“F11”, “F3”)
- (“F8”, “F3–F9”)
- (“E9”, “E6+E8+E10”)
- (“E3”, “–E1–E11–E8–E10”)
- (“E10”, “R10*F10”)
- (“E1”, “R1*F1”)
- (“E11”, “Q11/C11”)
- (“E8”, “Q8/C8”)
- (“F9”, “P9/I9”)
- (“F3”, “P3/I3”)

The program then iterates through all key-value pairs in the dictionary, and for each pair, it will search through all variables in the parse tree. If any variable matches a key in the dictionary, the program replaces the variable in the *Expression* with the value for that key and flags the key as “used”. All keys started as “unused”. We demonstrate this process on the example as follows:

- For pair (“F10”, “F3–F9”), the program finds “F3” and “F9” in the dictionary. Making the substitutions “F3”→”P3/I3” and “F9”→”P9/I9”, the “F3” and “F9” keys are marked as “used”, and the value at key “F10” is replaced with “P3/I3–P9/I9”.
- For pair (“F1”, “F3”), the program finds “F1” in the dictionary. Making the substitution “F3”→“P3/I3”, the “F3” key is marked as “used”, and the value at key “F1” is replaced with “P3/I3”. Note that the fact that “F3” is already marked as “used” does not affect this step.
- The same substitution from (“F1”, “F3”) is made on pair (“F11”, “F3”) to make (“F11”, “P3/I3”).
- The same substitutions from (“F1”, “F3–F9”) are made on pair (“F8”, “F3–F9”) to make (“F8”, “P3/I3-P9/I9”).
- For pair (“E9”, “E6+E8+E10”), the program finds “E8” and “E10” in the dictionary. Making the substitutions “E8”→“Q8/C8” and “E10”→“R10*F10”, the “E8” and “E10” keys are marked as “used”, and the value at key “E9” is replaced with “E6+Q8/C8+R10*F10”.

This substitution process continues with more steps until it terminates based on the conditions described above. All “used” entries have key variables that were substituted out for other expressions. As such, “used” keys are intermediate variables. Conversely, “unused” keys are variables that do not exist in any of the value expressions. These keys are the domain-generalized forms of the final state equations. After removing the “used” values from the dictionary, the contents of the dictionary will be as follows:

- (“E3”, “–R1*P3/I3–Q11/C11–Q8/C8–R10*(P3/I3–P9/I9)”)
- (“E9”, “E6+Q8/C8+R10*(P3/I3–P9/I9)”)

- (“F8”, “P3/I3–P9/I9”)
- (“F11”, “P3/I3”)

For a set of state equations to be useful to mechanical engineers, there must be at least one state variable. These are indicated in the equations above by “P” and “Q” variables. If there are no state variables in a bond graph for a system, that system is static, and the state equations for that system are meaningless. If the user generates a bond graph in BoGL Web for a system with no states, all keys in the dictionary will be marked as “used” by the time the substitution concludes. The dictionary will then be emptied, and no state equations will display.

All remaining key-value pairs in the dictionary are the domain-generalized forms of the state equations for this bond graph. We then make a final substitution to introduce domain-specific variables to the equations.

8.1.7.6 Domain-Specific Variable Substitution

The key in each dictionary entry is formatted as either “E” or “F” followed by the ID of the element. An “E” indicates an effort equation, meaning that the differentiated state variable is a *generalized momentum*, or “P” variable. An “F” indicates a flow equation, meaning that the state variable is a *generalized displacement*, or “Q” variable. In the first step of the domain substitution, the program iterates through all keys and replaces each instance of “EX” – with “X” the element ID – with “PX’” and each instance of “FX” with “QX’”.

In the second step, all variables in both the key and value expressions are replaced with domain-specific symbols. In the example generation from *Figure 74*, we make the following substitutions to show membership in a particular domain. These substitutions are shown in Table 13. Each variable is formatted as a letter indicating the unit (e.g., “m” for “mass”) and an integer indicating the ID of the element, which will be written as X in the table. All substitution values

are lone variables except for the mechanical spring constants. Since the domain-generalized variables assume that they are *compliances*, the algorithm substitutions the inverse of each spring constant to convert it to a *stiffness* constant.

Table 13: A modified symbol table to show substitutions.

Original Symbol	*Translational Symbol*	*Rotational Symbol*	*Electrical Symbol*
EX	FX	τX	VX
FX	vX	ωX	iX
RX	bX	DX	RX
CX	1/KX	1/KtX	CX
IX	mX	JX	LX
TX	DX	RX	TX
GX	rX	rX	rX
PX	pX	LX	ϕX
QX	xX	θX	qX

The resulting equations use standard mechanical engineering symbols. Making the above substitutions, the example equations from Figure 74 are as follows:

- (“p3’”, “–b1*p3/m3–x11/(1/K11) –x8/(1/K8) –b10*(p3/m3–p9/m9)”)
- (“p9’”, “F6+x8/(1/K8)+b10*(p3/m3–p9/m9)”)
- (“x8’”, “p3/m3–p9/m9”)
- (“x11’”, “p3/m3”)

For each pair, the program concatenates the key and value with an “=” sign in between and sends the resulting state equations to the frontend to display:

- “p3’=–b1*p3/m3–x11/(1/K11) –x8/(1/K8) –b10*(p3/m3–p9/m9)”
- “p9’=F6+x8/(1/K8)+b10*(p3/m3–p9/m9)”
- “x8’=p3/m3–p9/m9”

- “x11’=p3/m3”

8.1.7.7 Pseudocode

We provide pseudocode to describe the above procedure. Our first step is to procure the *BondGraphWrapper*. In the code below, ‘E’ is the element dictionary, ‘BS’ is *bondsBySource*, and ‘BT’ is *bondsByTarget.*

```
Make dictionaries E, BS, BT
For each element e in G
        Add map in BS from e to an empty list
        Add map in BT from e to an empty list
For each edge e in G with head s and tail t
        Add map in BS from s to t
        Add map in BT from t to s
```

We then make the *CausalWrappers.* For each leaf node, check if the causal stroke on the incident bond is adjacent to that leaf node. If it is,

```
Push the chosen "starting" element onto the stack
While the stack is not empty
        Pop element e off the stack
        If e has an unvisited neighbor n with adjacent causality
                Push e onto the stack
                Create a new wrapper with element n and add it as a child wrapper to the 'e' wrapper
                Push n onto the stack
        Else if the list field of wrappers in the 'e' wrapper is nonempty
                If e is a leaf node or 0-junction
                        Set the Expression in the wrapper to the Expression of the child wrapper
                Else if e is a 1-junction
                        Let x be zero
                        For each child wrapper g of e
                                Add the Expression for g to x
                        Set the Expression in the wrapper to x
```

```
                Else if e is a transformer
                        Set the Expression in the wrapper to ("T" + the ID) * the Expression of the
child wrapper
                Else
                        Set the Expression in the wrapper to ("G" + the ID) * the Expression of the
child wrapper
        Else
                Set the Expression in the wrapper to "E" + the ID
Add map in subs from j -> Expression in wrapper for j
```

If the causal stroke is not adjacent to the starting element,

```
Push the chosen "starting" element onto the stack
While the stack is not empty
        Pop element e off the stack
        If e has an unvisited neighbor n with non-adjacent causality
                Push e onto the stack
                Create a new wrapper with element n and add it as a child wrapper to the 'e' wrapper
                Push n onto the stack
        Else if the list field of wrappers in the 'e' wrapper is nonempty
                If e is a leaf node or 1-junction
                        Set the Expression in the wrapper to the Expression of the child wrapper
                Else if e is a 0-junction
                        Let x be zero
                        For each child wrapper g of e
                                Add the Expression for g to x
                        Set the Expression in the wrapper to x
                Else if e is a transformer
                        Set the Expression in the wrapper to ("T" + the ID) * the Expression of the
                        child wrapper
                Else
                        Set the Expression in the wrapper to ("G" + the ID) * the Expression of the
                        child wrapper
        Else
                Set the Expression in the wrapper to "F" + the ID
```

```
Add map in subs from j -> Expression in wrapper for j
```

After creating the *CausalWrappers*, we enter the equation substitution step:

```
Set prevNumUsed to -1
Set numUsed to 0
While numUsed > prevNumUsed
        prevNumUsed = numUsed
        For each key-Expression pair (e,w) in subs dictionary
                 For each variable v in w
                       If v = e
                               Replace v with Expression x at key v
                               Mark (v,x) as used
                               Add 1 to numUsed
For each key-Expression pair (e,w) in subs dictionary
        If (e,w) is marked used
                Remove (e,w)
```

We then make our final, domain-specific variable substitution step. “domainSubs” is the dictionary with the mappings described in Table 13.

```
For each key-Expression pair (e,w) in subs dictionary
        If e name = 'E' + number
                Replace e with 'P' + number + '
        Else
                Replace e with 'Q' + number + '
        For each variable v in e, w
                Replace v with map from v in domainSubs
```

8.2 User Interface Implementation

In this section, we will discuss the implementation details of the user interface. The goal of our project was to replicate the user interface of BoGL Desktop, so our implementation will reuse all of the design decisions that were made by the 2020 MQP team (Courville et al., 2020).

8.2.1 BoGL Web User Interface Control Flow

In planning out the front-end of our application, we created the UI control flow diagram found in Figure 75. This diagram provides a high level view of interactions between the user and the UI. The blue squares below represent sections of the code, the gray squares represent objects and sources of user interaction, and the arrows detail actions that are triggered by objects or user actions. This section will discuss the major sections of the diagram according to the type of user interaction that triggers the action.

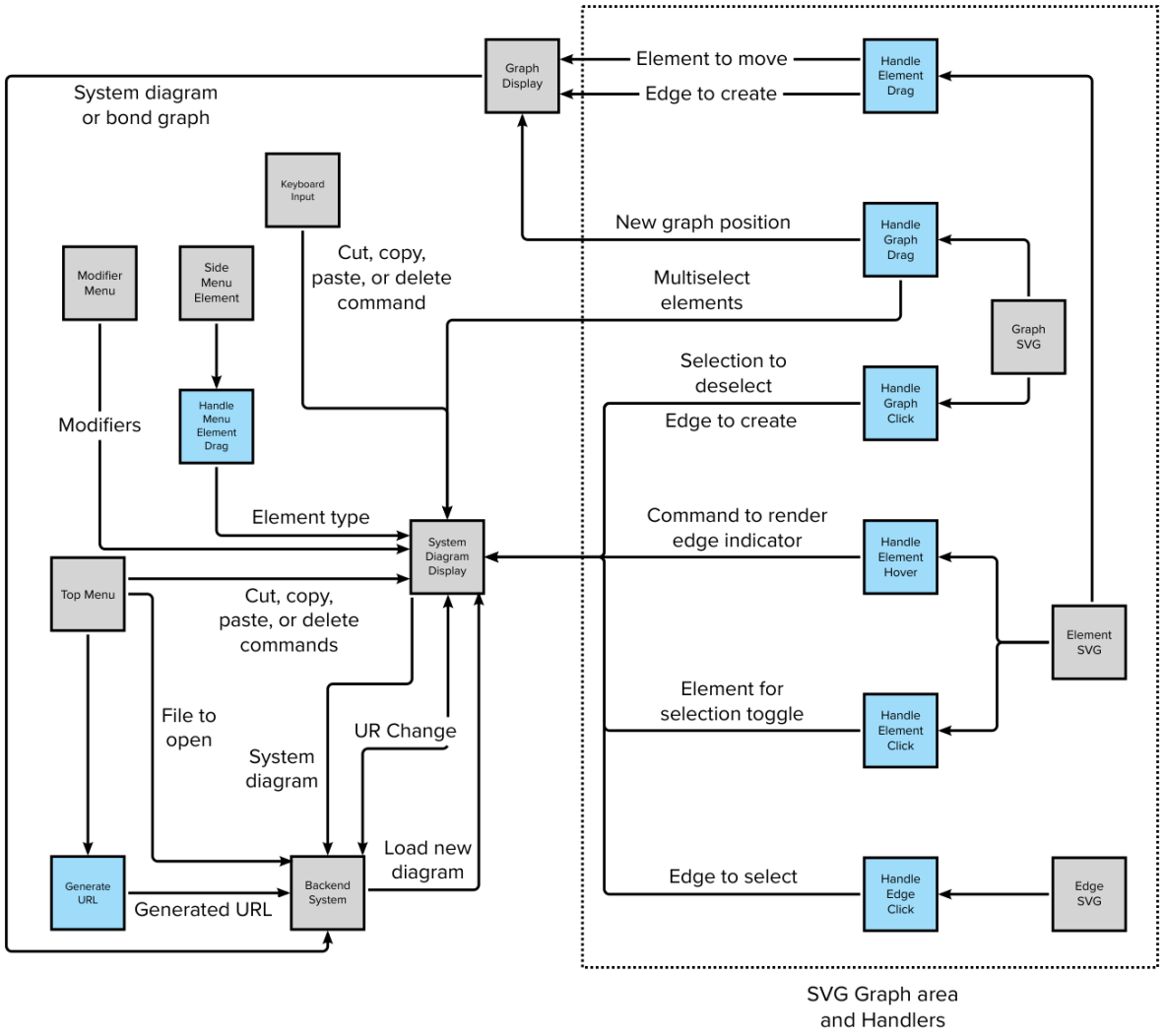


Figure 75: BoGL Web user interface control flow diagram.

One major site of user interaction in BoGL Web is the top menu. Figure 75 shows that the top menu connects to the graph display, the system diagram display specifically, and the backend. The File menu interacts with the system diagram, issuing a command to turn the system diagram into a string that can be saved to the backend. The opposite of this interaction occurs when a file is opened, in which case the File menu triggers a system diagram string to be sent from the backend to the system diagram display. Image export, which is also triggered from the

File menu, requests the current state of the graph display so it can be converted to an image. Likewise, generate URL requests the current system diagram so that it can be compressed into a URL query string. The edit menu in the top menu can trigger cut, copy, paste, delete, and Undo/Redo commands being sent to the system diagram, which changes the state of the diagram.

Another method for user interaction in BoGL Web is keyboard shortcuts. There are six shortcuts: Ctrl + C, Ctrl + X, Ctrl + V, Ctrl + Z, Ctrl + Y, and Ctrl + A. These shortcuts allow a user to copy, cut, and paste selections, undo, and redo actions, and select everything in a graph display, respectively. The shortcuts therefore send commands to the graph display, in the case of undo/redo and selecting all, or the system diagram display specifically, in the case of the rest.

The Side Element menu only interacts with one aspect of BoGL Web: the system diagram display. When users drag an element from the Side Element menu into the system diagram display, the element type being dragged is sent from the menu to the display so the element can be rendered.

Another menu that interacts exclusively with the system diagram display is the modifier menu. This menu sends commands to change velocities and modifiers to the system diagram display as the user interacts with the menu. It also receives commands from the system diagram display to disable or enable modifiers and to disable or enable the velocity selector, based on what elements and edges are selected.

The SVG graph area can accept user interaction in several ways. The user can drag an element or the background of the canvas, click an element, edge, or the background of the canvas, and hover over an element. Figure 75 shows these interactions in detail. One note is that a few actions, dragging elements and the graph, can affect both the system diagram display and the bond graph display, while all other SVG interactions just affect the system diagram display.

These interactions send commands for actions like creating edges, changing element selections (see Section 8.2.3.9 for a more detailed description of multi-selection), and rendering the symbol that shows edge compatibility.

8.2.2 User Interface Menus

In this section, we discuss how various menus in the user interface were implemented. The menus allow the user to access all system functionality that is not built into the graph areas, such as editing modifiers and velocities, adding elements, and exporting and importing system diagrams. Since these menus are simple and static, all of them use Ant Design components.

8.2.2.1 Top Menu Implementation

The top menu of BoGL Web includes three main dropdown menus, the File, Edit, and Help Menus, and an array of icon buttons. As noted in Section 6.5.1, we used Ant Design menu components to make the three dropdown menus. The styling of these menus matches the rest of the UI, so it is preferable for us to use them, but we discovered that Ant Design submenus have some click action bugs that make them less preferable. When a user clicks into a nested Ant Design submenu, the submenu before the menu they clicked often disappears after a slight delay, as seen in Figure 76. This does not prevent the user from accessing any part of the menu, but it does keep them from backtracking into a parent submenu. For example, clicking into the Mechanical Translation submenu would close the submenu showing the three categories of example files, preventing the user from backtracking, and seeing the Electrical example files. This is clearly not ideal and is difficult to fix from the Ant Design side since we do not have an effective way of editing Ant Design's JavaScript. As a result, we decided to use the HTML and

CSS generated by the Ant Design menus but re-create JavaScript that opens and closes the menu and submenus.

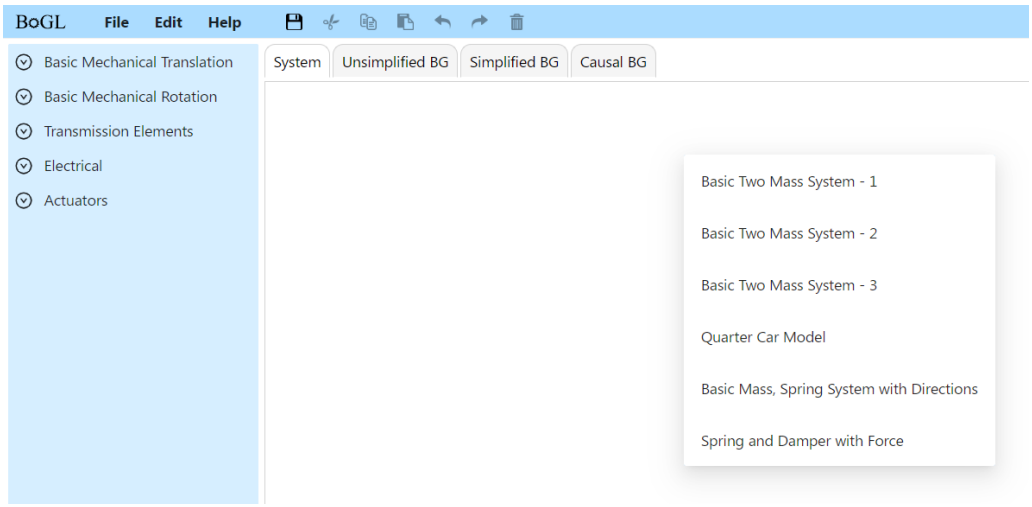


Figure 76: Bug where part of the Help menu is disconnected from its parent menus.

Each Ant Design menu or submenu has two parts: a button that triggers the menu to open and the menu itself. The process of reassigning control of the Ant Design menus to our TypeScript code begins with inserting the HTML for the menus into the DOM (Document Object Model) tree. This occurs when an Ant Design click action on the menu's button is triggered. Since our goal is to add new click actions to the menus' buttons that will be called when the user opens a menu, we need to open all the submenus so that the menu buttons within them can be assigned click actions. On page load, the three top menu buttons for the File, Edit, and Help menus are clicked and assigned click actions. The code then monitors whether the submenus have rendered yet and, when they do, opens any submenus under the new submenu. The menus are identified using an object that maps their IDs in the DOM tree to an index number that can be used to access them. Another menu maps menu indices to their submenus, which lets

the recursive function know when to stop. To avoid letting the user see that all the menus are open when they are programmatically opened on page load, their CSS display style is set to "none."

One challenge in changing the Ant Design menus' CSS to hide and show them is that any time an Ant Design click action for the menus runs, the class list of the menu is reset. This wipes out any CSS classes applied by our code, and it occurs not only when the Ant Design menu click action is called, but also whenever the user clicks elsewhere on the page. At first this made it difficult to apply styles consistently to the menu HTML objects, but then we started using custom HTML attributes to style the menus instead of CSS classes. This prevented our CSS changes from being removed when the Ant Design classes were added and made it easy for us to override the Ant Design styling where needed. We used an attribute "hidden menu" with values of "true" and "false" to indicate whether the menu should be hidden or not.

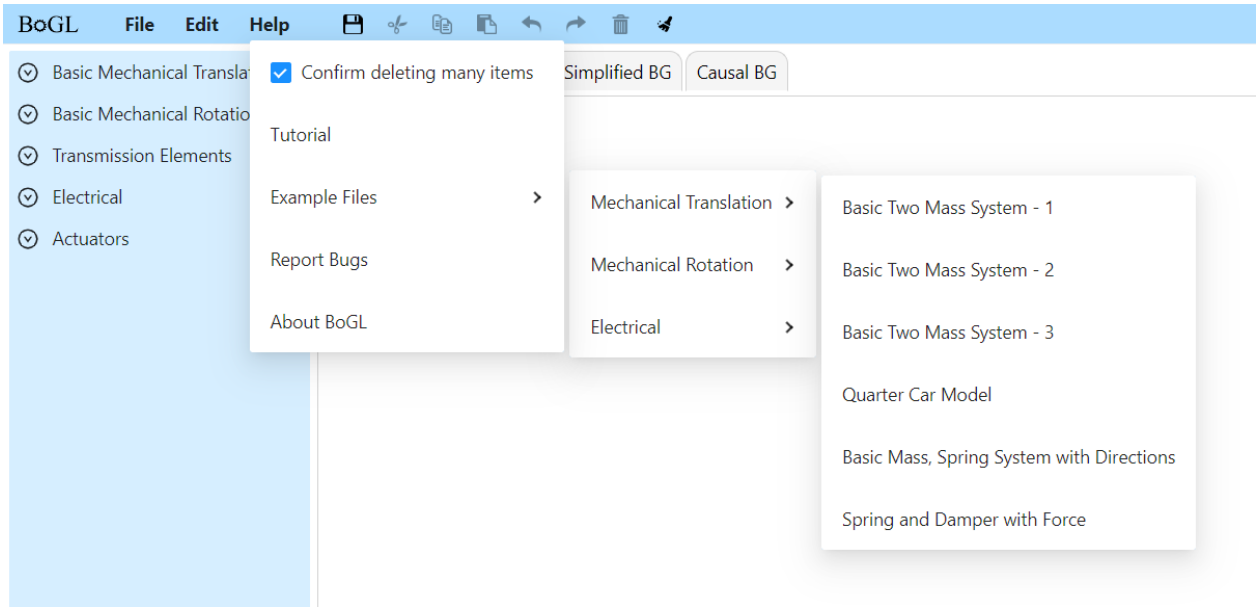


Figure 77: Connected menu after the menu bug was fixed.

On clicking a submenu button, four things occur. First, the visibility of the menu corresponding to the button is toggled. Second, if the menu is not one of the top three menus, its CSS is changed to place it correctly relative to its parent menu. This is necessary because previously the Ant Design code was responsible for placing the menu, so we needed to replicate that functionality. Third, if the menu was previously hidden, the click action checks whether its submenus, if it has any, have been assigned click actions. This will only happen once, on first clicking a parent menu, and the assignment of click actions to submenus is recorded in the object that maps parent menus to their submenus. Finally, the click action cycles through all menus that are not the menu associated with the button, or one of its parents, and hides them. The current menu's parents are determined using the parent to submenu map. All other menus are hidden because only one menu and its parents should be open at a time. Figure 77 shows how submenus were displayed once the menu bug was fixed.

8.2.2.2 Element Menu Implementation

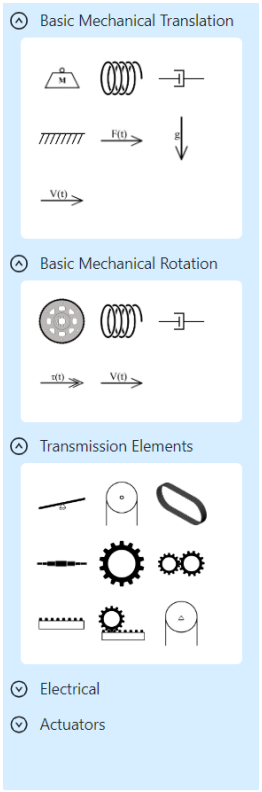


Figure 78: The element menu in BoGL Web.

The element menu pictured in Figure 78 uses Ant Design Collapse components to separate each category of elements that can be added to a system diagram. The Collapse component uses custom CSS to match the BoGL Desktop collapsible sections. The element options are generated programmatically on page to tie them to a list of available elements in TypeScript. This singular list of elements makes it easy to add a new element to BoGL Web and keep track of the ones that are already there. Each element option is an image tag that displays the element's SVG with an event listener that sets the system diagram's dragging element variable to the current element when the element is dragged out of the menu. The dragging

element variable lets the system diagram canvas know whether to add an element to the canvas on mouse up, and if it should, which element to add.

8.2.2.3 Modifier Menu Implementation

The modifier menu, shown in Figure 79, uses a styled Ant Design Collapse element for its main container. This allows the menu to collapse by clicking on its header and to switch its arrow direction as the menu opens and closes. The modifier menu has two main sections: the modifier checkboxes and the velocity selector. While the functionality here is the same as BoGL Desktop, we will explain how the menu works because it is integral to our implementation of the menu.

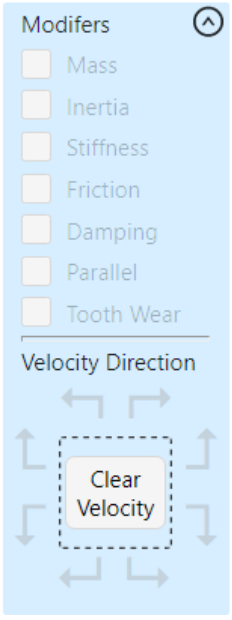


Figure 79: The modifier menu.

The first part of the modifier menu, the checkboxes, allow the user to assign all modifiers other than velocity. These checkboxes update to reflect the modifiers for the current selection,

and by interacting with the checkboxes, the user can set the modifiers for the selection. One keynote is that there are restrictions on which elements can receive certain modifiers. First, since edges only support the velocity modifier, they are not affected by the modifier checkboxes. Second, each element only supports a few modifiers, so only certain elements are eligible for receiving a given modifier. As such, when a modifier is added to the selection, only eligible elements are given the new modifier.

The modifier selection checkboxes have four states that depend on the current selection. If none of the elements in the selection support a given modifier, it is disabled, making it unclickable and greyed out. If any elements support the modifier but no elements, have it, the checkbox is enabled but unchecked. If some elements have the modifier but other elements do not, the checkbox is enabled and in a partially filled, intermediate state. Finally, if all elements in the selection have a modifier, its checkbox is enabled and checked. Whenever a modifier is checked, in a full or intermediate way, clicking the checkbox again will result in unchecking the checkbox, even when the modifier can be applied to all elements in the selection. Given this, certain selections may cause a modifier checkbox to flip between an unchecked and intermediate state. Whenever a modifier checkbox is unchecked, the modifier is removed from all elements in the selection. Ant Design checkboxes support enabling intermediate values and disabling the checkbox with booleans, so we were able to replicate the functionality of BoGL Desktop using only the Ant Design checkboxes, as shown in Figure 80.

Figure 80: Example of a disabled checkbox and an intermediately filled checkbox.

The second part of the modifier menu allows the user to assign velocities to the current selection. This area has a button labelled Clear Velocity surrounded by eight arrows representing the eight velocity options. Each arrow is an Ant Design button with modified CSS that preserves the button press animation while hiding the button background. Unlike the other modifiers, velocity can be applied to any element or bond. Because of this, the velocity buttons have no intermediate state. Another change from the other modifiers is that the entire velocity menu represents one modifier value, so only one velocity option can be selected. Selecting a new velocity when one is already selected results in the current selection being unselected and the new velocity being selected in its place. Clearing the velocity with the center button deselects the currently selected velocity. The only time the velocity buttons are disabled is when there is no current selection.

8.2.3 SVG Graphs

Most of the user's interaction with BoGL Web happens in the SVG graph area. This is the rectangular area on each of the four tabs that holds the system diagram or bond graph. In this section, we will describe how the UI for this section is generated and how user actions update the graph.

8.2.3.1 Generic Graph Display Implementation

The system diagram and the bond graph display areas have several similarities. These include moving nodes, updating edge end positions to move with nodes, panning, zooming, and multi-select. These aspects of the graph display are almost identical between a system diagram and a bond graph. Given this, we decided to make the system diagram and bond graph display objects subclasses of a shared parent class. This graph displays objects that use generic elements

and edge objects to control the positions of elements and objects as they are dragged around. This object also captures shared keyboard inputs for selecting all objects, using the arrow keys to pan the graph, undo/redo, and deleting elements. Additionally, it captures click events for multi-select. Finally, the graph parent object handles all graph panning and zooming, including zooming with the zoom slider.

8.2.3.2 System Diagram Display Additions

The system diagram display object adds extra functionality and visual rendering to the system diagram. This display object is a subclass of the graph element object that holds extra information about system diagram elements, such as applied modifiers, type of element, and velocity. This graph element subclass also has a copy function to facilitate copying a selection in a system diagram. One major addition to the base graph display in the system diagram display object is the ability to render each element with an image representing its type, a box around the image, and all necessary click actions for an element. The box around the image turns blue when the node is selected. The only visual aspect added to edges in the system diagram is the ability to show a velocity arrow, which is not needed in bond graphs.

In addition to visual changes, the system diagram display object adds extra functionality to the base graph display. One addition is key combinations for copying, cutting, pasting, and deleting elements and edges, which is not something you can do in bond graphs. Another change is allowing system diagram elements to update based on the modifier and velocity menus and to display the update visually. Additionally, the system diagram display object can create system diagram elements when an element is dragged into the graph area and can create edges when the user clicks or drags on the edge of system diagram elements.

8.2.3.3 Bond Graph Display Additions

Unlike the system diagram display, the bond graph displays make only visual changes to the base graph display, not functional changes. In a bond graph display, the graph nodes are text nodes instead of images with boxes around them. This text node displays the name of the bond graph node and turns blue when the node is selected. Originally, the base graph display object connected the ends of edges to the center of their node, which worked well for system diagram nodes, but not for bond graph nodes where the edge was visible beneath the text. As a result, we made a function that can calculate the position along the edge of a text node that an edge should connect to point toward the center of the node while not actually extending to the node's center. This function takes in a source position, target position, and the target node's width and height. Finding the endpoints for one edge requires calling this function twice, once going from the source to the target and once going from the target to the source. Later, we determined that we could use this method of drawing edges for system diagrams too, because ending the edge at the border of the element box prevented area multi-selection from selecting edges hidden underneath system diagram elements. Given this, we used this function as the default way of drawing edges in the base graph display object. We further modified system diagram displays to snap the edges to the center of the node when the element is being dragged, since calculating the edge end position constantly while dragging causes lag.

The only other visual change that the bond graph display adds to the base graph display is endings for edges. These endings include a causal stroke with no arrow, a casual stroke with an arrow, and an arrow with no causal stroke. An edge ending may also have none of these markings. These endings are created using the marker-end and marker-start SVG attributes. These attributes take a pre-defined marker shape and place it at the end of a stroke. The markers

for an edge are changed depending on the type of edge it is, which is dictated by the backend in the bond graph object it sends to the frontend. Since using a blue marker for selected bonds requires a new marker, there are six markers to apply to one end of an edge.

8.2.3.4 Pan and Zoom Implementation

One benefit of using D3.js is that it can automatically handle zooming and panning done with the mouse. Mouse interactions with the UI are captured using mouse events, which hold information about what sort of mouse interaction occurred and what element it targeted. These mouse events are caught using the *zoom* event, which is called any time the user clicks and drags the graph display, scrolls their mouse wheel, or pinches in or out on a trackpad. The *zoom* event is sandwiched between the *zoomstart* and *zoomend* events. These bookend events are called once while *zoom* is called continuously as the user zooms or pans the graph. Whenever *zoom* is called, the event it generates has two parameters, *translate* and *scale,* that indicate how the graph should be translated from the origin and scaled. When the user changes the scale, D3.js automatically calculates the correct graph translation to make the graph appear to zoom in on the mouse's current location.

While zooming triggered by mouse events is easy thanks to D3.js, programming the zoom slider in the zoom menu was much more challenging. This slider zooms in on the current center of the user's view instead of the current mouse position, since the mouse needs to be on the zoom slider while dragging it. As this action does not trigger a zoom event, we had to calculate the translation offset for this zoom ourselves. The equations to calculate the x and y translation are as follows, where *ps* is the previous scale (range 25 - 175), *s* is the new scale (range 25 - 175), *ix* is the initial graph x position, *iy* is the initial graph y position, *x* is current graph x position, and *y* is current graph y position:

$$\text{xTrans} = \text{x} - \frac{(ps - s) * (x - ix)}{ps + (s > ps\,?\,1 : -1)}$$
$$\text{yTrans} = \text{y} - \frac{(ps - s) * (y - iy)}{ps + (s > ps\,?\,1 : -1)}$$

In these equations the ? and : are ternary operations. If the statement before the ? is true, then the value before the : will be used. If the statement is false, then the value after the : will be used.

8.2.3.5 Element Creation Implementation

In BoGL Web, there is only way for a user to add an element to a preexisting system diagram graph. A menu on the left side of the application holds all available elements divided into type categories. When the user wants to add a system diagram element, they can drag the element from the menu into the graph area, where it will appear at their mouse position. This is accomplished with a mouse event tied to each image in the menu that triggers when the image is dragged. The event sets a variable *draggingElement* in the SystemDiagramDisplay object to an object representing the given element type, for example, an object representing mass or spring. When the user ends a drag anywhere on the screen, *draggingElement* is set to null unless the drag ends within the graph. If the user drags into the graph, the current *draggingElement* is added to the graph at the mouse's position.

8.2.3.6 Edge Creation Implementation

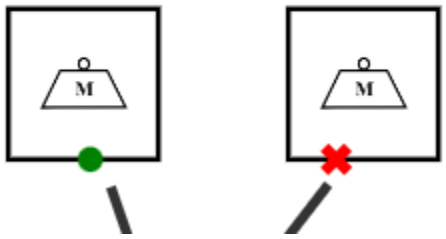


Figure 81: Example of green circle and red X indicators.

In BoGL Web, there are two ways for a user to make an edge. One path to creating an edge is clicking the edge of a system diagram element and clicking anywhere inside another element. The other method for making an edge is to drag from the edge of an element to anywhere inside another element. When the user's mouse is within the outer edge of a system diagram element, a green circle or red X appears. Figure 81 shows the green circle and red X on a mass element. The green circle indicates that the two elements can be connected with an edge, while the red X indicates that the elements are incompatible and cannot be connected. If an edge is not currently being created, this will always be a green circle. Once the user clicks or drags in this outer edge, a line stretching from the center of the element to the cursor will follow the mouse's movements, indicating that an edge can be created. If the user clicks in another element that is compatible with the source element, an edge will be created between the elements and the moving line will disappear. If the user clicks on the canvas, the source element, or an incompatible element, the moving line will disappear, but no edge will be generated. Figure 82 shows a full state diagram detailing how user actions affect the symbol (green circle or red X) showing edge availability and the current state of edge creation.

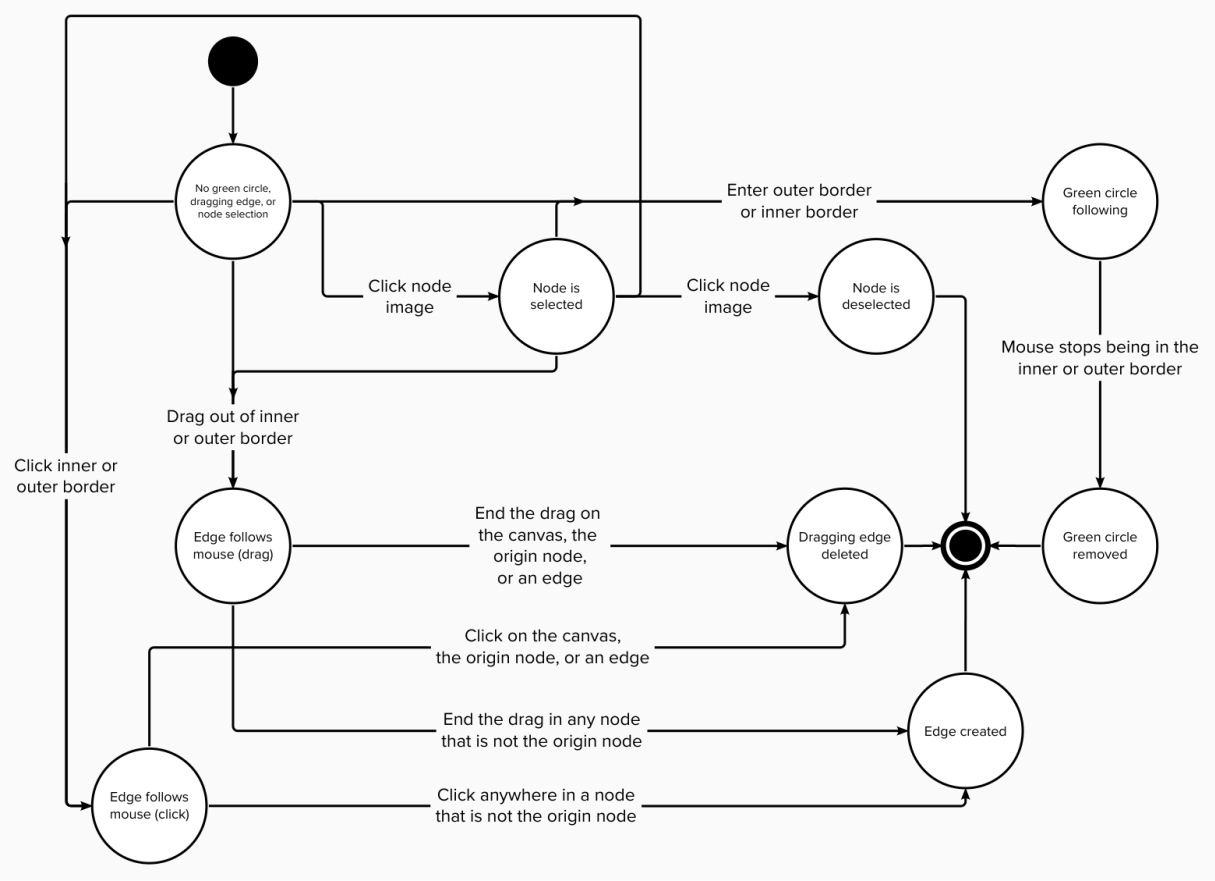


Figure 82: State diagram showing edge creation flow.

One important feature of edge creation is the ability to reject edges which cannot be created. To implement this feature, we first investigated what elements would be valid to connect. We found that there were two times when an edge could be rejected. The first is if an element already has too many connections, and the second is if the two elements trying to be connected are not in the same compatibility group.

Table 14 and Table 15 show the maximum number of connections elements can have and the connection compatibility groups for elements. We first have the maximum number of connections that an element can have:

Table 14: Table containing the maximum number of edges certain elements can have.

Element Name	Maximum Edges
System_E_Capacitor	2
System_E_Resistor	2
System_E_Inductor	2
System_E_Ground	2
System_E_Transformer	4
System_E_Junction	4
System_MT_Spring	2
System_MT_Damper	2
System_MR_Damper	2
System_MR_Spring	2
System_MR_Shaft	2

Next, we have the compatibility groups that determine which categories of elements can connect to each other. Elements can only be connected to elements in their category.

Table 15: Table containing the compatibility groups for various elements.

Compatibility Groups			
Mechanical Translation	**Mechanical Rotation**	**Electrical**	**Oscillators**
System_MT_Mass	System_MR_Spring	System_E_Capacitor	System_O_VC_Transducer
System_MT_Spring	System_O_PM_Motor	System_O_PM_Motor	System_O_PM_Motor
System_MR_Belt	System_MR_Rack	System_E_Current_Input	
System_MR_Pully	System_MR_Gear	System_E_Voltage_Input	
System_MR_Lever	System_MR_Velocity_Input	System_E_Inductor	
System_MR_Rack	System_MR_Pully_Grounded	System_E_Transformer	
System_MT_Velocity_Input	System_MR_Torque_Input	System_E_Junction	
System_MT_Damper	System_MR_Belt	System_E_Ground	
System_MT_Force_Input	System_MR_Shaft	System_E_Resistor	
	System_MR_Pully		
	System_MR_Lever		
	System_MR_Flywheel		
	System_MR_Damper		

We were able to embed the restriction on the number of connections within the ElementType object on the frontend. We added a property to the constructor of ElementType

which took in a number. We then would count the current number of edges connected to a given element in the system diagram and check if we exceeded this value set in ElementType. If we did exceed this value, we would not allow the user to create the edge and show a red X instead of the green circle.

We created several compatibilities sets to restrict which elements can be connected to each other. These sets represent the major classes of elements including translational, rotational, electrical, and more. The sets contained the IDs of any elements which could connect to each other within these categories. To check that two elements are compatible, we check if any set contains both. If such a set exists, we know the elements are compatible. As a note, while it is not in the table, we consider the gravity element its own compatibility group that cannot connect to any other element. BoGL Desktop used maps to define compatibility groups, however we found that using sets simplified the setup while still allowing us to check compatibility in constant time.

8.2.3.7 Cut, Copy, and Paste Implementation

Cut, copy, and paste are features in BoGL Desktop that allow the user to duplicate a selection of elements and edges. These functionalities can be accessed through the Edit menu, the icon buttons on the top menu bar, or through keyboard commands. When the user issues a copy command, duplicates of the selected element and edge objects are created. These new objects are given new IDs and are stored in a variable in the SystemDiagramDisplay object. The only difference between the copy and cut actions is that the cut action also deletes the current selection after copying. When the user issues a paste command, the most recently copied elements and edges are inserted back into the system diagram at a fixed offset from their original copy position. The pasted items are then copied again and re-stored in the copy lists so that the

user can paste the selection again without adding elements with duplicate keys to the system diagram. Figure 83 shows a system diagram that has been copied and pasted. Note that the copied system diagram is selected after pasting.

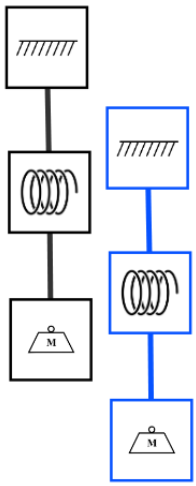


Figure 83: A system diagram copied and pasted.

While the menu items and icon buttons are easy to connect to the copy, cut, and paste functions, capturing multi-key keyboard events proved to be a little more difficult. Two variables in the SystemDiagramDisplay object track whether the Control (Ctrl) key is being held down and track the last key to be pressed. Duplicate key presses are ignored. Checking whether Ctrl has been pressed and looking at the last key to be pressed lets the code process key combinations like Ctrl+C for copy, Ctrl+X for cut, Ctrl+V for paste, and Ctrl+A for select all. Our implementation allows a user to hold down Ctrl and press multiple keys in sequence to complete multiple actions. This functionality allows advanced users to easily copy a selection and paste it multiple times without toggling the Ctrl key with every action.

8.2.3.8 Delete Element or Edge Implementation

The user can delete the current selection using the Edit menu or icon buttons on the top menu bar, or by pressing ‘d’ or backspace. When the user issues a delete command, all elements and edges in the current selection will be deleted. If any element in the selection has an edge connected to it that is not selected, that edge will also be deleted, as it can no longer connect the elements at its ends.

8.2.3.9 Select Element or Edge Implementation

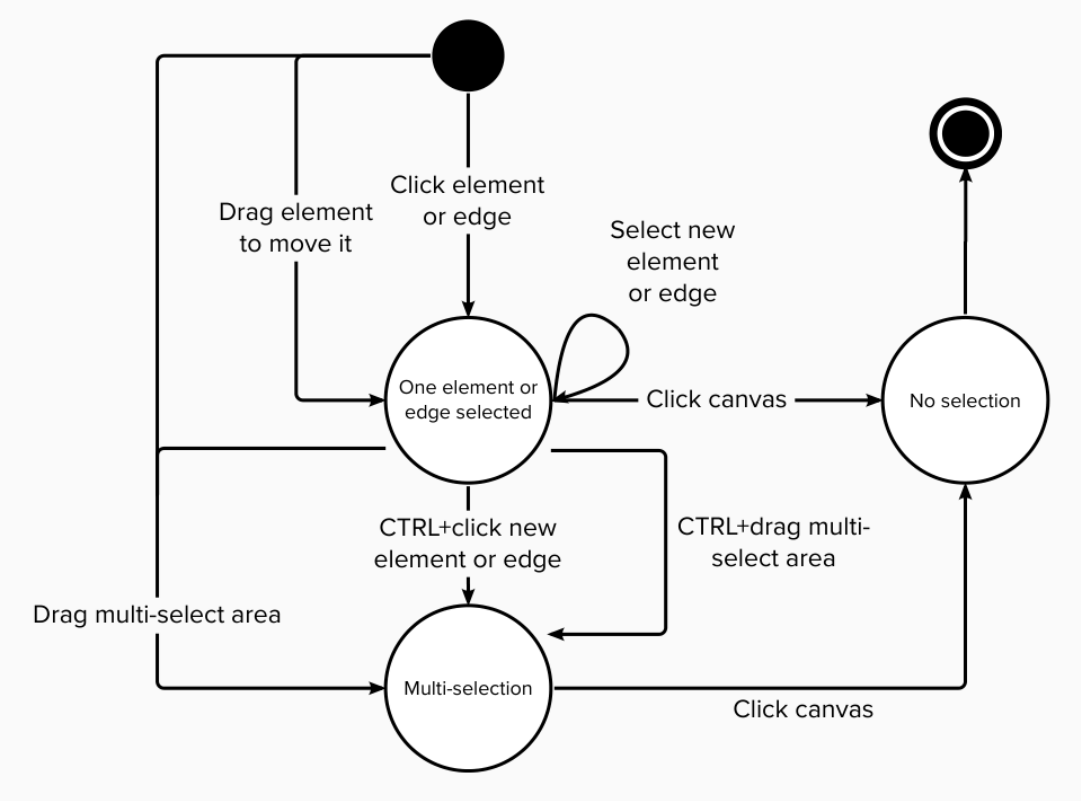


Figure 84: Selection state diagram

There are two methods for selecting elements and edges in BoGL Web. A user can click an individual element/edge to highlight only that object. They can also drag with the left mouse click to select the contents of a rectangular area on the graph. The user can hold down Ctrl while performing either of these actions to change the way the selection is affected. As Figure 84 shows, there are four ways to make or edit a selection. A user can click an individual element or edge outside of the current selection to clear the selection and select the new element/edge. By holding Ctrl then clicking an element or edge, the selection of that element / edge is toggled without clearing the current selection. For example, Ctrl clicking an element in a selection will remove it, while Ctrl clicking an element outside the selection will add the element to the selection. Figure 85 shows an example of multi-selection in BoGL Web.

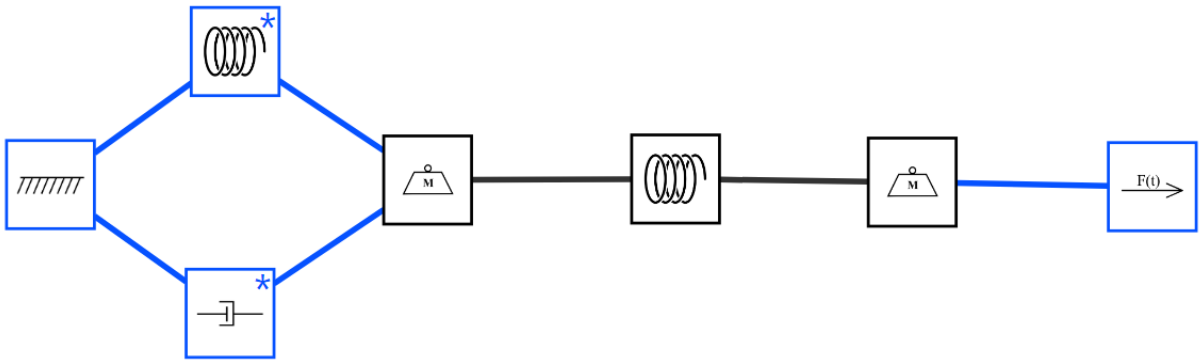


Figure 85: Multi-selection example.

The other two selection methods use area selection. If a user drags to select area without holding Ctrl, the selection will be cleared and replaced with all elements and edges in the selection area. This includes any edge or element that overlaps any part of the selection area. If the user holds Ctrl while dragging a selection area, all elements and edges in the area will have

their selection toggled without affecting selected elements outside the selection. Whenever the user clicks the canvas, the selection will clear.

8.2.3.10 Multi-Elements

In BoGL Desktop there are several elements which we have classified as multi-elements. While they show up as one element in the element menu, these multi-elements are collections of common element combinations connected. Thus, when one is dragged onto the canvas, it generates multiple connected elements. The gear pair and the rack and pinion are examples of multi-elements in BoGL Web. The rack and pinion multi-element is shown in Figure 86. There were several considerations made when implementing this feature in BoGL Web. The first was that, if in the future there was a need for additional, more complicated multi-elements, BoGL Web should be able to support these elements. We also wanted multi-elements to be act as several regular elements after being dragged into the screen. That is, once added to the canvas, a multi-element will be no different from a regular element aside from the fact that it added more than one element to the screen as opposed to just one element. This was the functionality of BoGL Desktop.

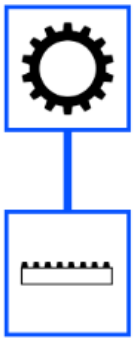


Figure 86: The rack and pinion multi-element after being dragged into the canvas.

To implement multi-elements, we started by creating a TypeScript class which inherited ElementType. Since a multi-element is just an Element with additional functionality, inheritance was logical here. Next, we added several new fields to the class. These included a list of sub-elements, a list of edges between the sub-elements, and a list of offsets. The list of sub-elements contains the ids of each element which makes up the multi-element. We next have a list of each edge in the multi-element. This is a list where each element is a list with two elements where each element is the position of one part of the edge in the list of sub-elements. Lastly, we have a list of offsets. This is a list of lists with x and y offsets which specify how far the element at that position in the sub-element list is places from the mouse when the multi-element is dragged into the screen. By implementing multi-elements this way, we are allowing for more multi-elements to be added in the future if needed.

8.2.3.10 Clear System Diagram

In BoGL Desktop, the File menu has an option “New” that clears the system diagram canvas and resets the user’s current “Save As” filename, requiring them to enter a new filename on saving. While clearing the system diagram canvas is useful functionality for the user, we did not think that the filename reset needed to be intrinsically tied to the clear functionality, since a user may want to clear the canvas without starting a new file. Thus, we re-named the “New” button to “Clear System Diagram,” moved it to the Edit menu, and added an icon for it in the icon button bar, as shown in Figure 87.

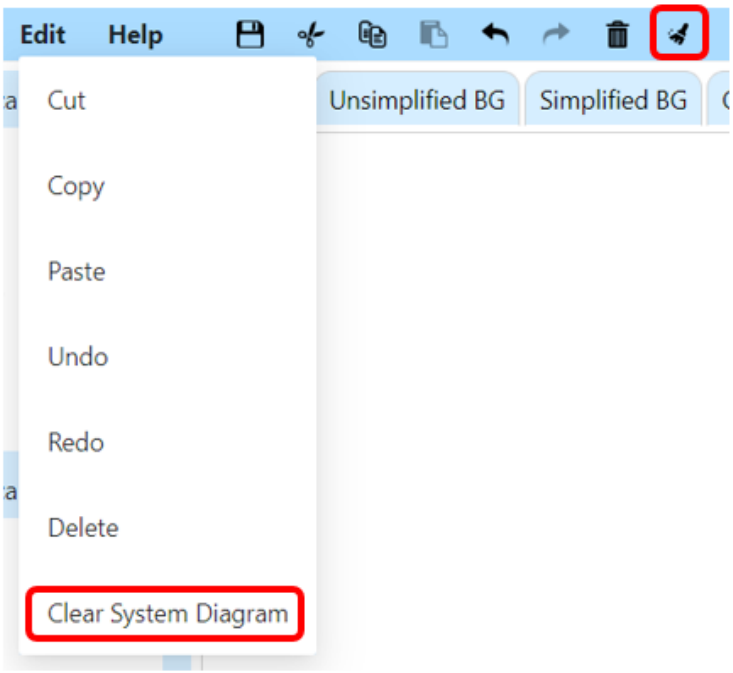


Figure 87: Edit menu and icon button array updated to add a Clear System Diagram option.

8.3 New Features

BoGL Web contains several features that were added to improve what could be done in BoGL Desktop. This section will detail the implementation of these features.

8.3.1 URL Implementation

One of the requirements of BoGL Web was that we would be able to generate URLs which represented System Diagrams. There were several things to consider when implementing this feature. This first is that URLs can only be a maximum of 2000 characters long. If the URL becomes any longer, certain browsers may not allow it to be entered into the search bar. This means that we cannot store an arbitrarily large System Diagram in a URL. To increase the size of System Diagrams that can be stored, we would need to create a compression algorithm which

would reduce the number of characters needed to store an element of edge in the System Diagram and allow for larger System Diagrams to be stored. We next needed to consider the information which had to be stored in a URL to capture all the information necessary to reproduce the System Diagram. We will use Figure 88 as an example system diagram to convert to a URL.

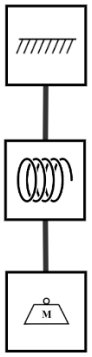


Figure 88: Simple system diagram to encode to a URL.

System Diagram elements need to store several pieces of information. The first piece of information needed is a type of ID that can be permanently tied to each object. This ID will be used to determine several things including what image should be used to represent the element in the canvas, and what modifiers the element can have. There are several pieces of additional information that must be stored in the URL, which Figure 89 shows. First are the x and y coordinates of the element. This is the position that the element is placed on the canvas at the time that the URL is generated. This position originally contains a long decimal, so to compress the URL we truncate this decimal to one place. The next piece of information is the velocity associated with an element; this is a number between zero and eight, where zero is no velocity modifier, and one through eight corresponds to different velocity directions. Next, we store

modifiers associated with the element. These modifiers are stored as integers, each of which corresponds to a single modifier. Finally, each element is assigned a unique ID to be used when determining connections between edges and elements. For each edge, there are several pieces of information that must be stored. First are the two elements that the edge is connected to. These are stored as integers which represent the unique ids for the element connected to the edge. The edge also stores a velocity which is represented the same way as velocity for the elements. In Figure 89, green represents delimiting characters that provide structure to objects, red represents keys in the object, blue represents an empty list, and yellow represents a number.


```
{
  "elements": [
    {"id": 7, "x": -191.3, "y": -48.7, "modifiers": [], "velocity": 0, "type": 0},
    {"id": 8, "x": -192.1, "y": -148.7, "modifiers": [], "velocity": 0, "type": 1},
    {"id": 9, "x": -191.3, "y": -251, "modifiers": [], "velocity": 0, "type": 3}
  ],
  "bonds": [{"source": 9, "target": 8, "velocity": 0}, {"source": 8, "target": 7, "velocity": 0}]
}
```


Figure 89: JSON object generated from Figure 88.

To further compress this URL, we took common sequences of characters and assigned them a single character to represent the sequence in the URL. A few characters representing other symbols are combined into another character abbreviation. The mapping for sequences to individual characters in order of assignment is shown in Table 16. Before mapping, the string is condensed as shown in Figure 90, with the red key labels removed. These labels are redundant because the position of a piece of information indicates what key it corresponds to as long as you maintain a set order in the element JSON objects.

{[{7,-191.3,-48.7,[],0,0},{8,-192.1,-148.7,[],0,1},{9,-191.3,-251.0,[],0,3}],[{9,8,0},{8,7,0}]}

Figure 90: Condensed object generated from Figure 89.

Table 16: Map from symbol sequences to characters for URL conversion.

Symbol Sequence	Character
{	f
[]	d
}, {	c
}],	a
}]}	e
}, {{	b
},	g
,	h
hdh	i
h[	j
]h	k
}]	l
{	m
ljm	n

Adding this mapping from a sequence to a single character significantly reduces the length of the URL, since several of these sequences are being mapped to a single character for each element. An example of this can be seen in Figure 91.

https://boglweb.github.io/?q=f7h-191.3h-48.7i0h0c8h-192.1h-148.7i0h1c9h-191.3h-251.0i0h3n9h8h0c8h7h0e_

Figure 91: URL generated from the object in Figure 90.

With this information, we can encode a URL that represents the system diagram in a way that minimizes the length of the URL while storing all the data necessary to replicate a system

diagram. This is important because URLs are only guaranteed to be supported by modern browsers while they are under two thousand characters, since a few browsers cut support shortly after this size, such as most versions of Internet Explorer. Additionally, extremely long URLs are difficult to share, and they look suspicious to users.

One future project that could complicate this URL encoding is the addition of custom labels to elements and/or modifiers in system diagrams. These labels cannot be encoded by ID, so the entire string length must be put in the URL. Viable solutions to this issue are to strip custom labels when generating a system diagram URL, only storing the diagram itself, or to store custom labels in their string form unless the resulting URL exceeds two thousand characters. For this reason, custom labels will lower the maximum size of a URL-generated system diagram, but we are unsure by how much this size will decrease. In any case where the generated URL exceeds two thousand characters, with or without labels, the application should inform the user that the diagram cannot be put in URL form and offer that they can download a file to share instead.

Now that we can generate a URL, we needed a way for the user to trigger this generation and copy the URL. To trigger the generation of a URL, users select the Generate URL button located in the File menu.

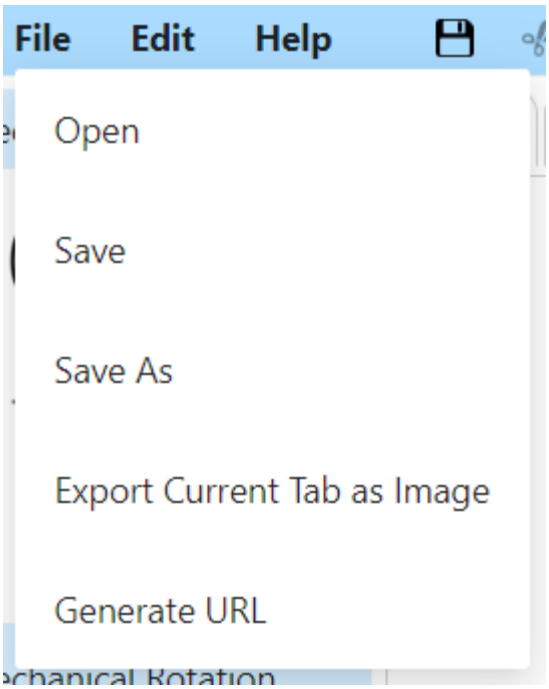


Figure 92: Generate URL button.

This button will display a modal with a button to copy the URL to the system's clipboard as shown in Figure 93.

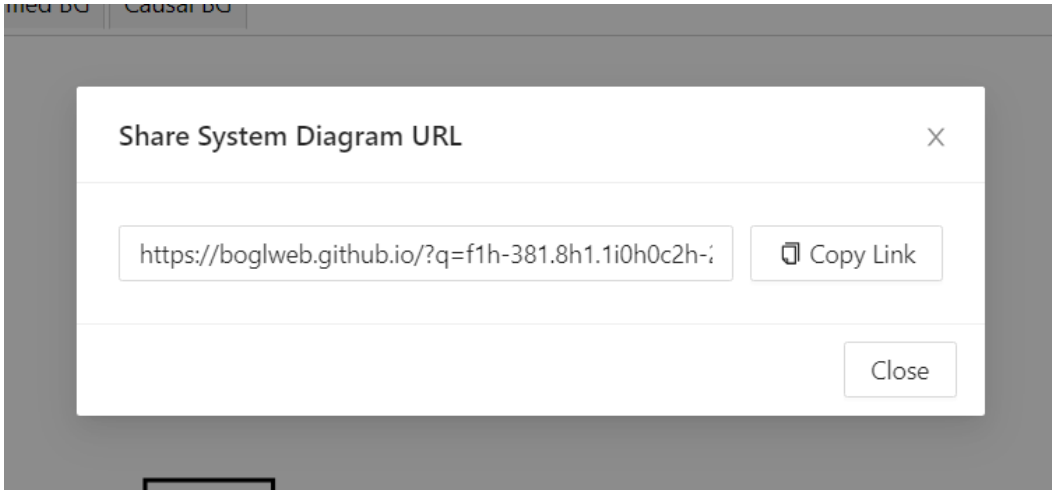


Figure 93: Modal showing a System Diagram URL with the Copy Link button.

Once the Copy Link button is pressed, a message pops up confirming to the user that the URL has been copied to the clipboard as shown in Figure 94.

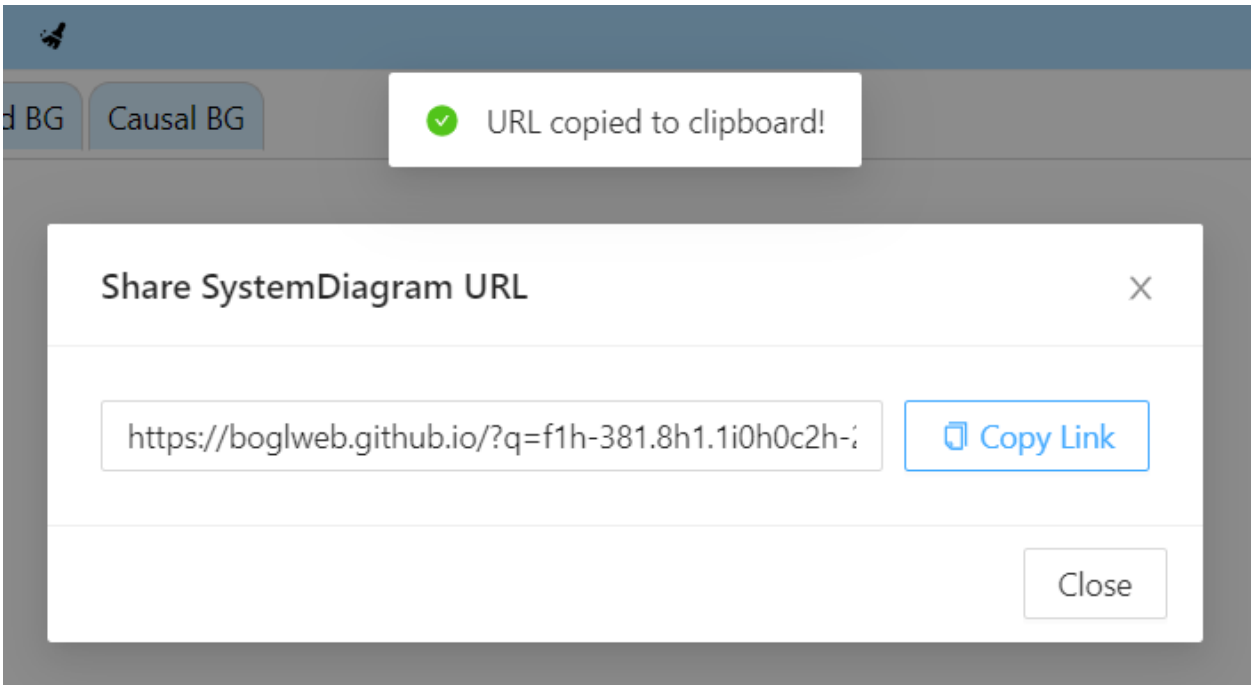


Figure 94: URL copied to clipboard message as displayed when the Copy Link button is pressed.

When testing URLs, we encountered an issue where System Diagrams loaded from the example menu and System Diagrams constructed manually would yield different element orders in the JSON string. We easily fixed this by deserializing the JSON into a C# object on the backend. We then created a JSON string from this object which ensured that the data in the string was in a standardized order.

8.3.2 State Equation UI Implementation

Originally, as described in Section 7.2.2, we expected to offer guidance on what UI would be ideal for the state equation system and to leave the implementation of the state equation

UI to the next group that works on BoGL Web. Our analysis in that section covered entering labels and values, displaying state equations, showing state variables on the bond graph, and connecting state variables on the graph to their equations. As it turned out, we had more time to work on the state equation UI at the end of the project than we anticipated, so portions of this UI were added to BoGL Web. This section details the implementation of the mockups we thought were ideal except for the mockup for entering labels and values in the system diagram tab. This mockup fell outside of the scope of what we could accomplish in the time we had left.

The ability to display state equations was the first part of the state equation UI added to BoGL Web. We adopted the mockup shown in Section 7.2.2.3 that replaces the system diagram element menu with a panel that shows state equations. Notably, we combined this mockup with the mockup from Section 7.2.2.4 that has a separate menu show all the state variables in the bond graph. When a user clicks on the causal bond graph tab, this variable menu shows up first and has a button at the top that lets the user switch to the state equation menu. In the state equation menu, the user can click a similar button to return to the state variable menu. Figure 95 shows both menus and the buttons that let the user switch between them.

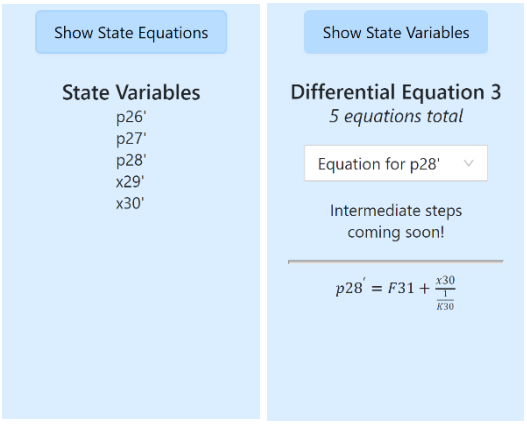


Figure 95: The state variable (left) and state equation (right) displays, respectively (cropped).

The system diagram menu in Figure 78, the state variable and state equation displays in Figure 95, and a blank menu are all stored as separate elements overlayed in the same space on the left side of the application. These menus are shown or hidden appropriately when the user changes tabs or clicks one of the state equations/variable menu buttons. A blank menu was added to replace the system diagram element menu for the unsimplified and simplified bond graph tabs since they do not display state equations and they also do not accept elements dragged into the graph area (see Figure 96). Like the graph area tabs, this system easily handles menu switches while still preserving the state of each menu. For example, expanded menus in the system diagram element menu or the selected state equation in the state equation menu will stay the same as the user switches tabs.

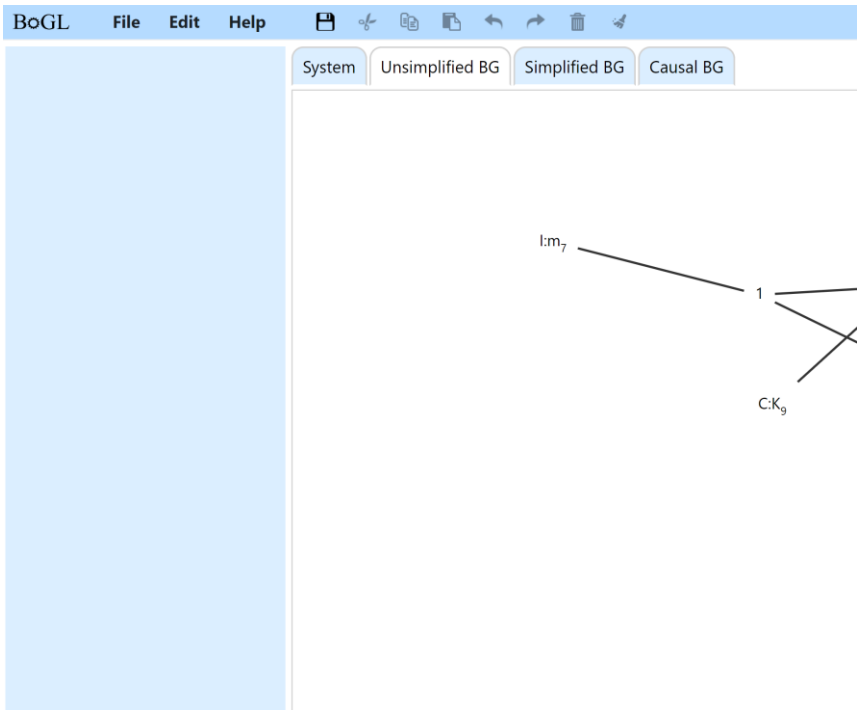


Figure 96: The blank left panel, as displayed when the user is on the Unsimplified BG and Simplified BG tabs.

As settled in the mockup in Section 7.2.2.3, the state equation menu displays intermediate equations and a final state equation for a given state variable, which Figure 95 shows. The menu currently uses LaTeX.js to format the final equation, which is a library that translates a LaTeX string into a tag structure like HTML or SVG tags. CSS files that come with this library use the tags to properly format the equation, allowing for complexities like stacked fractions and subscripts. The backend produces final state equations as LaTeX strings so that they can be rendered easily in the frontend, and it will eventually provide intermediate steps in this format as well. Currently, Figure 95 shows fake intermediate steps since that portion of the backend still requires work, but real intermediate equations can easily replace these placeholders.

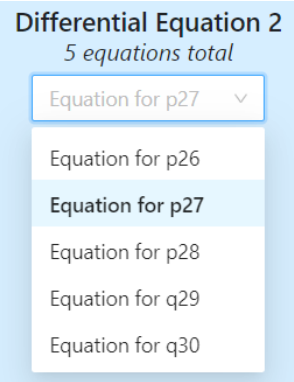


Figure 97: A dropdown in the state equation menu that lets a user pick which equation to display.

Another departure from the state equation UI mockup in Section 7.2.2.3 is that the equations are no longer accessed through Previous and Next buttons at the bottom of the menu. This system required the user to potentially cycle through several equations before finding the one they were looking for, and it gave no immediate indication of which equation the menu was currently displaying. This navigation scheme was changed to a dropdown menu, shown in Figure

97, that lists the state variable of each equation. This allows the user to jump to any equation they want and gives them more information to help distinguish the equations from each other. The dropdown is implemented using an Ant Design dropdown, the same one used to pick a causal bond graph option in the same tab.

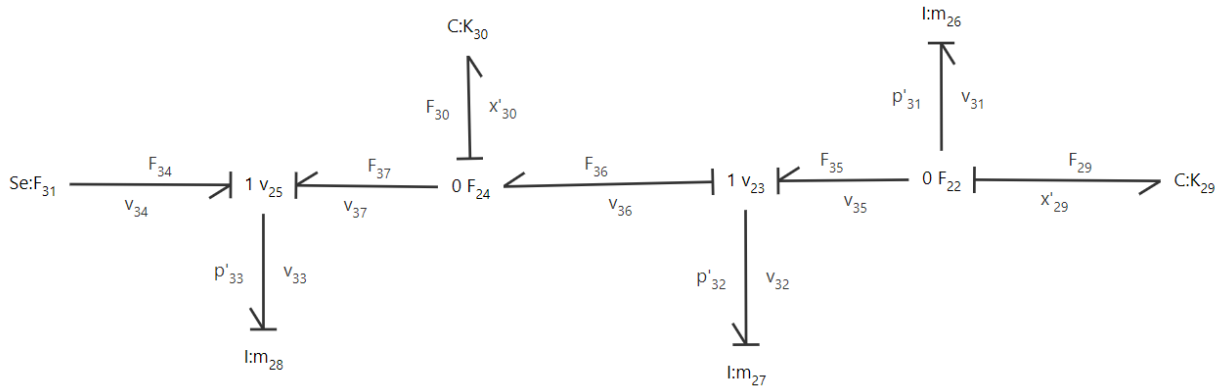


Figure 98: Bond graph with state variable labels and bond labels.

Bond labeling is the final part of the state equation UI implemented currently in BoGL Web. Each bond has an effort and flow label attached to it, as shown in Figure 98. The labels on bonds are assigned according to the domain of leaf nodes in the bond graph, which can be in the translational, rotational, or electrical domain. A bond graph can have more than one domain if the graph contains a gyrator, so the domain of given bonds and elements is determined by performing a breadth first search starting at a leaf node that does not progress past gyrators. This way, two sections of a bond graph connected by a gyrator, for example, can be assigned different domain labels. These domain labels are detailed in Table 1. I and C elements additionally change the bonds connected to them to use the labels found in Table 2.

Each label in the bond graph has a unique ID subscript that makes it easy to match the labels on the bond graph to the ones in the state equations. The effort and flow labels follow the center of their bond and keep either the front or end of each text box near the bond, depending on the slope of the bond. The text boxes are attached such that the left-most or lower text box will attach at its end and the other will attach at its front, preventing the bond labels from stretching across the bond they belong to. The effort and flow labels are also placed such that the effort label is placed to the top and left of the bond and the flow label is placed on the other side, as is conventional for bond graphs. The labels may overlap with other bonds or bond graph labels depending on the layout of the graph, but the attachment point of the bond labels ensures that the graph can at least be arranged such that all labels are readable and not overlapping.

8.3.3 Graph Drawing Implementation

Before investigating a layout algorithm, we first had to determine what aesthetic criteria a bond graph layout needed to be considered "good." We determined that the main criteria were a minimal number of edge crossings, adjacent vertices being close together, and non-adjacent vertices being far apart.

To meet these aesthetic criteria, we chose to implement a force directed algorithm to create a layout for the bond graphs we generated. We chose this algorithm because it was conceptually simple and because one teammate had implemented it before. While the layouts it created were not always the most visually appealing, they were better than just placing all vertices at random locations. We discuss the mathematical details of the force directed algorithm as well as other algorithms which may be able to improve upon the current implementation in Appendix D.

To implement the force directed algorithm, we created a class called BondGraphEmbedder. The beginning of this class includes several constants that tune the layout algorithm. These parameters can be used to change the relaxed length of an edge, repulsive force constant, attractive force constant, the maximum number of iterations the algorithm can run, and the *epsilon* value that the force must be under for the algorithm to terminate. The final values we settled on were a repulsive constant of 10000, an attractive constant of 15, a relaxed length of 100, a maximum of 500 iterations, and an epsilon of 10. We had several variables that were used to keep track of vital information between iterations of the algorithm. These included the maximum amount the force has changed, the current number of iterations, whether the positions of the vertices have been optimized, and the bond graph being laid out.

We will now discuss an example of the layout algorithm. This discussion will use the system diagram seen in Figure 99. To create a layout for the bond graph, we start with all the vertices at a random location (see Figure 100). We then check if we have reached any of our stopping criteria, reached the maximum number of iterations, or a maximum force less than the epsilon as described above. Next, for each vertex *v*, with an arbitrary order, we calculate the force acting on *v*. For our example we will compute the force on the vertex named “B _Damper R:”. We start by finding all vertices which are adjacent to *v* and put them in one set, then find all vertices which are not adjacent to *v* and put them in another set. In our example, the adjacent vertex is named “1” and the non-adjacent vertices are “M _Mass I:” and “k _Spring C:”. We can then compute the attractive force between *v* and all elements adjacent to *v*, and the repulsive force between *v* and all elements not adjacent to *v*. We included Figure 101 to help visualize the direction of these forces. In this figure, orange arrows are repulsive forces, and green arrows are attractive forces.

After computing all these individual forces, we sum them up. This sum creates a vector which we can use to determine how this element should move to improve the layout of the graph. We then move *v* based on this vector and recompute these forces again in the next iteration (see Figure 102). In each iteration we also find the vector *x*, with the maximum magnitude and record it. At the start of the next iteration, we check if the magnitude of *x* is below the epsilon we chose as a stopping criterion. We also check to see if we have reached the maximum number of iterations. Once we reach a stopping point, we have a final layout for the bond graph (see Figure 103).

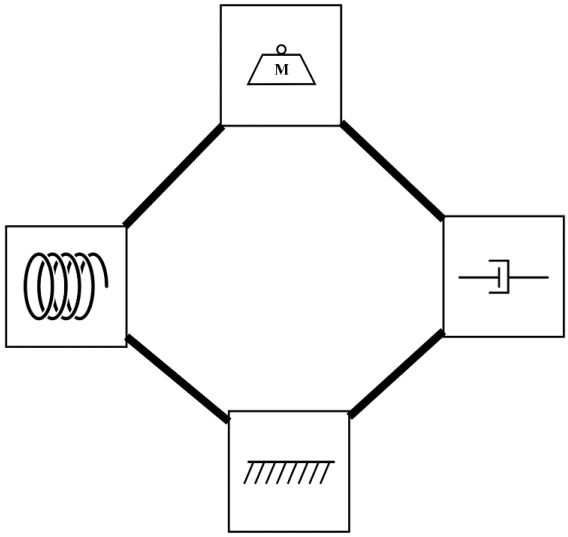


Figure 99: Example system diagram for graph layout algorithm.

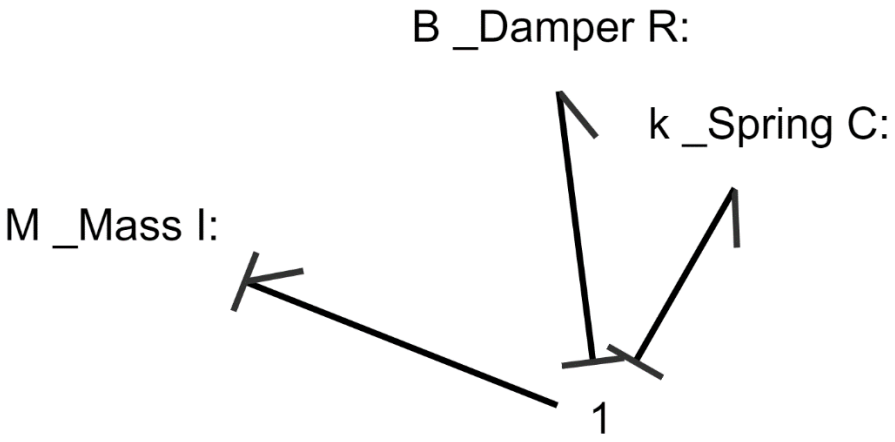


Figure 100: Bond Graph with random vertex locations.

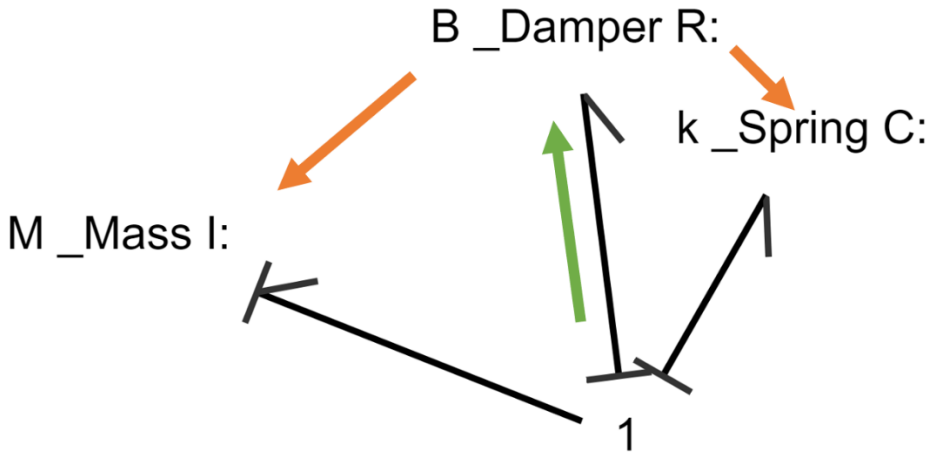


Figure 101: Force directions for random vertex locations.

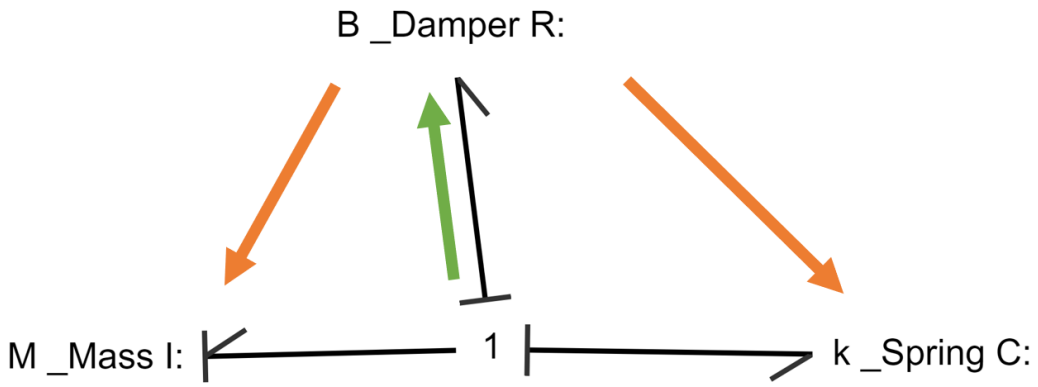


Figure 102: Force directions for intermediate vertex locations.

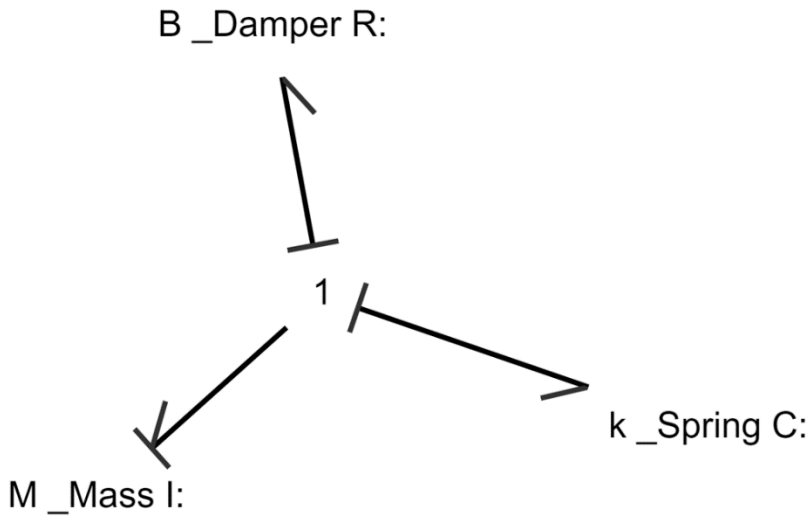


Figure 103: Final bond graph layout.

This algorithm can create reasonable layouts for the bond graphs generated in BoGL Web. To create some example layouts, we started with the system diagrams in Figure 104, Figure 105, and Figure 106. We used BoGL Web to generate layouts for these system diagrams that can be seen in Figure 107, Figure 108, and Figure 109. It is important to note that there was

no panning, zooming, or moving of elements done before capturing these images, The location of all vertices as well as the pan and zoom levels were all computed by the algorithms.

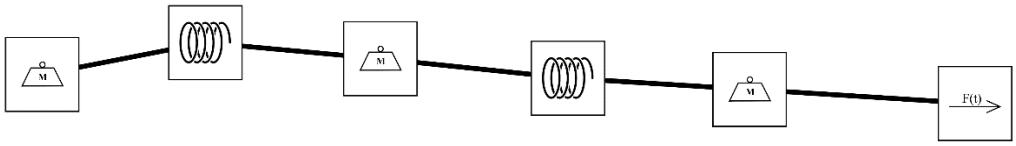


Figure 104: Example system diagram one.

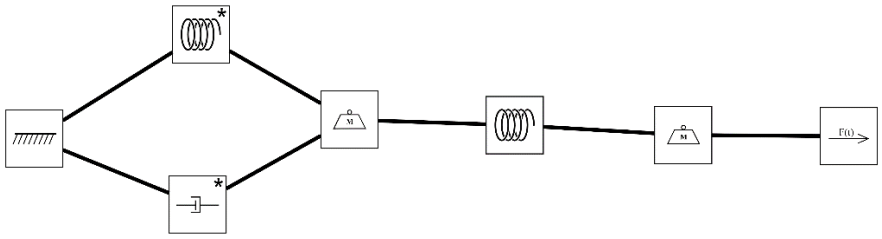


Figure 105: Example system diagram two.

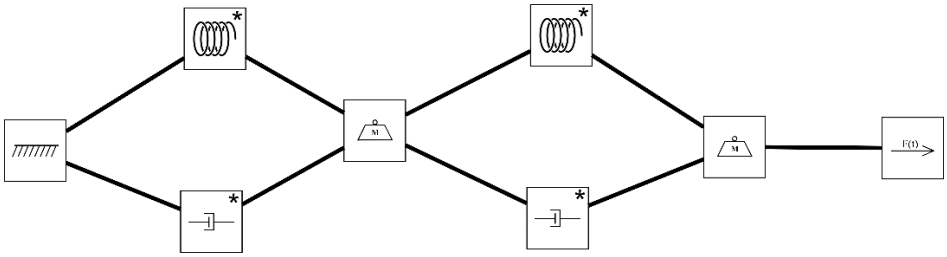


Figure 106: Example System Diagram three.

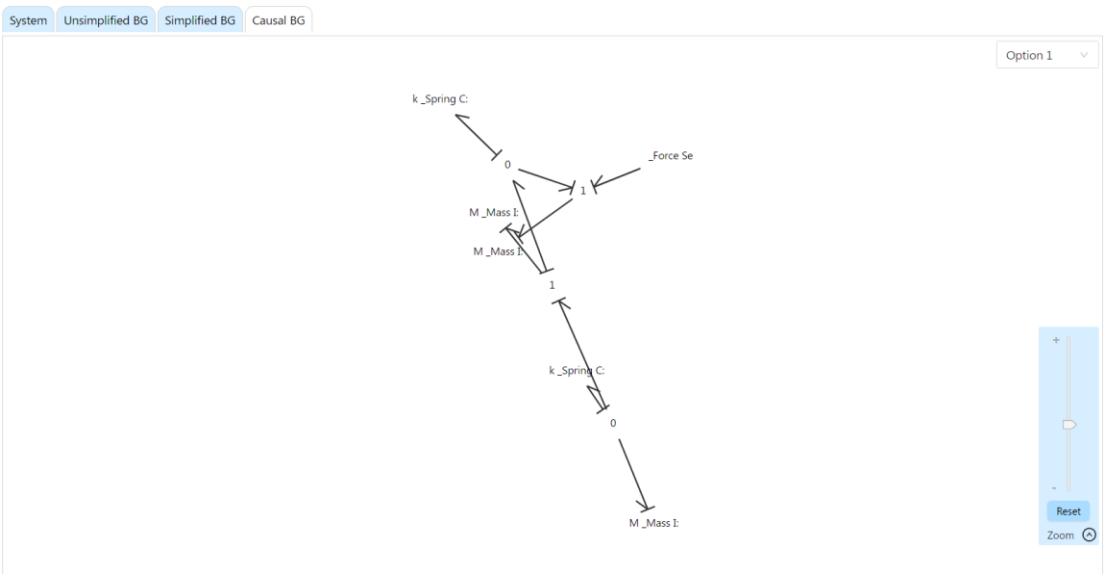


Figure 107: Force directed algorithm output for example one.

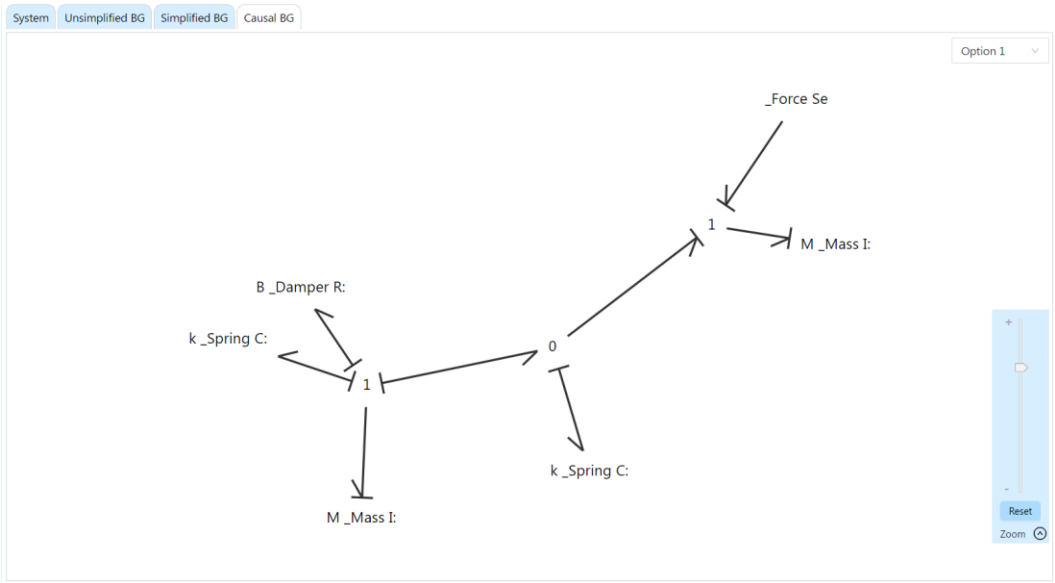


Figure 108: Force directed algorithm output for example two.

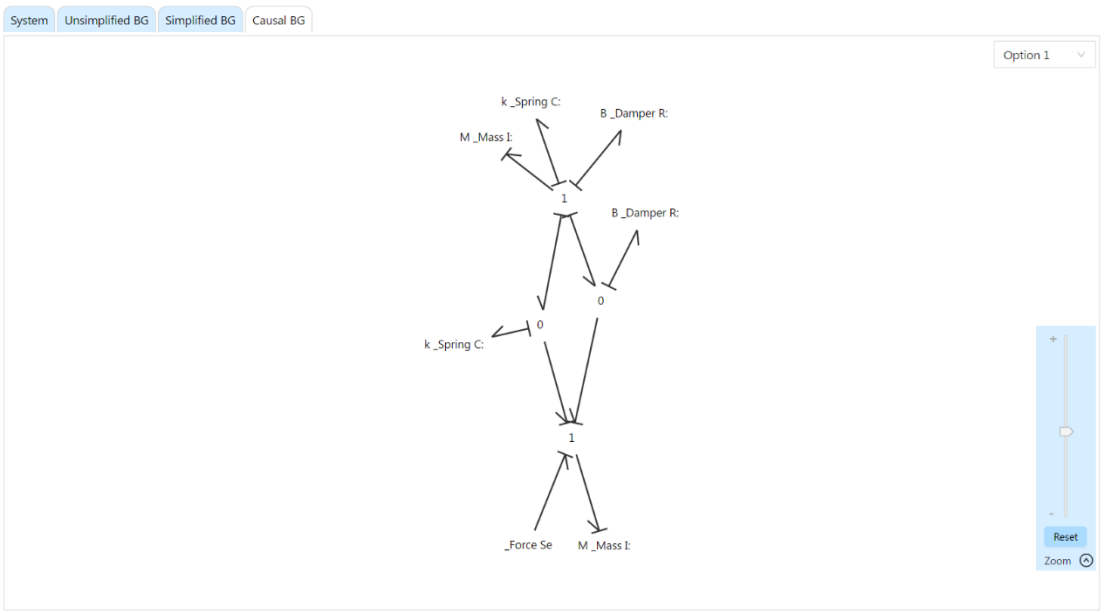


Figure 109: Force directed algorithm output for example three.

To evaluate the effectiveness of the layout algorithm we compared the layouts created in BoGL Web to those created in BoGL Desktop (see Figure 110, Figure 111, and Figure 112). From these comparisons we can see several advantages to the layout algorithm used in BoGL Web. The first is that all elements of the bond graph are always on the screen. In BoGL Desktop, several elements of the bond graph are placed off the screen in the layout. This is confusing to the user since parts of the bond graph that they would expect to see are not immediately visible. The second major issue is that elements of the bond graph are often placed at the exact same point, making it exceedingly difficult to distinguish overlapping elements. This is also an issue with the layout algorithm used in BoGL Web, but the BoGL Web algorithm does not place elements directly on top of each other and when there is an overlap between elements, it is much less significant than in BoGL Desktop. Part of this, however, is due to the lack of boxes around element names in BoGL Web, so this point was not the deciding factor for choosing this layout algorithm.

The main advantage of the BoGL Desktop algorithm over the BoGL Web algorithm is processing time. BoGL Desktop generates a layout for the bond graph in a few milliseconds whereas BoGL Web takes several seconds. This processing time is not a huge issue when there are a small number of graphs, but for system diagrams which generate many options for causal bond graphs, the user can be waiting for over a minute for the system to finish its processing.

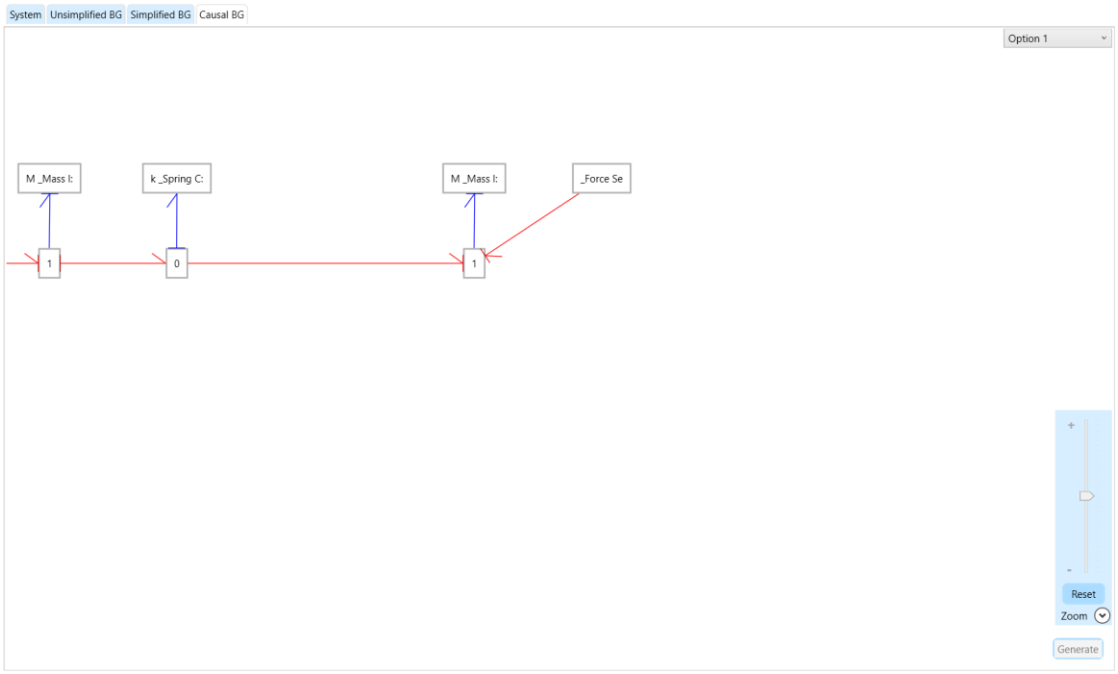


Figure 110: BoGL Desktop layout for example one.

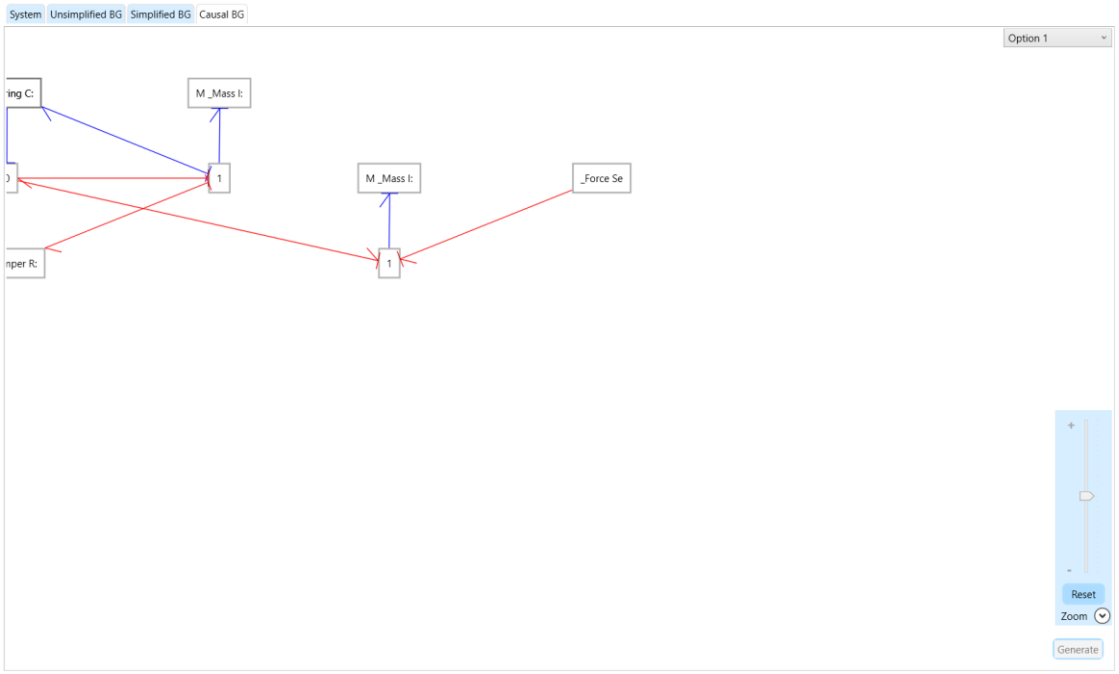


Figure 111: BoGL Desktop layout for example two.

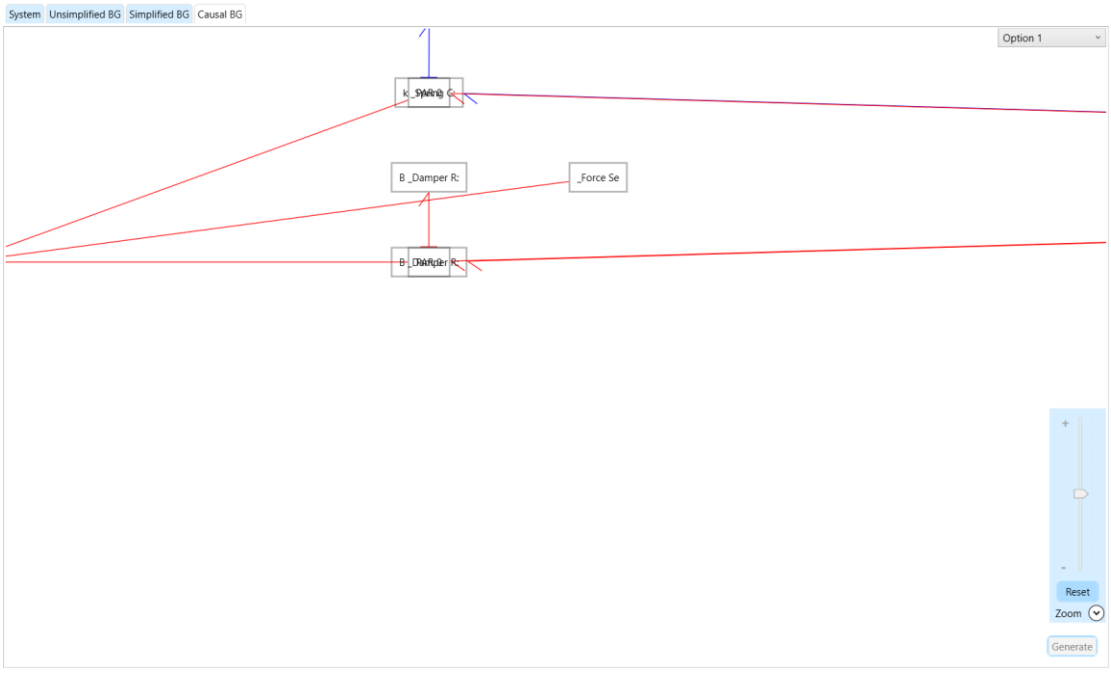


Figure 112: BoGL Desktop layout for example three.

One other issue which was not captured in the comparison with BoGL Desktop is that the new algorithm does not always find the layout that best fits our aesthetic criteria (such as the good layout in Figure 109). This is because the algorithm can get "stuck" in a solution that appears locally optimal but is not globally optimal. This means that there may be a better layout which could be constructed, but the algorithm is not able to find it because, to the algorithm, it appears that it has found the best solution.

In general, this algorithm works well enough to be a first step in laying out bond graphs aesthetically in BoGL Web.

8.3.4 Tutorial Implementation

The tutorial for BoGL Desktop consisted of a set of "How To" videos which walked the user through various aspects of the software. For BoGL Web, we instead opted for an interactive walkthrough of the website. This would allow for users to interact with the tutorial which is still on the website instead of having to go to another webpage. Additionally, users would be seeing the actual website when going through the tutorial instead of just watching a video.

To implement this feature, we tried several libraries including IntroJS, Blazor.IntroJS, and PSC.Blazor.Componets.Tours. Unfortunately, the only tutorial library we were able to get working was IntroJS. The other libraries all appeared to either be out of date, or their documentation was not thorough enough to get set up. Fortunately, IntroJS performed all the necessary functions we needed for our tutorial.

To use IntroJS we first needed to include the library in our project. This was not simple as we were not using a standard dependency manager for JavaScript libraries. To include the library, we ended up finding the URL of the library and including a link to it in the HTML.

To trigger the tutorial, we created a TypeScript function which contained each step of the tutorial. A step contained an element selector and a message. The element selector was used to determine what element of the user interface would be selected when a given part of the tutorial was being shown and this function was called through JSInterop.

Overall, the tutorial is functioning well, however, there are a few issues which should be investigated in the future. The main issue is that the tutorial does not react to different screen sizes well. For smaller screens, parts of the tutorial will appear off screen, preventing the user from being able to select the next button. A second issue is that the gifs in the tutorial are small. A future project should look at a way to make these images larger.

8.4 Deployment

When initially looking for platforms to deploy BoGL Web to, our team investigated several options. These included Microsoft Azure, GitHub Pages, Amazon Web Services, a WPI Server, and a few other cloud hosting services. Initially, due to the abundance of tutorials, and a student license, allowing us to test our deployment, we settled on using Microsoft Azure. Specifically, we would be using Azure Static Web Pages. At a high level, this service provides a location for our users to download our website. Since we chose Blazor WebAssembly, we did not need a server to host a backend, so this was all we would need to run the website.

Getting the website deployed to Azure was not the easiest task. After creating a Static Web App using the Azure user interface in Figure 113, we attempted to deploy our website using GitHub Actions. GitHub Actions are automated processes that are run whenever code is committed to a specified branch. We started by investigating this option for deployment since this style of automation is an industry standard technique and will not require all developers to

learn how to deploy the website using another method. Additionally, it makes sure that the latest version of our main branch in GitHub is always deployed to our user facing website. Unfortunately, this was quite difficult to set up.

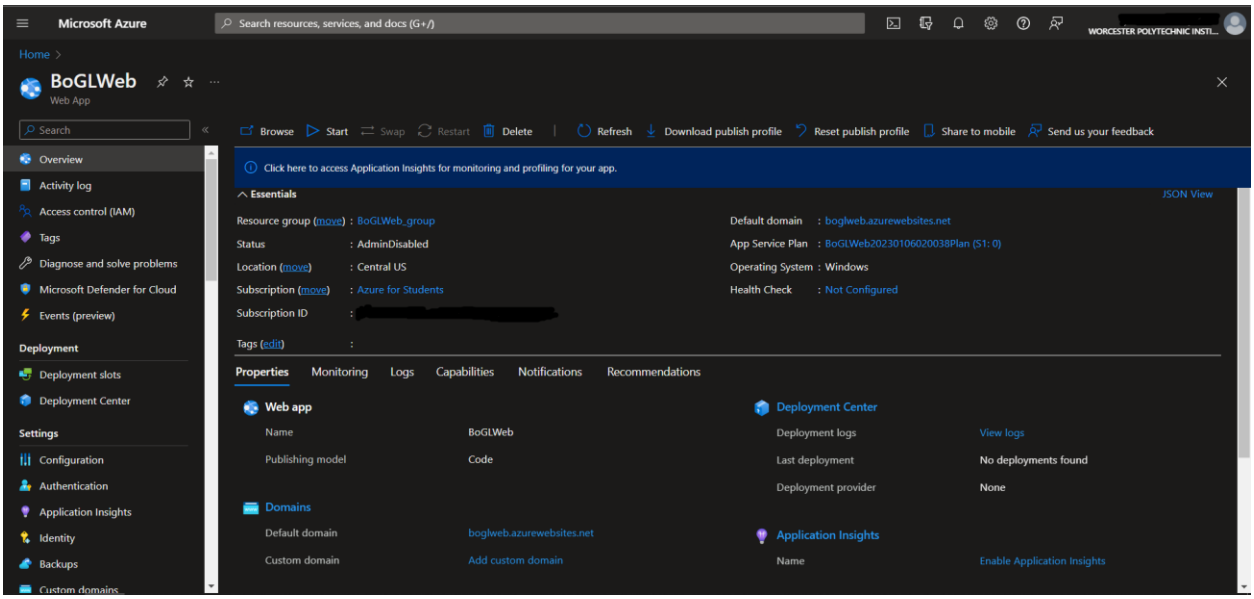


Figure 113*: Azure user interface.*

We ran into many issues when first attempting to use this technique. Because of this we, pivoted to deploying using the Microsoft Visual Studio publish tool pictured in Figure 114. Using this tool, we were able to easily deploy our website to Azure, however, we then discovered several other issues. The first was an issue with D3.js. When deployed to Azure, our website was secured using HTTPS. A requirement of this is that any dependencies we downloaded would also use HTTPS. At the time of our first deployment, we had not been using a version of D3.js which was secured by HTTPS. This was a simple fix, and just involved changing the link to this dependency. The next issue that we encountered was that no rules, rulesets, or example files were being loaded. We quickly discovered that this issue was caused

by the server not assigning routes to files on the server aside from the main website. This meant that we were unable to reach the rules, rulesets, or examples with a URL. We spent a lot of time investigating this problem with no successful solutions. Instead, we pivoted to using GitHub Pages to host rules, rulesets, and examples. This turned out to be a successful solution and was simple to implement.

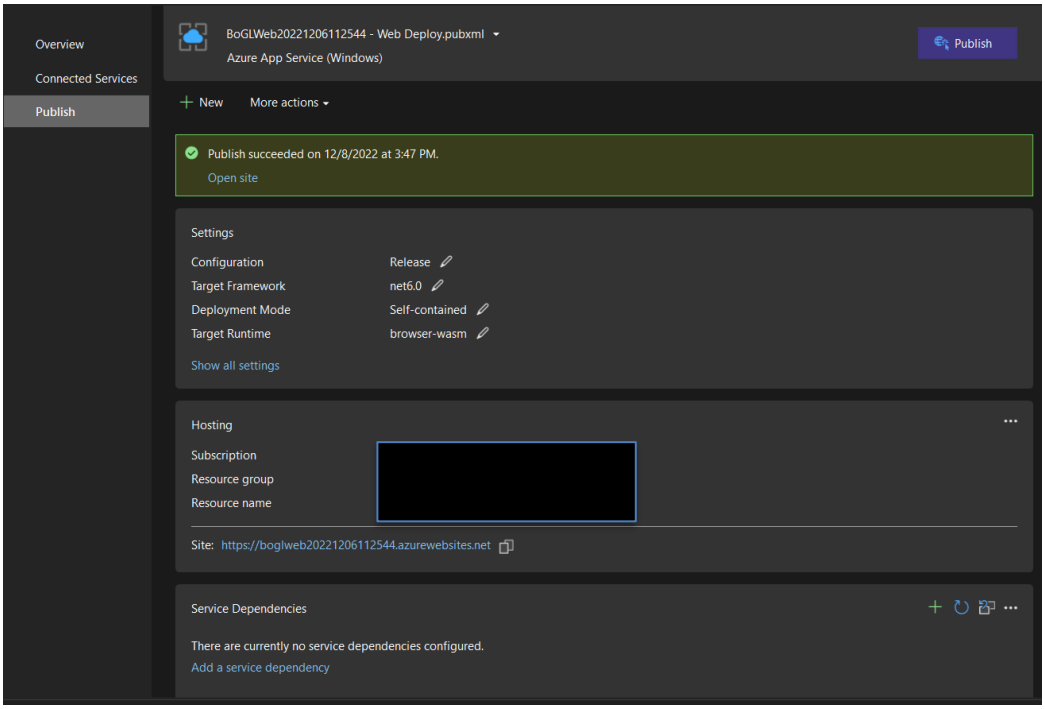


Figure 114: Visual Studio publish interface.

Now that we had the website working with manual deployment, we went back to working on setting up automatic deployment through GitHub Pages. The issue that we experienced on the first attempted turned out to be related to missing authentication keys to access the repository and the Azure account which hosted the website. Once these keys were added we successfully set up automated deployment.

Unfortunately, we soon realized that Azure may not be a sustainable solution for hosting the website. When we had originally chosen to use Azure, we used their price calculator to estimate the cost to run the website each month. The original estimate was that the website would be free to host, which was one of the main factors for choosing Azure. Now that we had a website deployed to Azure, the charges did not match this estimate. The assumption is that this was due to the website being larger than the limit for a free website, but we were unable to confirm this. At this point we began to look for other hosting options.

We chose to try and deploy to GitHub Pages next. We chose this platform as an option due to our team having some experience with the technology, and hosting a website being free. This is something that we had been able to confirm through the website we used to host our rules, rulesets, and examples, and from prior experience. Before attempting to deploy to this platform we did some research and discovered an article discussing the process (Swimberghe, 2020). Although there was limited evidence of Blazor running on GitHub Pages, the article we found made the process seem simple, so we found this route promising.

To deploy to GitHub Pages, we followed a tutorial titled “How to deploy ASP.NET Blazor WebAssembly to GitHub Pages” (Swimberghe, 2020). The first step to deploying the website was setting up a GitHub Action (see Section 6.6.1.5) to build the website. The first step was to build the website. This was done by calling an action which can build .NET code. We then ran a command which took the built project and published the files necessary. We then took these published files and committed them to a branch which was set up to automatically deploy to GitHub Pages. Finally, we created a .nojekyll files with was used to tell the website to host files with an _ in their names. We have now successfully deployed the website.

GitHub Pages is how the website is currently deployed. This solution is working well, and we have not found any issues with GitHub Pages as the time of publishing this report.

9 Testing

In this section, we discuss the process used for testing various aspects of the website. These include UI Testing and Backend Testing.

9.1 UI Testing

UI testing for BoGL Web consisted of three main steps: testing features individually, testing overall browser compatibility, and testing specific functionality in compatible browsers. The first step of testing was simple and just consisted of manual testing features in several situations as they were programmed. While this testing was not automated, it captured most errors before the feature was merged into a stable version of the application.

The second method of testing the UI was more systematic. One of BoGL Web's major goals is to allow students on any operating system (OS), using any common browser, to access the application, as the requirement of using a Windows OS made BoGL Desktop inaccessible to many students. Thus, a major focus of our testing was to ensure that BoGL Web could be used in all major browsers and OSs. To do this, we used a paid tool called BrowserStack that lets you test websites on macOS and Windows with seven browsers. These were Chrome, Firefox, Opera, Edge, Safari, Yandex, and Internet Explorer. The tool also lets you visit websites using different versions of a browser.

To determine BoGL Web's compatibility with different browsers, we opened the site on the most recent version of each browser offered by BrowserStack in each version of Windows and macOS supported by BrowserStack. This simultaneously tests BoGL Web's compatibility with different browsers and different versions of OSs. Our goal was to check whether all modern

OSs and browsers support the application. Table 17 shows the results of this testing, where red indicates incompatibility, yellow indicates compatibility issues on an initial load of the page that disappear on subsequent loads, green indicates compatibility, and grey indicates that the browser is not supported by a given OS. We considered a browser to be compatible if it was able to load the website without obvious errors in appearance or functionality.

Table 17: BoGL Web OS and browser compatibility

OS		Browser						
		Firefox	Opera	Chrome	Edge	Safari	Yandex	Internet Explorer
Windows	Windows 11							
	Windows 10							
	Windows 8.1							
	Windows 8							
	Windows 7							
	Windows XP							
macOS	Ventura							
	Monterey							
	Big Sur							
	Catalina							
	Mojave							
	High Sierra							
	Sierra							
	El Capitan							
	Yosemite							
	Mavericks							
	Mountain Lion							
	Lion							
	Snow Leopard							

Testing with BrowserStack revealed promising results (see Figure 115). Table 17 shows that Firefox, Opera, Chrome, Edge, and Safari on macOS all support BoGL Web, with backwards compatibility to several older versions of Windows and macOS. W3Counter's browser statistics (*W3Counter: Global Web Stats*, n.d.) show that these five browsers made up 94.3% of the marker share in February 2023, and W3Schools' browser statistics (*Browser Statistics*, n.d.) concur, claiming that 99.2% of visitors to W3Schools in February 2023 used one of these five. The two browsers that do not support BoGL Web at all are Internet Explorer and Yandex. Yandex is a small Russian-based browser ("Yandex," 2023) and Internet Explorer has been discontinued (v-hearya, n.d.), so we were not particularly concerned by BoGL Web lacking compatibility with these two browsers. Additionally, Table 17 shows that Safari for Windows does not support BoGL Web. This is because Windows only supports an old version of Safari and further updates to the Windows version have been discontinued (Abdul, 2022), so we also did not consider this to be an issue.

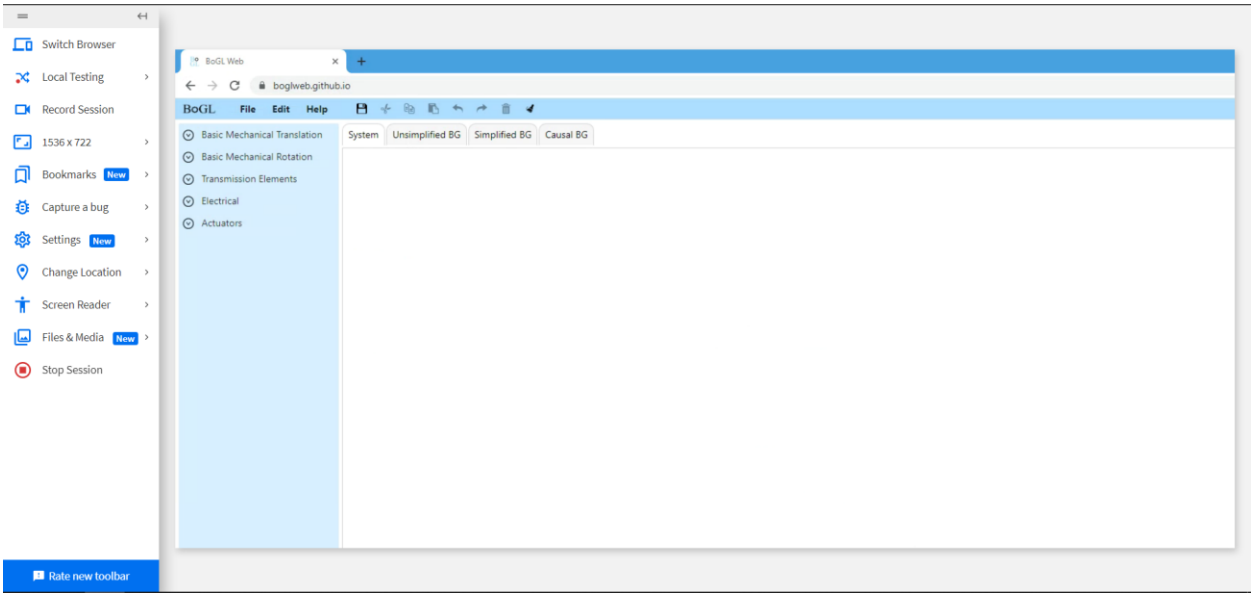


Figure 115: Example of BrowserStack being used with the latest version of Google Chrome.

BoGL Web compatibility mainly follows each browser's adoption of ECMAScript 2015, also known as ES6, which was the second major revision to JavaScript (*JavaScript ES6*, n.d.). This update added several major features to JavaScript, to the extent that most JavaScript programmers either prefer to use ES6 or use a polyfilling library to replicate most of its features in ES5, the previous version of JavaScript. Internet Explorer, which is now retired, was the last major browser that did not support ES6, thus we decided not to use polyfilled JavaScript in BoGL Web, since it offered little benefit in terms of compatibility. We also discovered during development that some browser-specific differences in support for CSS rules and JavaScript APIs required us to change the way we wrote BoGL Web functionality. Unlike the issue with supporting ES5, these changes were easy to make and have been added to the application without any issues.

According to NetMarketShare, the most commonly used Windows OS version is Windows 10, and the most common macOS version is macOS Catalina (*Operating System Market Share*, n.d.). Given this, we decided to do a more thorough test on Windows 10 and macOS Catalina with Edge, Firefox, Chrome, and Opera, plus Safari on macOS. The testing followed these steps:

1. Open the Mechanical Translation section of the element menu.
2. Drag a mass into the canvas.
3. Copy-paste the mass twice (so that there are three total), then delete one.
4. Undo, then redo this deletion.
5. Make an edge between the two remaining masses.
6. Add the friction modifier to one of the masses.
7. Add any velocity to the bond and move it around.
8. Generate a bond graph.

9. Export the system diagram and bond graph as PNGs and SVGs and check that all four are correct.
10. Save the system diagram to a file.
11. Delete all elements on the canvas then open the generated file.
12. Generate and open a URL, then confirm that the resulting system diagram is correct.
13. Open the “Basic Two Mass System – 1” example and confirm that it is correct.

We performed this sequence of tests on the nine browsers listed above and discovered two key issues. First, all the macOS browsers had a bug where velocity arrows did not render correctly in exported PNGs. The other issues were with Safari, consisting of an issue with the modifier menu being visible and opening the file explorer. None of the Windows browsers had issues and the macOS browsers, excluding Safari, only had the image export issue and nothing else. After testing, we fixed these issues and retested each step that failed on a given browser until it passed.

9.2 Backend Testing

We required backend testing for some of the features of this project. Our testing procedures can be split into two categories, automated and manual. These are described in detail in the next two sections. Through these methods, we were able to ensure that certain algorithms in the backend were working properly without relying on the primary user interface.

9.2.1 Automated Testing

We verified some of the backend features using Playwright (*Fast and Reliable End-to-End Testing for Modern Web Apps | Playwright*, n.d.). Playwright is an automated testing

platform made by Microsoft that supports .Net, C# and TypeScript. Having an automated system allowed our team to test and retest a sequence of many user inputs without having to individually track the system state at each step. We used this to test our undo/redo system. We needed a method of testing its underlying stack object before implementing the CanvasChange system that would change the system diagrams and bond graphs. We did this testing by using Playwrights ability to set expectations of the current state to check the size of the stack and location of the pointer after each button press.

For each test, we created two separate files. One is a Razor file, in which we provided a layout for the UI page that is used during testing. Any tests made through .Net can be activated by running a command shell prompt from the codebase. The program will then run the application and open a new page not otherwise accessible from the main interface. .Net will design this page based on the UI provided in the Razor file. For the testing procedure, the program will run all designated testing methods in a second C# file.

For the undo/redo testing algorithm, the .Net procedure creates a new page with four buttons and two text fields. The buttons control the “prev,” “next,” “clear,” and “edit” functionalities for the stack. In the regular BoGL Web interface, these would correspond to the “redo” button, “undo” button, “clear canvas” button, and “edit” button. We also store a global variable *hashListElement* for adding values to the stack. Each call to “edit” adds the current value of *hashListElement* to the stack and increments the value by 1. This ensures that each new value added to the stack is unique, which streamlines the testing process. For this test, our C# file steps the EditionList through the following procedure and checkpoints:

1. Initialize an EditionList object.
2. Check that *size* is set to 0 and *index* to -1.

3. Add 10 items to the *EditionList* by clicking the "edit" button 10 times. Every time an item is added, the program checks that *size* and *index* have both increased by 1.
4. Press the 'undo' button 7 times. Every time the button is clicked, the program checks that *size* does not change, but *index* decreases by 1.
5. Press the 'redo' button 5 times. Every time the button is clicked, the program checks that *size* does not change, but *index* increases by 1.
6. Press the 'edit' button once. The program checks that the remaining values in the stack – of which there should be 2 – are overwritten with a new value at the location of the pointer.
7. Press the 'clear' button once. The program checks that the *size* and *index* values have reverted to their defaults of 0 and -1.
8. Press the 'edit' button once, then 'undo' once, then 'redo' once. The program checks that the 'edit,' 'undo,' and 'redo' buttons have functioned the same as they did previously.
9. Press the 'edit' button 4 times.
10. Press the 'redo' button 5 times. As 'undo' has not been pressed since the last time 'edit' was called, the pointer should be at the end of the stack. The program checks that *size* and *index* do not change.
11. Press the 'undo' button 13 times. After 5 calls, the *index* should decrease to 0, and the pointer should move to the beginning of the list. For the subsequent 8 calls, the pointer and *index* should not move. After all presses, the program checks that the *index* is 0 and the pointer is at the beginning of the list.

9.2.2 Manual Testing

During development, some of the features we included in the backend did not have some form of display in the frontend. For example, one of the last features we implemented was a method of generating state equations. Instead of using a rough version of the state equation interface in the app, we tested generation using console outputs. Whenever the ‘generate’ button was clicked in the interface, the backend automatically formulates the differential state equations from the generated bond graph and stores those equations in the backend. To check that these equations were correct, we temporarily added a separate button to the canvas that would print the string values to console.

10 User Evaluations

In this section, we discuss the survey conducted in the ME 4322 Modeling and Analysis of Mechatronic Systems course during the last week of C Term 2023. In our study, we prompted students to perform tasks in BoGL Web and answer questions about their experience.

10.1 User Evaluation Design

We started by creating a list of requirements for our survey. This would help in the creation of our survey questions and ensure that the user study would allow us to collect all the data we needed to evaluate the effectiveness of BoGL Web.

10.1.1 Goals and Requirements

This survey was designed to collect user feedback about the usability of BoGL Web. We aimed to provide the students with a general overview of the survey, including their rights as a participant. We wanted the user to:

- Test the tutorial and evaluate how easy to find and effective it was.
- Have the user test distinct functions for constructing system diagrams within the BoGL Web interface.
- Provide images, URLs, or downloadable files of the system diagrams or bond graphs they used for testing.

One of the challenges in designing a survey was determining how to give students credit for their responses. We wanted our team to receive feedback anonymously while still providing Professor Radhakrishnan, the course professor, with the names of everyone who helped complete

it. In the end, we decided to have Professor Radhakrishnan record names by having students email him separately with the images of bond graphs and system diagrams. We also experienced another challenge. We originally planned to have the students complete a homework assignment using the application. However, for final implementation, we decided to have the students only work on a mechanical system we describe and then complete the general feedback portion of the survey. This process is described in detail in the next section.

10.1.2 Creating the Survey

When creating the survey, we first added a preamble, one we modified from a template provided by Worcester Polytechnic Institute. In this preamble, we started by providing the students with a general overview of the survey. We told the students that their feedback will be used to improve future iterations of the application. We also notified them that Professor Radhakrishnan would be receiving their names but that we, the developers, would not.

The second part of our survey tested BoGL Web's usability by prompting the user to perform a series of tasks, enter their results, and then rate the quality of their experience. They were first asked to find the tutorial. If the user gave, they were provided with the location of the tutorial. Either way, they are asked to complete the tutorial. The user is then asked to describe how easy it would be to use BoGL Web after following the tutorial. After that, we provided an image of a mechanical network in Figure 116 and asked students to create a system diagram for it in the application. We then asked them to provide how long it took them to make the system diagram, how easy it was to make the system diagram, and how easy it would be to construct the system diagram a second time. We then had the student construct a new copy of the system diagram and provide the same information.

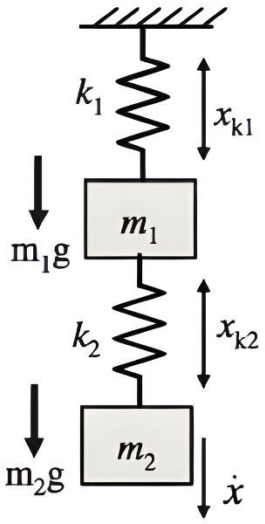


Figure 116: The mechanical network we provided in the survey for student users to construct in BoGL Web.

The last part of our survey asked the students for general information about their app experience. In the survey, we asked about how easy it was to construct a system diagram, generate a bond graph, and edit object modifiers. We then included fields where students could identify bugs they discovered, and list features they wanted future iterations of the application to include. We finished by asking that the students rate their satisfaction with all operable features of the application that they tested. A full copy of the survey can be found in Appendix A – ME/RBE 4322 Survey, and the results can be found in Appendix B – Survey Data.

10.1.3 Implementing the Survey

We administered this survey to students in ME/RBE 4322 Modeling and Analysis of Mechatronic Systems during the last week of C-Term 2023. Before delivery, we gave a short demonstration of BoGL Web features to the class, lasting about 10 minutes. We then released the survey to the students and gave them the rest of the class period to complete it. While the

students worked on the survey, our team proctored and answered questions about the interface, the features, and how to proceed when a bug was encountered in the application. We analyze the results of the survey in the next section.

10.2 Analysis of User Feedback

In this section, we will discuss the data that was collected during the user study. We begin with a general overview of the data collected. We will then look at the quantitative data and discuss conclusions that can be drawn. We then move on to the qualitative data that was collected and discuss the importance of the comments we received. Finally, we will discuss some key takeaways.

10.2.1 Discussion of the Data

We collected data from thirty-five students primarily through multiple-choice and short answer responses. We asked each test user to rank their comfort level with a particular task, we asked the user to provide their response as an answer to a multiple-choice question. We asked users to rate their satisfaction with individual BoGL Web features as “extremely dissatisfied,” “somewhat dissatisfied,” “neither satisfied nor dissatisfied,” “somewhat satisfied,” or “extremely satisfied.” To better compare these scores, we assigned each ranking a scale from one to five. We record all responses to these kinds of questions as raw text values. The survey also asked the users some questions about how easily they would be able to replicate some of the tasks they completed.

10.2.2 Quantitative Data Analysis

We now provide an overview of the quantitative data from our survey. To gauge general user satisfaction with the application, we looked at average satisfaction scores awarded by user testers for individual features. We then gathered data to summarize overall user satisfaction with the application. We go on to discuss the feature improvements users wanted to see in future iterations of the application. We also provide a list of the bugs users found during testing.

10.2.2.1 System Diagram Statistics

We accumulated general statistics on how well users were able to construct system diagrams in the application. On average, each student spent 4.38 minutes (or 4 minutes and 23 seconds) making a system diagram and generating the respective bond graph. Since bond graph generation occurs with only the click of a button, we assume most of this time is spent making the system diagram. Eighty-one percent of users said they would be able to easily replicate any system diagram they were asked to construct. We also asked each user to rate the difficulty of system diagram creation as “very easy,” “somewhat easy,” “neither easy nor difficult,” “somewhat difficult,” or “difficult.” To best analyze these ratings, we relabeled them as 1, 2, 3, 4, and 5 respectively and found the average of all such rankings. Overall, the average difficulty rating was 1.47, indicating that user experience with system diagram creation was split between “easy” and “very easy.”

10.2.2.2 User Satisfaction with Features

We asked users to rate their satisfaction with individual BoGL Web features as “extremely dissatisfied,” “somewhat dissatisfied,” “neither satisfied nor dissatisfied,” “somewhat satisfied,” or “extremely satisfied”. To better compare these scores, we assigned each ranking a

scale from one to five. To determine the overall satisfaction score for each feature, we took the average of every score for that feature. The overall scores we obtained as averages over all thirty-five students are as follows:

- 4.88 for URL generation.
- 4.66 for generating a bond graph.
- 4.63 for using the tutorial.
- 4.53 for adding modifiers.
- 4.47 for loading example files.
- 4.47 for responsiveness.
- 4.44 for saving a system diagram to a .bogl file.
- 4.41 for opening a .bogl file.
- 4.38 for creating a system diagram.

We compare these scores in Figure 117:

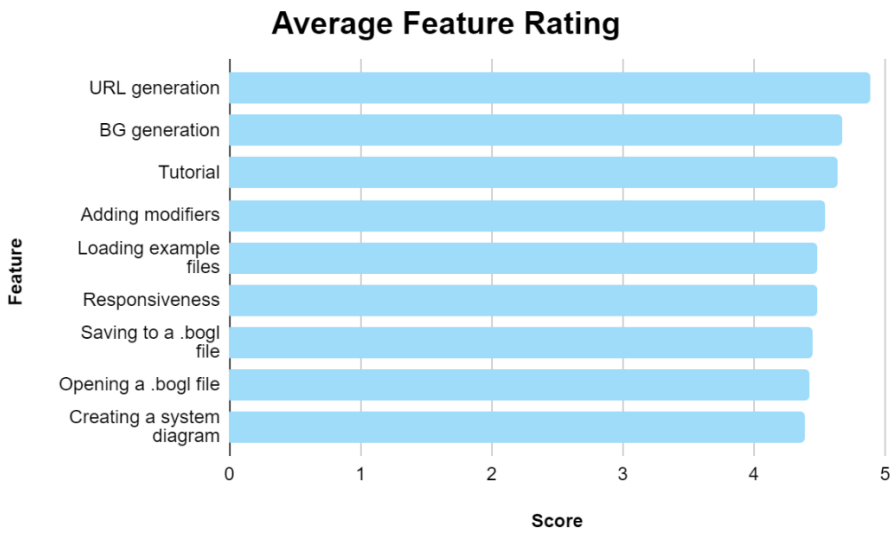


Figure 117: The average satisfaction rating for all features listed above.

We found that 91% of users were satisfied with saving to a .bogl file, 93% were satisfied with opening a .bogl file, and 93% were satisfied with creating a system diagram. All other features earned positive reactions from at least 95% of users. URL generation had the highest satisfaction rate, with favorable reception from 100% of users.

10.2.2.3 Improvements Requested by Users

A total of 41% of students requested improvements in the application. We gave higher priority to considering changes requested by more than one person, as this indicated more widespread problems in the BoGL Web interface and functionality. Four students suggested adding a label with an element's name that appears when the user hovers over that element in the canvas. Three students requested that elements in different domains with the same name have different images on the canvas. Two students asked for state equation generation as a feature. Two students requested that instructions on how to pan the window are provided somewhere in the application. Two students requested that there be modifiers available for elements in the electrical domain. The examples they provided were power dissipation, linkages, and AC/DC current.

We also received several suggestions given by only one person each. We gave lower precedence to these suggestions because we felt they did not represent the preferences of many of our users. One student requested that we move the "back" and "next" buttons in the tutorial and show it on first entry. One suggestion was for collision detection to be implemented in the system diagram canvas to keep elements from overlapping. We had one student request that we improve the bond graph layout algorithm to improve visibility in the canvas. One student wanted to be able to add an element to the canvas without dragging it in from the left menu; we assumed they wanted to be able to click on an element in the left menu and have it appear on the canvas.

Students also asked for more elements to be added as options in the system diagram, and for bugs introduced by pulleys in the system diagram to be resolved.

We also received comments from the section of the survey dedicated to the tutorial. Two students requested that the tutorial provide the names of all available system diagram elements. We received two requests for the tutorial navigation buttons – “back,” “done,” and “next” – to be moved so that the user can locate them more easily. One student asked for a fix for the bounding box and buttons getting pushed offscreen when the screen height is too small. Another student requested that the tutorial specify that gravity elements should not be connected to anything else.

10.2.2.4 Bugs Discovered by Users

Four students noticed that at least one element modifier was not set correctly in a system diagram generated from a .bogl file. One student noticed that another system diagram, when opened from a .bogl file, was missing a velocity modifier. Two students observed that a system diagram loaded from a URL was missing a component. One student noticed that an extra gravity element appeared in a generated causal bond graph.

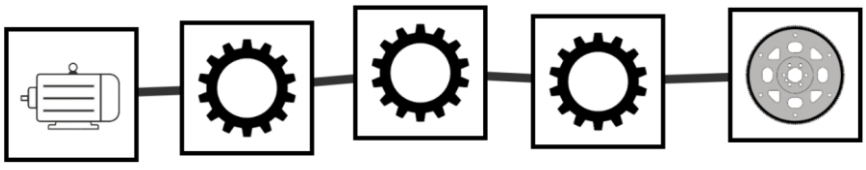


Figure 118: A system diagram that caused BoGL Web to crash when a user attempted to generate a bond graph from it.

Students also observed some issues that caused BoGL Web to crash. For example, the application would throw an error when the user attempted to generate bond graphs from some

system diagrams; an example is shown in Figure 118. We also observed that users could not download and open system diagrams from .bogl files in certain browsers. Four students encountered this problem when running BoGL Web in Firefox, and one more was unable to use file capabilities in Brave.

10.2.3 Qualitative Data Analysis

We observed in the data we received that all respondents had previously taken ME 4322 at WPI, and two said they had already completed the course. Most students were comfortable with modeling using system diagrams and bond graphs. No one had used BoGL Web as a modeling aid. One prominent result we noticed was overall user appreciation of the tutorial; all respondents were able to locate the tutorial, and all found it useful for using the application. As evidenced by the numerical appreciation scores assigned in the previous section, most people were at least somewhat satisfied by each feature.

10.2.4 Response to Survey Data

Overall, we observed general satisfaction with the application from our test subjects. From the improvements and bugs listed, we determined what we wanted to change in the application. We added tooltips with element names for system diagram icons in the left menu. We also fixed the file system bug for downloading and opening .bogl files in Firefox. We changed the system diagram icons for “ground” in the electrical domain to match the industry standard icon and to differentiate it from mechanical “ground.” In the future, we also plan to change the spring icon in the translation and rotational mechanical domains. We also decided to investigate new places to put the tutorial buttons to prevent them from being pushed off the screen. We added a new section in the tutorial explaining how to pan the canvas. In addition to specifying in the tutorial

that the “gravity” element should not be connected to any other element; we also created a separate compatibility group for gravity so that it cannot be attached to the rest of the system diagram.

We also predetermined which improvements or bugs provided by users to disregard. Some of the problems in our application were caused by missing or incorrect rules or rulesets, which we assumed Professor Radhakrishnan would handle. For example, any errors that the program threw when generating a bond graph with pulleys were caused by an omission of certain rules in the rotational domain. Any issues we found with gravity elements in the bond graph we also attributed to ruleset problems. We left adding support for new elements for future iterations of the BoGL Web project. We also determined that preventing element collision was not a useful feature to implement.

11 Conclusion

The goal of this MQP was to create a web application based on two previous MQPs which created a Windows Desktop application to help students taking ME 4322 to build models using Bond Graphs. Our team was able to successfully build this application using Blazor WebAssembly. The new web application matches the functionality of the previous desktop application in features, processing speed, and usability. Our team also added several additional features including a full undo/redo system, a system for exporting system diagrams using URLs, an interactive tutorial, and code that generates differential equations from bond graphs.

Our team also worked to create a set of mockups for displaying state equations which we then scored using a decision matrix. Using this decision matrix as well as through conversations with our advisors, we selected a set of mockups to be implemented. We then implemented a user interface to display state variables, state equations, and labels for the bond graph. While this is most of the work that needs to be done to fully support state equations, a future project will need to implement a user interface which allows users to assign labels and values to elements and modifiers.

Our team tested the effectiveness of BoGL Web by surveying the C-Term ME 4322 class. We found that students had a positive reaction to the application with many wishing they had been able to use it earlier in the course. Most student dissatisfaction came from a bug that prevented Firefox users from opening or saving .bogl files, plus other small UI, and backend bugs. These bugs were fixed in D-Term. Students also requested improvement of a few features, such as labeling system diagram elements with hover text, and asked that some features, such as panning the graph, be added to the tutorial.

Overall, this project was able to meet its goal of recreating BoGL Desktop as a web application, as well as adding several additional features to improve the user experience.

11.1 Student Reflection

In this section we will start with a discussion of the courses that helped our team with this project. We will then discuss the knowledge and skills that our team needed to complete this project. We will finish with how this project helped our team improve upon these skills.

These are several courses that helped our team when working on this project:

CS 3733, Software Engineering and CS 509, Design of Software Systems – This was a software engineering project, so it follows that the skills we learned during a software engineering class would be most applicable. In this course, our team members learned how to organize a large-scale software project, effectively use design patterns to structure code, and how to set achievable short-term goals.

CS 4241, Webware – This class taught us about many aspects of web development. The course covers the basics of JavaScript, explains how to debug web applications, and helps students build awareness of what is possible on a website. All these learned skills were useful in the creation of BoGL Web.

MIS 585, User Experience Design – This class gives students the tools they need to analyze and develop user-friendly UIs, which helped us develop mockups for the state equation section of the UI that we added.

MA 2271, Graph Theory – This class features introductory graph terminology, which we used to describe bond graphs from a more theoretical perspective.

CS 2103, Accelerate Object-Oriented Design Concepts – This class provides background on the composite design pattern and how to use it in parsing mathematical equations. We used a more complex string parsing process for generating state equations.

PH 1110, General Physics-Mechanics – This class provides background on how to compute position, velocity, and similar values for simple mechanical networks. The mathematical equations used to describe physical systems in this class were also used in state equation generation.

PH 1120, General Physics-Electricity and Magnetism – This class provides background on the relationship between current and voltage for elements such as resistors and capacitors. We used this information to aid in state equation generation.

For this project, our team first needed to develop an organization scheme. Using the skills, we had gained from Software Engineering, we were able to set up a Jira board to organize tasks. Throughout the term, we learned how to effectively use this board to control who was doing a task, specify when the task needed to be completed, and organize the branches in our GitHub. Effectively using GitHub was a skill we learned in Software Engineering that was reinforced throughout this project.

Deploying web applications was not something which our team had much experience with before, especially with Blazor project. We learned how to use Azure as well as GitHub Pages to deploy the web application. We also learned how to use continuous deployment to keep the web application up to date automatically whenever we merged a new branch into the main branch on GitHub. This was not something that is taught in a course, but it is used very commonly in industry.

Our team learned/improved upon our skills by using many tools when completing this project. These are many of the tools used in industry and they were extremely helpful throughout our project.

12 Future Work

There are several areas in which this project can be improved in the future. The first of these is by fully implementing the state equation UI proposed in Section 7.2.2.6. This feature will help students to check their homework solutions and, since we included the steps to view each state equation, students will be able to determine where they went wrong. This system will include a way to either copy and paste the state equations into a program such as MATLAB or solve them directly using BoGL Web.

12.1 Add Support for Element and Modifier Values and Labels

To fully support state equations, a future team will need to add support for labels and values. A proposed design for this system can also be found in Section 7.2.2.6. This system will allow for students to customize the state equations they create, thereby increasing readability. With the addition of labels and values, an update will need to be made to the system which generates URLs. This is because the compression system which we presented in Section 8.3.1 will no longer work since there will now be strings in the URLs. A future team will need to look at how to rework this system to support these labels and values being in the URLs.

12.2 Add Support for Lumped Parameter Models

In the field of mechanical engineering, there are other ways to model mechatronic systems, such as the lumped parameter modeling technique. This technique, like bond graphs, is used in ME/RBE 4322. Support for lumped parameter models would help students check their

solutions to homework problems using that modeling technique as well. This alternative way of modeling could be added as another tab alongside the system diagram and bond graph tabs.

12.3 Add Support for Rotating Elements

In a survey of students conducted in A-Term of 2022, there were several features mentioned that the students felt would improve BoGL Desktop. These features included the ability to rotate elements of a system diagram (see Figure 119). This would help students to better visualize the direction energy would flow through their system. For example, they would be able to rotate springs to be vertical instead of always horizontal if they were working on a system which was affected by gravity.

Figure 119: Example of Rotated Elements.

12.4 Add a Grid to the Canvas

Another feature that the students mentioned was the ability to have a grid for elements to snap to. This would help students create cleaner layouts for their system diagrams and could also be carried over to clean up the layouts of bond graphs. This feature could be toggled on and off in the Help menu with a checkbox like the multiple element deletion warning checkbox.

12.5 Add Support for User Accounts

One long term goal for the BoGL Web system is to add accounts so that students would be able to complete homework assignments using the web application. Developing a system to complete this task would require building a database to store user data, creating a way to log users in, and a way for professors to view assignments which have been submitted. A future team would need to look at how to design and implement these features to create a useful accounting system. These additional features would add to and improve the BoGL Web experience, helping students to better learn the content presented in ME 4322.

13 References

Abdul, S. (2022, August 18). *Can You Download Safari on Windows?* MUO. https://www.makeuseof.com/windows-download-safari/

About Bond Graphs. (n.d.). https://groups.csail.mit.edu/drl/journal_club/papers/Samantaray__2001__www.bondgraphs.com_about.pdf

Angular directives for Bootstrap. (n.d.). Retrieved October 3, 2022, from https://angular-ui.github.io/bootstrap/#!#getting_started

Angular Vs React: Difference Between Angular and React. (2021, August 23). InterviewBit. https://www.interviewbit.com/blog/angular-vs-react/

Angular—Understanding dependency injection. (n.d.). Retrieved October 10, 2022, from https://angular.io/guide/dependency-injection

Atlassian. (n.d.-a). *Jira | Issue & Project Tracking Software*. Atlassian. Retrieved March 16, 2023, from https://www.atlassian.com/software/jira

Atlassian. (n.d.-b). *What is version control | Atlassian Git Tutorial*. Atlassian. Retrieved March 20, 2023, from https://www.atlassian.com/git/tutorials/what-is-version-control

Axler, S. (2014). *Linear algebra done right*. Springer.

Blazor University—Blazor hosting models. (n.d.). Blazor University. Retrieved October 6, 2022, from https://blazor-university.com/overview/blazor-hosting-models

Blazorise Documentation. (n.d.). Blazorise. Retrieved October 3, 2022, from https://blazorise.com/docs

BlazorStrap 5. (n.d.). Retrieved October 3, 2022, from https://blazorstrap.io/V5/

Broenink, J. (1999). Introduction to Physical Systems Modelling with Bond Graphs. *University of Twente, Dept EE, Control Laboratory*.

Browser Statistics. (n.d.). Retrieved March 26, 2023, from https://www.w3schools.com/browsers/

Build software better, together. (n.d.). GitHub. Retrieved March 16, 2023, from https://github.com

Bulma: Free, open source, and modern CSS framework based on Flexbox. (n.d.). Retrieved April 9, 2023, from https://bulma.io/

Client. (2022, June 3). *Techopedia*. https://www.techopedia.com/definition/437/client

Courville, C., Hearst, H., & Jaber, T. (2020). *BoGL: An Application for Generating Bond Graphs*. Worcester Polytechnic Institute. https://web.wpi.edu/Pubs/E-project/Available/E-project-051820-023928/unrestricted/bogl_mqp_report.pdf

Create Your Azure Free Account Today | Microsoft Azure. (n.d.). Retrieved March 16, 2023, from https://azure.microsoft.com/en-us/free/search/?&ef_id=CjwKCAjw_MqgBhAGEiwAnYOAepxcuHwBjkDH1raVGUoYZbzTgHJTh41lbkdH7TP7C2OrpWsuCo7HlhoCXLoQAvD_BwE:G:s&OCID=AIDcmm5edswduu_SEM_CjwKCAjw_MqgBhAGEiwAnYOAepxcuHwBjkDH1raVGUoYZbzTgHJTh41lbkdH7TP7C2OrpWsuCo7HlhoCXLoQAvD_BwE:G:s&gclid=CjwKCAjw_MqgBhAGEiwAnYOAepxcuHwBjkDH1raVGUoYZbzTgHJTh41lbkdH7TP7C2OrpWsuCo7HlhoCXLoQAvD_BwE

De Moor, T. (2022, June 2). *14 Best React Component Libraries in 2022*. X-Team Blog - The Most-Loved Company for Engineers. https://x-team.com/blog/best-react-component-libraries/

Definition of asynchronous | Dictionary.com. (n.d.). Www.Dictionary.Com. Retrieved March 20, 2023, from https://www.dictionary.com/browse/asynchronous

Definition of repository | Dictionary.com. (n.d.). Www.Dictionary.Com. Retrieved March 20, 2023, from https://www.dictionary.com/browse/repository

Design Patterns. (n.d.). Retrieved March 21, 2023, from https://refactoring.guru/design-patterns

Di Battista, G. (Ed.). (1999). *Graph drawing: Algorithms for the visualization of graphs*. Prentice Hall.

Eades, P. (1984). *A Heurisrics for Graph Drawing*. Congressus numerantium.

Factory method for designing pattern. (2015, August 17). *GeeksforGeeks*. https://www.geeksforgeeks.org/factory-method-for-designing-pattern/

Fast and reliable end-to-end testing for modern web apps | Playwright. (n.d.). Retrieved April 6, 2023, from https://playwright.dev/

Gamma, E. (Ed.). (1995). *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley.

Gazta, A. (2021, October 12). *Understanding the pros and cons of Angular Development*. GreyCampus. https://www.greycampus.com/blog/programming/good-and-bad-of-angular-development

Git. (n.d.). Retrieved March 16, 2023, from https://git-scm.com/

GitHub Pages. (n.d.). GitHub Pages. Retrieved March 16, 2023, from https://pages.github.com/

Goudar, Y. (2019, January 17). *One-way and Two-way Data Binding in Angular*. https://www.pluralsight.com/utilities/promo-only?noLaunch=true

Graham, R., Grotschel, M., & Lovasz, L. (1995). *The Handbook of Combinatorics* (Vol. 1). Elsevier Science B.V.

Grande, D., & Mancini, F. (2016). *Graph Grammar Based Automated Virtual Lab for Bond Graphs* (p. 66). Worcester Polytechnic Institute. https://web.wpi.edu/Pubs/E-project/Available/E-project-042816-141000/unrestricted/Grande_ManciniAutomatedDesignMQP_A.pdf

guardrex. (2023, February 21). *Host and deploy ASP.NET Core Blazor WebAssembly*. https://learn.microsoft.com/en-us/aspnet/core/blazor/host-and-deploy/webassembly

How to make PWAs installable—Progressive web apps (PWAs) | MDN. (n.d.). Retrieved October 6, 2022, from https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Installable_PWAs

Hughes, J. (2021, December 13). 7 Best React UI Framework and Component Libraries in 2022. *ThemeIsle Blog*. https://themeisle.com/blog/best-react-ui-framework/

Implementing Stacks in Data Structures [Updated]. (n.d.). Simplilearn.Com. Retrieved March 20, 2023, from https://www.simplilearn.com/tutorials/data-structure-tutorial/stacks-in-data-structures

Interactive tool for creating directed graphs using d3.js. (n.d.). Retrieved October 12, 2022, from https://bl.ocks.org/cjrd/6863459

Introduction to progressive web apps—Progressive web apps (PWAs) | MDN. (n.d.). Retrieved September 8, 2022, from https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Introduction

Introduction to the DOM - Web APIs | MDN. (2023, February 22). https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction

Introduction—Ant Design. (n.d.). Retrieved October 3, 2022, from https://ant.design/docs/spec/introduce

Janssen, T. (2018, June 19). *Design Patterns Explained – Dependency Injection with Code Examples*. Stackify. https://stackify.com/dependency-injection/

JavaScript ES6. (n.d.). Retrieved March 26, 2023, from https://www.w3schools.com/js/js_es6.asp

JetBrains: Essential tools for software developers and teams. (n.d.). JetBrains. Retrieved March 16, 2023, from https://www.jetbrains.com/

js.foundation, J. F.-. (n.d.). *JQuery*. Retrieved March 20, 2023, from https://jquery.com/

Kindermann, P. (2021, April 7). *Visualization of Graphs*. https://www.youtube.com/watch?v=jBpE21MIDN4&list=PLubYOWSl9mIuJXdt_pMYoTD8QkaX9kQgO

Koren, Y. (2005). Drawing Graphs by Eigenvectors: Theory and Practice. *Computers & Mathematics with Applications*, *49*(11–12), 1867–1888. ScienceDirect. https://doi.org/10.1016/j.camwa.2004.08.015

Kumar, M. (2022, May 3). *What are Directives in Angular?* Medium. https://blog.bitsrc.io/directives-in-angular-6160ce805416

Lazaris, L. (2009, September 12). *The Z-Index CSS Property: A Comprehensive Look*. Smashing Magazine. https://www.smashingmagazine.com/2009/09/the-z-index-css-property-a-comprehensive-look/

Linked List in a Data Structure: All You Need to Know. (n.d.). Simplilearn.Com. Retrieved March 20, 2023, from https://www.simplilearn.com/tutorials/data-structure-tutorial/linked-list-in-data-structure

Lyles, T. (2020, March 11). *Google now allows devs to view visual limitations in its Chrome dev tools.* The Verge. https://www.theverge.com/2020/3/11/21174735/google-chrome-dev-tools-new-color-blind-friendly

Malavasi, A. (n.d.). *Top 10 nice free Blazor components*. Retrieved October 3, 2022, from https://medium.com/@alexandre.malavasi/top-10-nice-free-blazor-components-b42875e56b28

MatBlazor—Material Design components for Blazor. (n.d.). Retrieved October 3, 2022, from https://www.matblazor.com/

Material Design. (n.d.-a). Material Design. Retrieved October 3, 2022, from https://material.io/design/introduction

Material Design. (n.d.-b). Retrieved March 20, 2023, from https://m3.material.io/

Meersman, R., Tari, Z., & Herrero, P. (Eds.). (2008). *On the move to meaningful internet systems, OTM 2008 workshops: OTM Confederated International Workshops and posters, ADI, AWeSoMe, COMBEK, EI2N, IWSSA, MONET, OnToContent+QSI, ORM, PerSys, RDDS, SEMELS, and SWWS 2008, Monterrey, Mexico, November 9-14, 2008, proceedings*. Springer.

Menu—Ant Design. (n.d.). Retrieved October 11, 2022, from https://ant.design/components/menu/

.NET and .NET Core official support policy. (n.d.). Microsoft. Retrieved February 24, 2023, from https://dotnet.microsoft.com/en-us/platform/support/policy/dotnet-core

Operating system market share. (n.d.). Retrieved March 26, 2023, from https://netmarketshare.com/operating-system-market-share.aspx?options=%7B%22filter%22%3A%7B%22%24and%22%3A%5B%7B%22de

viceType%22%3A%7B%22%24in%22%3A%5B%22Desktop%2Flaptop%22%5D%7D%7D%5D%7D%2C%22dateLabel%22%3A%22Custom%22%2C%22attributes%22%3A%22share%22%2C%22group%22%3A%22platformVersion%22%2C%22sort%22%3A%7B%22share%22%3A-1%7D%2C%22id%22%3A%22platformsDesktopVersions%22%2C%22dateInterval%22%3A%22Monthly%22%2C%22dateStart%22%3A%222022-11%22%2C%22dateEnd%22%3A%222022-11%22%2C%22segments%22%3A%22-1000%22%7D

Ouellette, A. (n.d.). *What is Bootstrap? An Awesome 2022 Beginner's Guide*. Retrieved October 3, 2022, from https://careerfoundry.com/en/blog/web-development/what-is-bootstrap-a-beginners-guide/

Overview—Joy UI. (n.d.). Retrieved October 3, 2022, from https://mui.com/joy-ui/getting-started/overview/

Overview—Material UI. (n.d.). Retrieved October 3, 2022, from https://mui.com/material-ui/getting-started/overview/

Overview—MUI Base. (n.d.). Retrieved October 3, 2022, from https://mui.com/base/getting-started/overview/

Palantir. (2020, May 4). *Scaling product design with Blueprint*. Medium. https://blog.palantir.com/scaling-product-design-with-blueprint-25492827bb4a

Palantir / Projects / Blueprint. (n.d.). Dribbble. Retrieved October 3, 2022, from https://dribbble.com/Palantir/collections/451877-Blueprint

React-Bootstrap. (n.d.). Retrieved October 3, 2022, from https://react-bootstrap.github.io/

Real-time ASP.NET with SignalR | .NET. (n.d.). Microsoft. Retrieved October 6, 2022, from https://dotnet.microsoft.com/en-us/apps/aspnet/signalr

Scanlon, B., Zhang, Z., Reedy, R., Pool, D., Save, H., Long, D., Chen, J., Wolock, D., Conway, B., & Winester, D. (2015). *Hydrologic Implications of GRACE Satellite Data in the Colorado River Basin*.

Service Worker API - Web APIs | MDN. (2023, March 13). https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API

Shah, K. (n.d.). *8 Best Angular UI Frameworks For Web Development*. Third Rock Techkno. Retrieved October 3, 2022, from https://www.thirdrocktechkno.com/blog/8-best-angular-ui-frameworks-for-web-development//

Sharma, I. (2022, February 22). *7 Most Popular Angular Component Library*. Software and Technology Blog - TatvaSoft. https://www.tatvasoft.com/outsourcing/2022/02/7-best-angular-ui-frameworks.html

Squillace, M., & Harper, Q. (2022, August 12). Hard choices for the Colorado River. *WyoFile*.

Start – React Bootstrap 5 Admin Dashboard Theme. (n.d.). Bootstrap Themes. Retrieved October 3, 2022, from https://themes.getbootstrap.com/product/start-react-bootstrap-5-admin-dashboard-theme/

SVG: Scalable Vector Graphics | MDN. (n.d.). Retrieved October 12, 2022, from https://developer.mozilla.org/en-US/docs/Web/SVG

Swimberghe, N. (2020, August 31). *How to deploy ASP.NET Blazor WebAssembly to GitHub Pages*. Swimburger. https://swimburger.net/blog/dotnet/how-to-deploy-aspnet-blazor-webassembly-to-github-pages

Team, A. C. (n.d.). *Angular Material*. Angular Material. Retrieved October 3, 2022, from https://material.angular.io/

The Good and the Bad of Angular Development. (2022, September 6). Altexsoft. https://www.altexsoft.com/blog/engineering/the-good-and-the-bad-of-angular-development/

Thornton, J., & Otto, M. (n.d.). *Get started with Bootstrap*. Retrieved March 20, 2023, from https://getbootstrap.com/docs/5.3/getting-started/introduction/

Top Backend Developer Skills You Must Have (2023). (2023, February 16). InterviewBit. https://www.interviewbit.com/blog/backend-developer-skills/

Userware. (n.d.-a). *OpenSilver*. Retrieved February 24, 2023, from https://opensilver.net/

Userware. (n.d.-b). *What's New—OpenSilver*. Retrieved February 24, 2023, from https://opensilver.net/links/whatsnew.aspx

Varkki, K. (n.d.). *The Most Popular React UI Component Libraries in 2022*. Retrieved October 3, 2022, from https://www.sitepoint.com/popular-react-ui-component-libraries/

v-hearya. (n.d.). *Lifecycle FAQ - Internet Explorer and Microsoft Edge*. Retrieved March 26, 2023, from https://learn.microsoft.com/en-us/lifecycle/faq/internet-explorer-microsoft-edge

Visual Studio: IDE and Code Editor for Software Developers and Teams. (n.d.). Visual Studio. Retrieved March 20, 2023, from https://visualstudio.microsoft.com

W3Counter: Global Web Stats. (n.d.). Retrieved March 26, 2023, from https://www.w3counter.com/globalstats.php

Web app manifests | MDN. (2023, March 3). https://developer.mozilla.org/en-US/docs/Web/Manifest

welcome home: Vim online. (n.d.). Retrieved March 16, 2023, from https://www.vim.org/

What are Data Structures? - Definition from WhatIs.com. (n.d.). Data Management. Retrieved March 20, 2023, from https://www.techtarget.com/searchdatamanagement/definition/data-structure

What Does a Front-End Developer Do? (2023, February 10). Coursera. https://www.coursera.org/articles/front-end-developer

What Is Ajax Programming—Explained—KeyCDN Support. (n.d.). KeyCDN. Retrieved March 20, 2023, from https://www.keycdn.com/support/ajax-programming

What is CSS? - Learn web development | MDN. (2023, February 23). https://developer.mozilla.org/en-US/docs/Learn/CSS/First_steps/What_is_CSS

What is HTTP? (n.d.). Cloudflare. Retrieved March 20, 2023, from https://www.cloudflare.com/learning/ddos/glossary/hypertext-transfer-protocol-http/

What is user interface (UI)? Definition from SearchAppArchitecture. (n.d.). App Architecture. Retrieved March 20, 2023, from https://www.techtarget.com/searchapparchitecture/definition/user-interface-UI

Wilson, R. J. (2010). *Introduction to Graph Theory* (5th ed.). Pearson Education.

XML Essentials—W3C. (n.d.). Retrieved March 20, 2023, from https://www.w3.org/standards/xml/core

Yandex. (2023). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Yandex&oldid=1146323970

14 Appendices

Appendix A – ME/RBE 4322 Survey

Below is an exported version of the Qualtrics survey we conducted during a ME/RBE 4322 class during C-Term 2023, which was used to evaluate the effectiveness of BoGL Web:

Informed Consent

Q1

Welcome to the BoGL Web User Study!

Investigators:

Margaret Earnest, Jakob Misbach, Anthony Vuolo

Advisors:

David C. Brown, Pradeep Radhakrishnan, Brigitte Servatius

Sponsor:

Pradeep Radhakrishnan

Title of Research Study:

BoGL Web: Web-Based Bond Graph Laboratory

Contact Information:

gr-boglweb@wpi.edu

Introduction:

BoGL Web is a web-based application that provides a canvas interface on which mechanical engineering students can model physical and electrical networks by constructing system

diagrams and converting them to bond graphs. The design, functionality, and software of BoGL Web is inspired by BoGL, a desktop application that also models bond graphs.

Purpose of the Study:

We are interested in understanding the quality of user experience for this application. You will be presented with information relevant to the quality of the BoGL Web interface and capabilities, and you will be asked to answer some questions about it. Please be assured that your responses will be kept completely confidential.

Procedures to be Followed:

You will first be asked to answer some questions about your experiences with bond graph modeling. You will then be asked to open BoGL Web and complete and/or assess a series of tasks in the application. At the end of the survey, you will be asked some questions about your experience with each feature.

Record Keeping and Confidentiality

Professor Radhakrishnan will be receiving your name in association with the survey for tracking who used the application to complete a specified homework assignment. All analysis will be done anonymously by the investigators.

Risk to Participants:

None.

Benefits to Participants:

We will use the feedback from this survey to make changes to BoGL Web with the end goal of improving user experience.

The study should take you between 30-60 minutes to complete. If you would like to contact the Principal Investigator in the study to discuss this research, please e-mail gr-boglweb@wpi.edu.

Your participation in this research is voluntary. Your refusal to participate will not result in any penalty to you or any loss of benefits to which you may otherwise be entitled. You may decide to stop participating in the research at any time without penalty or loss of other benefits. The project investigators retain the right to cancel or postpone the experimental procedures at any time they see fit.

By clicking the button below, you acknowledge that you are 18 years of age or older.

Please note that this survey will be best displayed on a laptop or desktop computer. Some features may be less compatible for use on a tablet or mobile device.

- ○ I consent; begin the study. (1)

Q17 What is your experience with the following course: Modeling and Analysis of Mechatronic Systems (ME 4322)?

- ○ I have already completed the course. (1)
- ○ I am currently taking the course. (2)
- ○ I have never taken the course. (4)

Q38 How comfortable are you with the idea of modeling using bond graphs and system diagrams?

- ○ Extremely uncomfortable (1)
- ○ Somewhat uncomfortable (2)
- ○ Neither comfortable nor uncomfortable (3)
- ○ Somewhat comfortable (4)
- ○ Extremely comfortable (5)

Q23 Have you ever used any form of BoGL (Bond Graph Lab) before?

- ○ Yes (1)

- No (2)

Open BoGL Web for Practical Portion

Q30 Go to the following link to access BoGL Web: https://boglweb.github.io/. Please keep this URL somewhere you can copy it from, as you will be asked to close and reopen the application several times.

Follow the Tutorial

Q32 In BoGL Web, try to find the tutorial. Proceed to the next page if you find the tutorial or cannot find it and give up.

Tutorial

Q27 Were you able to locate the tutorial?

- Yes (1)
- No (2)

If Q27 is "No"

Q48 The tutorial is an option in the 'Help' menu. Please complete the tutorial now and then return to the survey.

Q28 Did you find the tutorial helpful for learning how to use BoGL Web?

- Yes (2)
- No (3)

Q29 Are there any additions to the tutorial that you would like to see?

- Yes (4)

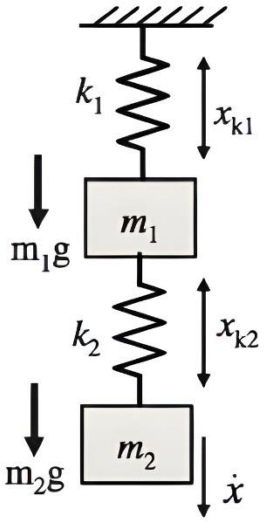
- No (5)

If Q29 is "Yes"

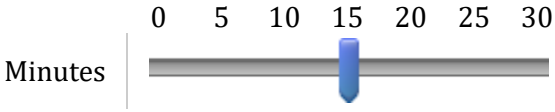
Q19 What additions would you like to see?

Make System Diagram 1

Q36 Construct a system diagram for this system in BoGL Web and generate a bond graph for it, then return to this survey. Do not close BoGL Web or delete your system diagram.



Q23 How long did it take you to create the system diagram and generate the bond graph? If it took you more than 30 minutes, set the slider to 30.



Q24 How easy was it to make the system diagram?

- Very easy (1)
- Somewhat easy (2)
- Neither easy nor difficult (3)

- ○ Somewhat difficult (4)
- ○ Very difficult (5)

Q25 How easily would you be able to make the same system diagram and bond graph a second time?

- ○ With great ease (1)
- ○ With some ease (2)
- ○ Not easily (4)

Remake System Diagram 1

Q28 Generate a URL from the system diagram and enter the URL into a new tab. Check off all that apply to the system diagram created from that URL, using the previous diagram as a reference.

- □ At least one component is missing (1)
- □ At least one bond is missing. (2)
- □ At least one modifier is not set correctly. (3)
- □ At least one extra component was added. (4)
- □ At least one extra bond was added. (5)
- □ This system diagram is identical to the previous one. (6)
- □ Another issue was identified. (7)

Q31 Please submit the URL you generated in the previous question.

Q29 Generate a .bogl file from the system diagram and open that file in BoGL Web a new tab. Check off all that apply to the system diagram created from that file, using the previous diagram as a reference.

- □ At least one component is missing. (1)
- □ At least one bond is missing. (2)
- □ At least one modifier is not set correctly. (3)
- □ At least one extra component was added. (4)
- □ At least one extra bond was added. (5)
- □ This system diagram is identical to the previous one. (6)
- □ Another issue was identified. (7)

Q30 Please upload the .bogl file you generated in the previous question.

Q32 Please screenshot your system diagram and upload it.

Complete Homework 4

Q31 Before proceeding, please complete Homework 4 in BoGL Web. When you are finished, return to the survey, but keep BoGL Web open.

End Practical Portion of Survey

Q34 This concludes the testing portion of the survey. You may close BoGL Web if you wish before proceeding.

Start of Block: Navigation and Tools

Q33 How easy was each task?

	Extremely difficult (13)	Somewhat difficult (14)	Neither easy nor difficult (15)	Somewhat easy (16)	Extremely easy (17)
Construct a System Diagram (1)	○	○	○	○	○
Generate a Bond Graph (2)	○	○	○	○	○
Add or Remove Modifiers (3)	○	○	○	○	○

Q25 Were there any features missing from BoGL Web that you would like to see in the future?

- ○ Yes (1)
- ○ No (2)

If Q25 is "Yes"

Q26 What additional features would you like to see?

Bugs

Q30 Were there any issues or bugs that you encountered while using the app?

- ○ Yes (1)
- ○ No (2)

If Q30 is "Yes"

Q31 What issues/bugs did you encounter?

Wrap-up

Q35 How satisfied are you with the overall performance and functionality of these features of the app?

	Extremely dissatisfied (9)	Somewhat dissatisfied (10)	Neither satisfied nor dissatisfied (11)	Somewhat satisfied (12)	Extremely satisfied (13)
Responsiveness (1)	○	○	○	○	○
Creating a System Diagram (4)	○	○	○	○	○
Generating a Bond Graph (5)	○	○	○	○	○
Adding Modifiers (6)	○	○	○	○	○
Using the Tutorial (7)	○	○	○	○	○
Generating URLs (8)	○	○	○	○	○
Saving a System Diagram to a .bogl File (9)	○	○	○	○	○
Opening a .bogl File (10)	○	○	○	○	○
Loading Example Files (13)	○	○	○	○	○

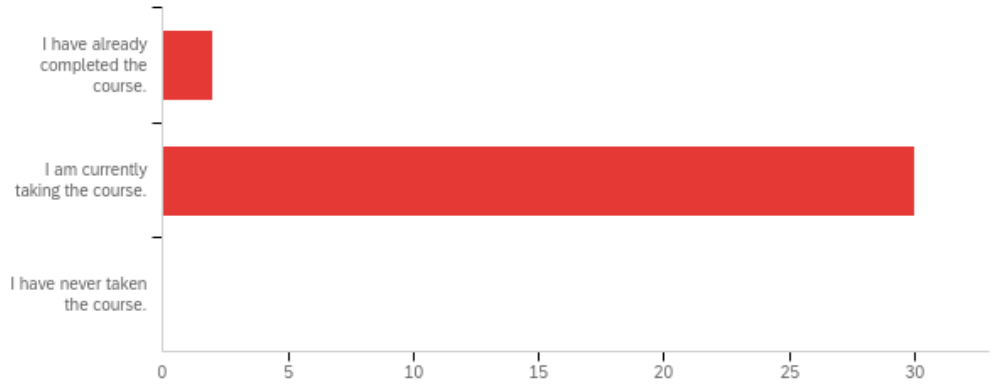
Appendix B – Survey Data

This is the raw data collected during the survey. Some question text has been summarized for conciseness; reference Appendix A – ME/RBE 4322 Survey for the full text of each question.

Q1 - Consent

All users consented.

Q17 - What is your experience with the following course: Modeling and Analysis of Mechatronic Systems (ME 4322)?

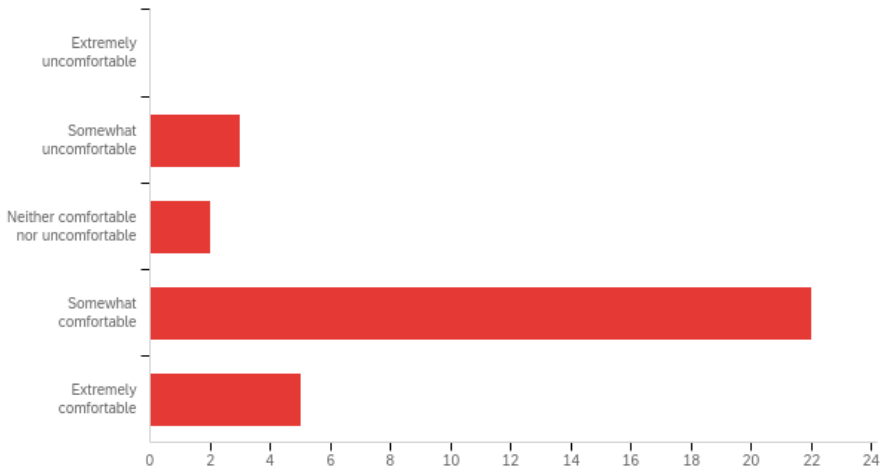


#	Field	Minimum	Maximum	Mean	Std Deviation	Variance	Count
1	Q17	1.00	2.00	1.94	0.24	0.06	32

#	Answer	%	Count
1	I have already completed the course.	6.25%	2
2	I am currently taking the course.	93.75%	30
4	I have never taken the course.	0.00%	0

	Total	100%	32

Q38 - How comfortable are you with the idea of modeling using bond graphs and system diagrams?



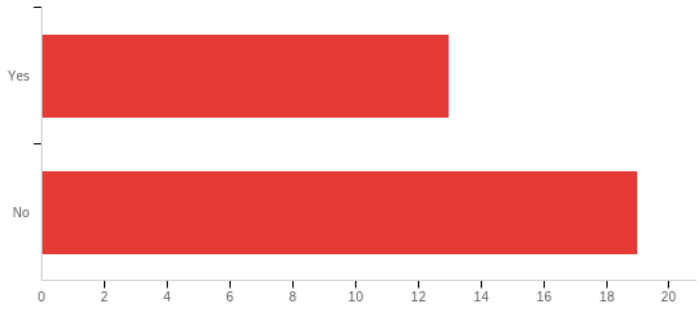
#	Field	Minimum	Maximum	Mean	Std Deviation	Variance	Count
1	Q38	2.00	5.00	3.91	0.76	0.58	32

#	Answer	%	Count
1	Extremely uncomfortable	0.00%	0
2	Somewhat uncomfortable	9.38%	3
3	Neither comfortable nor uncomfortable	6.25%	2
4	Somewhat comfortable	68.75%	22
5	Extremely comfortable	15.63%	5
	Total	100%	32

Q23 - Have you ever used any form of BoGL (Bond Graph Lab) before?

All users responded “No”.

Q25 - Were there any features missing from BoGL Web that you would like to see in the future?



#	Field	Minimum	Maximum	Mean	Std Deviation	Variance	Count
1	Q25	1.00	2.00	1.59	0.49	0.24	32

#	Answer	%	Count
1	Yes	40.63%	13
2	No	59.38%	19
	Total	100%	32

Q26 - What additional features would you like to see?

Labels for all of the elements in the drop downs, The ability to click on an element and then where you want it to go and have it appear without dragging, a pan function for the system, a way to update the view of the bond graphs so that you can change the organization of all 3 at once or sync them after changing one of them, it would also be nice if the zoom function was centered one the design space regardless of window size

Center the Zoom in the bottom right, add more elements, add descriptions of elements so we know what each one is, change pulleys such that you can attach things on each side of the pulley
State Equations; adding modifiers to electrical elements (such as power dissipation in a wire); linkages (i.e., revolute joints)
none
distinction of system elements, i.e., tortional vs translational spring
Chrome: The ability to pan the canvas (currently you can only move it by zooming out, moving the mouse, and then zooming in), When the site is opened for the first time, ask the user if they would like to see a tutorial, in the tutorial screen split back and next to the left and right edges and then have skip in the middle, ** Make it clear when a spring/damper is translational or rotational, Prevent links from going behind other items and linking to another item
when you put your mouse over the elements the name of the element appears
The addition of generating state equations may be helpful, although unfortunately I don't know how your group could implement this.
Elevator where carriage and counterweight move in both directions bond graph
When hovering over icons, have a text box that says what that button is (like if I hover over a spring, it should say translational spring). Increase default text size of the generated bond graphs
It'd be good to have different shapes used for some of the components in the system diagram. For example, the translational and torsional spring look the same, so there should be a visible difference to avoid confusion
I wish there was a label for the motor to switch between AC and DC.
When creating the bond graph, having the display be more organized and easy to follow so you don't have to drag and rearrange elements to be more easily understood.

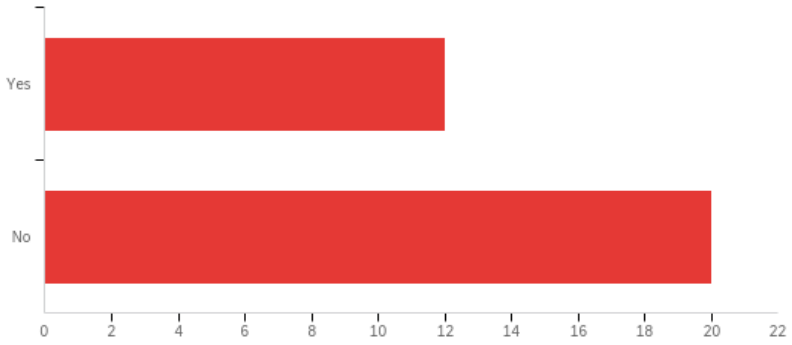
Q27 - Were you able to locate the tutorial?

All users responded “Yes.”

Q28 - Did you find the tutorial helpful for learning how to use BoGL Web?

All users responded “Yes.”

Q30 - Were there any issues or bugs that you encountered while using the app?



#	Field	Minimum	Maximum	Mean	Std Deviation	Variance	Count
1	Q30	1.00	2.00	1.63	0.48	0.23	32

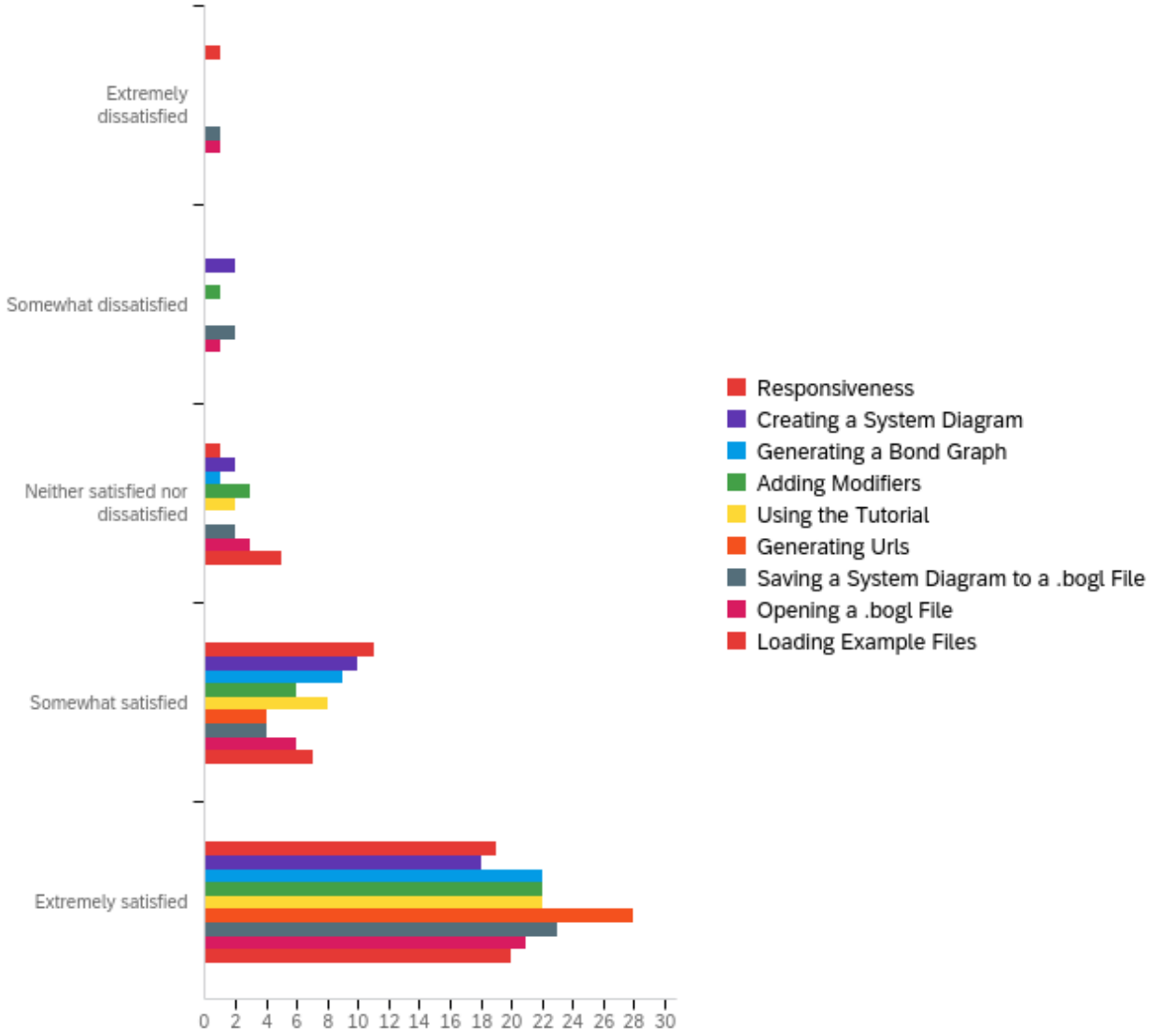
#	Answer	%	Count
1	Yes	37.50%	12
2	No	62.50%	20
	Total	100%	32

Q31 - What issues/bugs did you encounter?

There's an extra gravity element in the bond graph, it doesn't impact the bond graph and Se, but it is there's a text "gravity" in the final bond graph that's not connected to anything
I was unable to download the file from the program (the included download for this submission is from another student so I could proceed with the survey) I was using the chromium powered browser Brave on a windows machine.
Could not generate the system in problem 1a in homework 4
I made a motor-gear-gear-gear-wheel system and when I clicked generate the page crashed (Microsoft Edge))
Whenever I tried to save from Firefox, "An unhandled error has occurred" message came up. Whenever I tried to open a .bogl file in Firefox, nothing would happen.
bogl files do not save on Firefox

Firefox file I/O is broken. URL paste in Firefox deletes gravity.
I tried to make an elevator that had a counterweight on one side and a pulley with another elevator system on the other and it crashed. ITS LATE. WELCOME TO THE MANIA
I cannot save a .bogl file. I am using Firefox on Ubuntu 20.04
Bogl crashed while generating bond graph for elevator where carriage and counterweight move in same direction.
When I opened a bogl file, it was missing the velocity modifier
When copying and pasting the URL of a system and comparing the bond graphs, they were not consistent despite being the same system. In addition, when saving the system as a file and opening it in another tab, the bond graph was different and was missing components.

Q35 - How satisfied are you with the overall performance and functionality of these features of the app?



#	Field	Minimum	Maximum	Mean	Std Deviation	Variance	Count
1	Responsiveness	9.00	13.00	12.47	0.83	0.69	32
2	Creating a System Diagram	10.00	13.00	12.38	0.86	0.73	32
3	Generating a Bond Graph	11.00	13.00	12.66	0.54	0.29	32
4	Adding Modifiers	10.00	13.00	12.53	0.79	0.62	32
5	Using the Tutorial	11.00	13.00	12.63	0.60	0.36	32
6	Generating URLs	12.00	13.00	12.88	0.33	0.11	32
7	Saving a System Diagram to a .bogl File	9.00	13.00	12.44	1.06	1.12	32
8	Opening a .bogl File	9.00	13.00	12.41	1.00	0.99	32
9	Loading Example Files	11.00	13.00	12.47	0.75	0.56	32

#	Question	Extremely dissatisfied		Somewhat dissatisfied		Neither satisfied nor dissatisfied		Somewhat satisfied		Extremely satisfied		Total
1	Responsiveness	3.13%	1	0.00%	0	3.13%	1	34.38%	11	59.38%	19	32
2	Creating a System Diagram	0.00%	0	6.25%	2	6.25%	2	31.25%	10	56.25%	18	32
3	Generating a Bond Graph	0.00%	0	0.00%	0	3.13%	1	28.13%	9	68.75%	22	32
4	Adding Modifiers	0.00%	0	3.13%	1	9.38%	3	18.75%	6	68.75%	22	32
5	Using the Tutorial	0.00%	0	0.00%	0	6.25%	2	25.00%	8	68.75%	22	32
6	Generating URLs	0.00%	0	0.00%	0	0.00%	0	12.50%	4	87.50%	28	32
7	Saving a System Diagram to a .bogl File	3.13%	1	6.25%	2	6.25%	2	12.50%	4	71.88%	23	32
8	Opening a .bogl File	3.13%	1	3.13%	1	9.38%	3	18.75%	6	65.63%	21	32
9	Loading Example Files	0.00%	0	0.00%	0	15.63%	5	21.88%	7	62.50%	20	32

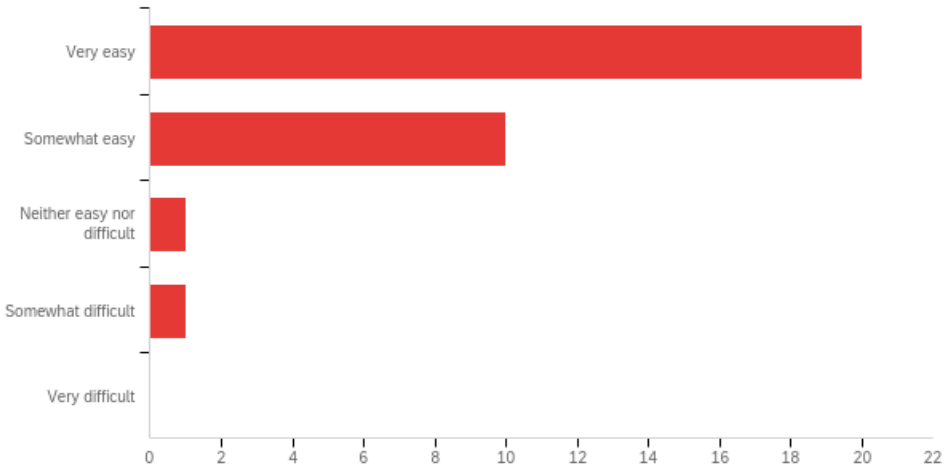
Q19 - What additions would you like to see?

The button that would typically advance the tutorial is in the same location as the button to go back to the previous page on the final slide which lead to some confusion why the tutorial was looping for me
More in depth explanation of the different elements
I was not able to see the next button even though it was there for others
point out names of the system elements
Make the "Done" button not gray, and put it in the location of the back button on the final page, so if you just keep pressing "next" you will finish the tutorial instead of going back one page
Please state that Gravity is not supposed to be connected to anything

Q23 - How long did it take you to create the system diagram and generate the bond graph? If it took you more than 30 minutes, set the slider to 30.

#	Field	Minimum	Maximum	Mean	Std Deviation	Variance	Count
1	Minutes	1.00	10.00	4.28	2.28	5.20	32

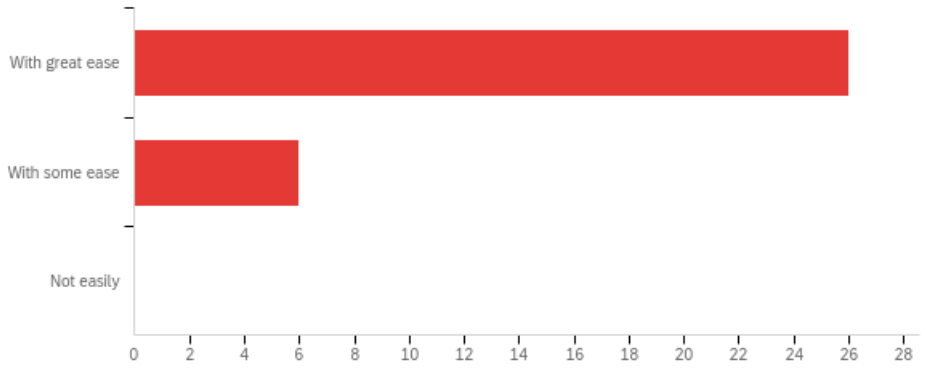
Q24 - How easy was it to make the system diagram?



#	Field	Minimum	Maximum	Mean	Std Deviation	Variance	Count
1	Q24	1.00	4.00	1.47	0.71	0.50	32

#	Answer	%	Count
1	Very easy	62.50%	20
2	Somewhat easy	31.25%	10
3	Neither easy nor difficult	3.13%	1
4	Somewhat difficult	3.13%	1
5	Very difficult	0.00%	0
	Total	100%	32

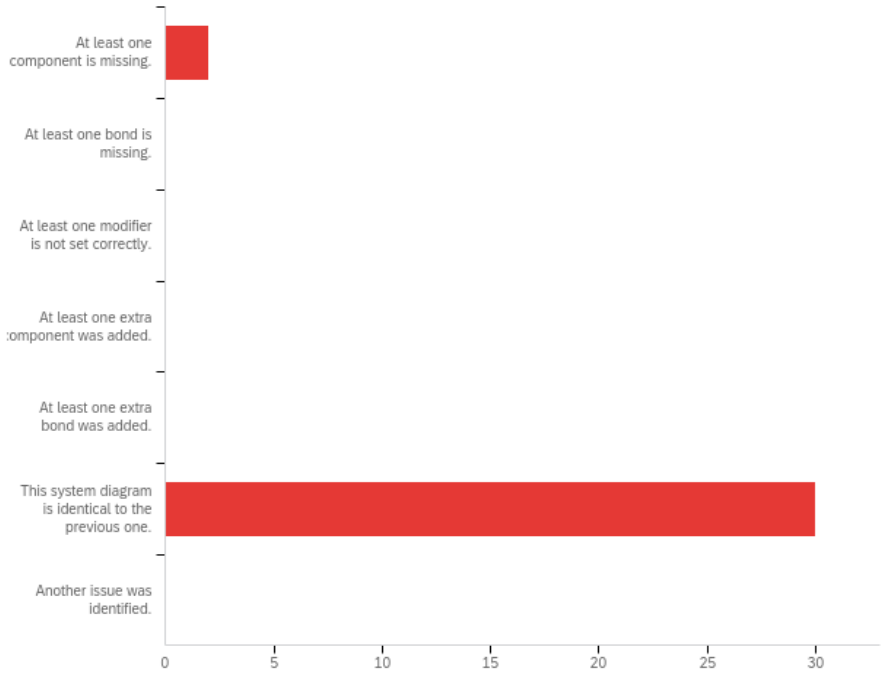
Q25 - How easily would you be able to make the same system diagram and bond graph a second time?



#	Field	Minimum	Maximum	Mean	Std Deviation	Variance	Count
1	Q25	1.00	2.00	1.19	0.39	0.15	32

#	Answer	%	Count
1	With great ease	81.25%	26
2	With some ease	18.75%	6
4	Not easily	0.00%	0
	Total	100%	32

Q28 - Generate a URL from the system diagram and enter the URL into a new tab. Check off all that apply to the system diagram created from that URL, using the previous diagram as a reference.



#	Answer	%	Count
1	At least one component is missing.	6.25%	2
2	At least one bond is missing.	0.00%	0
3	At least one modifier is not set correctly.	0.00%	0
4	At least one extra component was added.	0.00%	0
5	At least one extra bond was added.	0.00%	0
6	This system diagram is identical to the previous one.	93.75%	30
7	Another issue was identified.	0.00%	0
	Total	100%	32

Q31 - Please submit the URL you generated in the previous question.

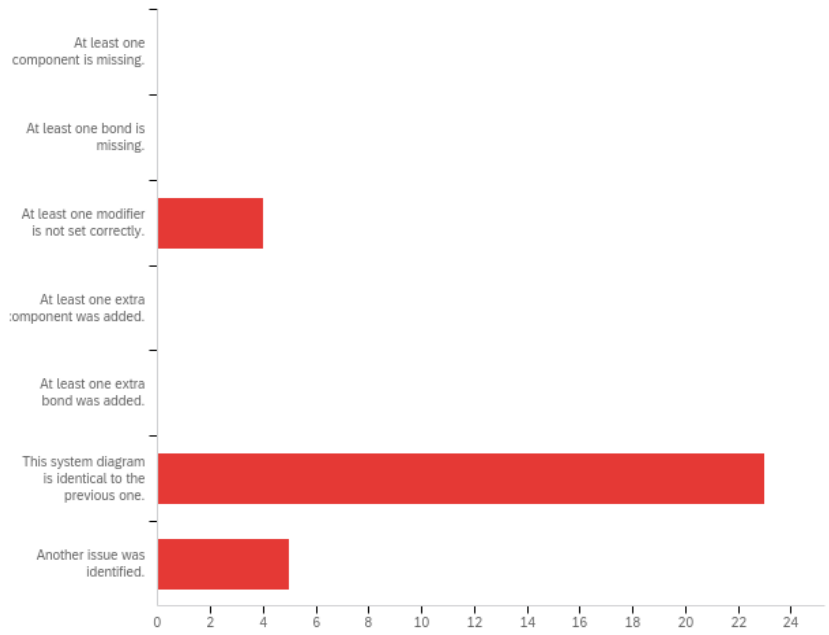
https://boglweb.github.io/?q=f1h-207.2h-220.0i0h1c2h-203.4h-93.6i0h0c3h-201.5h30.8i0h1c4h-201.5h159.1i0h0c5h-211.0h-350.2i0h3c6h-100.0h-87.9i0h5a1h-207.2h-220.0i0h1c2h-203.4h-93.6i0h0g0b2h-203.4h-93.6i0h0c3h-201.5h30.8i0h1g0b3h-201.5h30.8i0h1c4h-201.5h159.1i0h0g0b1h-207.2h-220.0i0h1c5h-211.0h-350.2i0h3g0e
https://boglweb.github.io/?q=f1h39.0h2.5i0h0c2h42.0h-172.5i0h3c3h40.0h172.5i0h0c4h40.0h83.5i0h1c5h-40.0h-11.5i0h5c6h-42.0h160.5i0h5c7h42.0h-85.5i0h1a4h40.0h83.5i0h1c1h39.0h2.5i0h0g0b3h40.0h172.5i0h0c4h40.0h83.5i0h1g0b7h42.0h-85.5i0h1c2h42.0h-172.5i0h3g0b1h39.0h2.5i0h0c7h42.0h-85.5i0h1g0e

https://boglweb.github.io/?q=f1h-564.5h-258.1i0h3c2h-564.7h-154.9i0h1c3h-563.1h-56.8i0h0c4h-565.0h52.6i0h1c5h-721.6h-170.0i0h5c6h-565.0h166.8i0h0a2h-564.7h-154.9i0h1c1h-564.5h-258.1i0h3g0b3h-563.1h-56.8i0h0c2h-564.7h-154.9i0h1g0b4h-565.0h52.6i0h1c3h-563.1h-56.8i0h0g0b4h-565.0h52.6i0h1c6h-565.0h166.8i0h0g0e
https://boglweb.github.io/?q=f8h-179.6h-109.9i0h0c9h-188.0h152.0i0h0c10h-181.0h30.8i0h1c11h-181.0h-189.8i0h1c12h-183.1h-280.8i0h3c13h-497.5h-202.4i0h5a12h-183.1h-280.8i0h3c11h-181.0h-189.8i0h1g0b11h-181.0h-189.8i0h1c8h-179.6h-109.9i0h0g0b8h-179.6h-109.9i0h0c10h-181.0h30.8i0h1g0b10h-181.0h30.8i0h1c9h-188.0h152.0i0h0g0e
https://boglweb.github.io/?q=f4h-332.5h-56.5i0h27c5h-220.5h-60.5i0h22c6h-121.5h-77.5i0h21a4h-332.5h-56.5i0h27c5h-220.5h-60.5i0h22g0b5h-220.5h-60.5i0h22c6h-121.5h-77.5i0h21g0e
https://boglweb.github.io/?q=f1h-174.5h90.8i0h0c2h-175.5h-50.2i0h0c3h-175.5h18.8i0h1c4h-174.5h-121.2i0h1c5h-175.5h-195.2i0h3c6h100.5h-110.2i0h5a5h-175.5h-195.2i0h3c4h-174.5h-121.2i0h1g0b4h-174.5h-121.2i0h1c2h-175.5h-50.2i0h0g0b2h-175.5h-50.2i0h0c3h-175.5h18.8i0h1g0b3h-175.5h18.8i0h1c1h-174.5h90.8i0h0g0e
https://boglweb.github.io/?q=f1h-315.0h-145.9i4h0c3h-317.9h45.4i4h0c4h-316.9h-58.3i0h1c5h-319.0h-316.6i0h3c6h-319.0h-234.2i0h1c8h-319.6h137.0i0h6c9h-414.6h-144.1i0h5a1h-315.0h-145.9i4h0c4h-316.9h-58.3i0h1g0b4h-316.9h-58.3i0h1c3h-317.9h45.4i4h0g0b5h-319.0h-316.6i0h3c6h-319.0h-234.2i0h1g0b6h-319.0h-234.2i0h1c1h-315.0h-145.9i4h0g0b3h-317.9h45.4i4h0c8h-319.6h137.0i0h6g0e
https://boglweb.github.io/?q=f7h-623.5h-318.0i0h1c8h-618.5h-237.0i4h0c9h-619.5h-156.0i0h1c10h-623.5h-55.0i4h0c11h-763.5h-265.0i0h5c12h-624.5h-395.0i0h3a12h-624.5h-395.0i0h3c7h-623.5h-318.0i0h1g0b7h-623.5h-318.0i0h1c8h-618.5h-237.0i4h0g0b9h-619.5h-156.0i0h1c8h-618.5h-237.0i4h0g0b9h-619.5h-156.0i0h1c10h-623.5h-55.0i4h0g0e
https://boglweb.github.io/?q=f10h-515.8h-445.5i0h3c11h-515.8h-351.5i0h1c12h-510.5h-254.9i0h0c13h-508.7h-159.0i0h1c14h-503.5h-62.3i0h0a10h-515.8h-445.5i0h3c11h-515.8h-351.5i0h1g0b11h-515.8h-351.5i0h1c12h-510.5h-254.9i0h0g0b12h-510.5h-254.9i0h0c13h-508.7h-159.0i0h1g0b13h-508.7h-159.0i0h1c14h-503.5h-62.3i0h0g0e
https://boglweb.github.io/?q=f12h1.4h-234.0i0h3c13h-0.1h-128.8i0h1c14h-0.1h-8.7i0h0c15h-1.4h111.3i0h1c16h1.4h234.1i0h0c6h-206.0h-240.6i0h5a12h1.4h-234.0i0h3c13h-0.1h-128.8i0h1g0b13h-0.1h-128.8i0h1c14h-0.1h-8.7i0h0g0b14h-0.1h-8.7i0h0c15h-1.4h111.3i0h1g0b15h-1.4h111.3i0h1c16h1.4h234.1i0h0g0e
https://boglweb.github.io/?q=f1h-542.5h-274.0i0h3c2h-518.5h-166.0i0h1c3h-508.5h-64.0i0h0c4h-505.5h76.0i0h1c5h-512.5h191.0i0h0c6h-734.5h-359.0i0h5a1h-542.5h-274.0i0h3c2h-518.5h-166.0i0h1g0b2h-518.5h-166.0i0h1c3h-508.5h-64.0i0h0g0b3h-508.5h-64.0i0h0c4h-505.5h76.0i0h1g0b4h-505.5h76.0i0h1c5h-512.5h191.0i0h0g0e
https://boglweb.github.io/?q=f1h-409.5h-261.9i0h3c2h-409.5h-188.9i0h1c3h-403.5h-115.9i0h0c4h-401.5h-31.9i0h1c5h-398.5h55.1i0h0c6h-574.5h15.1i0h5a2h-409.5h-188.9i0h1c1h-409.5h-261.9i0h3g0b3h-403.5h-115.9i0h0c2h-409.5h-188.9i0h1g0b4h-401.5h-31.9i0h1c3h-403.5h-115.9i0h0g0b5h-398.5h55.1i0h0c4h-401.5h-31.9i0h1g0e
https://boglweb.github.io/?q=f1h-1527.4h55.4i0h3c2h-1525.8h192.2i0h0c3h-1523.0h351.7i0h0c4h-1617.2h119.5i0h1c5h-1599.5h274.3i0h1c6h-1407.9h192.2i0h5a1h-1527.4h55.4i0h3c4h-1617.2h119.5i0h1g0b4h-1617.2h119.5i0h1c2h-1525.8h192.2i0h0g0b2h-1525.8h192.2i0h0c5h-1599.5h274.3i0h1g0b5h-1599.5h274.3i0h1c3h-1523.0h351.7i0h0g0e
https://boglweb.github.io/?q=f1h-391.5h-186.7i0h3c2h-391.5h-66.7i0h1c3h-385.2h62.6i0h0c4h-393.3h177.6i0h1c5h-394.9h286.2i0h0c6h-462.9h63.8i0h5c7h-470.9h286.8i0h5a1h-391.5h-186.7i0h3c2h-391.5h-66.7i0h1g0b2h-391.5h-66.7i0h1c3h-385.2h62.6i0h0g0b3h-385.2h62.6i0h0c4h-393.3h177.6i0h1g0b4h-393.3h177.6i0h1c5h-394.9h286.2i0h0g0e
https://boglweb.github.io/?q=f1h-470.5h-247.5i0h3c3h-464.5h254.5i4h0c4h-468.5h-136.5i0h1c5h-466.5h99.5i0h1c7h-356.5h-93.5i0h5c8h-467.5h-18.5i4h0a3h-464.5h254.5i4h0c5h-466.5h99.5i0h1g0b4h-468.5h-136.5i0h1c8h-467.5h-18.5i4h0g0b8h-467.5h-18.5i4h0c5h-466.5h99.5i0h1g0b4h-468.5h-136.5i0h1c1h-470.5h-247.5i0h3g0e
https://boglweb.github.io/?q=f1h-218.5h-31.2i4h0c2h-223.5h137.8i4h0c3h-213.5h-217.2i0h3c4h-225.5h-123.2i0h1c5h-212.5h49.8i0h1c6h-432.5h-67.2i0h5a2h-223.5h137.8i4h0c5h-212.5h49.8i0h1g0b5h-212.5h49.8i0h1c1h-218.5h-31.2i4h0g0b1h-218.5h-31.2i4h0c4h-225.5h-123.2i0h1g0b4h-225.5h-123.2i0h1c3h-213.5h-217.2i0h3g0e

https://boglweb.github.io/?q=f7h-246.7h-200.9i0h3c8h-246.0h-130.3i0h1c9h-252.8h-57.0i0h0c10h-251.8h21.4i0h1c11h-262.0h100.1i4h0c14h-39.7h-68.5i0h5a7h-246.7h-200.9i0h3c8h-246.0h-130.3i0h1g0b8h-246.0h-130.3i0h1c9h-252.8h-57.0i0h0g0b9h-252.8h-57.0i0h0c10h-251.8h21.4i0h1g0b10h-251.8h21.4i0h1c11h-262.0h100.1i4h0g0e
https://boglweb.github.io/?q=f1h-531.5h-206.1i0h3c3h-528.5h-123.1i0h1c4h-524.5h-40.1i4h0c5h-525.5h36.9i0h1c6h-523.5h119.9i4h0c7h-394.5h-93.1i0h5a1h-531.5h-206.1i0h3c3h-528.5h-123.1i0h1g0b3h-528.5h-123.1i0h1c4h-524.5h-40.1i4h0g0b4h-524.5h-40.1i4h0c5h-525.5h36.9i0h1g0b5h-525.5h36.9i0h1c6h-523.5h119.9i4h0g0e
https://boglweb.github.io/?q=f2h-539.5h-139.5i0h3c3h-537.5h-26.5i0h1c4h-534.5h76.5i0h0c5h-529.5h169.5i0h1c6h-530.5h274.5i0h0c8h-434.5h-234.5i0h5a3h-537.5h-26.5i0h1c2h-539.5h-139.5i0h3g0b3h-537.5h-26.5i0h1c4h-534.5h76.5i0h0g0b4h-534.5h76.5i0h0c5h-529.5h169.5i0h1g0b5h-529.5h169.5i0h1c6h-530.5h274.5i0h0g0e
https://boglweb.github.io/?q=f1h-365.5h-235.5i0h3c2h-362.5h-129.5i0h1c3h-362.5h-43.5i0h0c4h-356.5h54.5i0h1c5h-351.5h139.5i0h0c6h-458.5h-244.5i0h5a1h-365.5h-235.5i0h3c2h-362.5h-129.5i0h1g0b2h-362.5h-129.5i0h1c3h-362.5h-43.5i0h0g0b3h-362.5h-43.5i0h0c4h-356.5h54.5i0h1g0b4h-356.5h54.5i0h1c5h-351.5h139.5i0h0g0e
https://boglweb.github.io/?q=f11h-80.6h94.3i4h0c12h-84.0h-83.1i0h0c13h-85.7h2.2i0h1c14h-84.0h-169.2i0h1c15h-90.0h-251.9i0h3c16h-240.9h-135.9i0h5a14h-84.0h-169.2i0h1c12h-84.0h-83.1i0h0g0b12h-84.0h-83.1i0h0c13h-85.7h2.2i0h1g0b13h-85.7h2.2i0h1c11h-80.6h94.3i4h0g0b15h-90.0h-251.9i0h3c14h-84.0h-169.2i0h1g0e
https://boglweb.github.io/?q=f1h-89.0h-254.5i0h3c2h-88.0h-146.5i0h1c3h-87.0h-44.5i0h0c4h-89.0h60.5i0h1c5h-89.0h165.5i0h0c6h-180.0h-38.5i0h5c7h-178.0h161.5i0h5a2h-88.0h-146.5i0h1c1h-89.0h-254.5i0h3g0b2h-88.0h-146.5i0h1c3h-87.0h-44.5i0h0g0b3h-87.0h-44.5i0h0c4h-89.0h60.5i0h1g0b4h-89.0h60.5i0h1c5h-89.0h165.5i0h0g0e
https://boglweb.github.io/?q=f5h-110.2h-121.1i0h0c6h-102.2h-25.1i0h1c7h-111.2h-194.1i0h1c8h-103.2h96.9i4h0c9h-109.2h-263.1i0h3c10h-279.2h-129.1i0h5a9h-109.2h-263.1i0h3c7h-111.2h-194.1i0h1g0b7h-111.2h-194.1i0h1c5h-110.2h-121.1i0h0g0b5h-110.2h-121.1i0h0c6h-102.2h-25.1i0h1g0b6h-102.2h-25.1i0h1c8h-103.2h96.9i4h0g0e
https://boglweb.github.io/?q=f8h-309.5h-207.5i0h3c9h-308.5h-78.5i0h1c10h-298.5h11.5i0h0c11h-300.5h109.5i0h1c12h-296.5h206.5i4h0c13h-472.5h-84.5i0h5a8h-309.5h-207.5i0h3c9h-308.5h-78.5i0h1g0b9h-308.5h-78.5i0h1c10h-298.5h11.5i0h0g0b10h-298.5h11.5i0h0c11h-300.5h109.5i0h1g0b11h-300.5h109.5i0h1c12h-296.5h206.5i4h0g0e
https://boglweb.github.io/?q=f1h-201.1h-251.6i0h3c2h-201.1h-153.6i0h1c3h-202.1h-44.6i0h0c4h-202.1h60.4i0h1c5h-204.1h173.4i4h0c6h-588.1h-298.6i0h5a2h-201.1h-153.6i0h1c3h-202.1h-44.6i0h0g0b3h-202.1h-44.6i0h0c4h-202.1h60.4i0h1g0b4h-202.1h60.4i0h1c5h-204.1h173.4i4h0g0b1h-201.1h-251.6i0h3c2h-201.1h-153.6i0h1g0e
https://boglweb.github.io/?q=f1h-153.2h-134.7i0h3c2h-152.4h-47.2i0h1c3h-150.8h36.7i4h0c4h-151.6h123.0i0h1c6h-148.4h210.1i4h0c8h-470.1h-138.3i0h5a1h-153.2h-134.7i0h3c2h-152.4h-47.2i0h1g0b2h-152.4h-47.2i0h1c3h-150.8h36.7i4h0g0b3h-150.8h36.7i4h0c4h-151.6h123.0i0h1g0b4h-151.6h123.0i0h1c6h-148.4h210.1i4h0g0e
https://boglweb.github.io/?q=f1h-462.5h-275.5i0h3c2h-461.5h-185.5i0h1c3h-457.5h-94.5i0h0c4h-454.5h-3.5i0h1c5h-461.5h77.5i0h0c6h-340.5h-170.5i0h5a1h-462.5h-275.5i0h3c2h-461.5h-185.5i0h1g0b2h-461.5h-185.5i0h1c3h-457.5h-94.5i0h0g0b3h-457.5h-94.5i0h0c4h-454.5h-3.5i0h1g0b4h-454.5h-3.5i0h1c5h-461.5h77.5i0h0g0e
https://boglweb.github.io/?q=f3h-283.5h-239.5i0h3c4h-278.5h-136.5i0h1c5h-280.5h-47.5i4h0c6h-283.5h42.5i0h1c7h-278.5h148.5i4h0c8h-115.5h-118.5i0h5a3h-283.5h-239.5i0h3c4h-278.5h-136.5i0h1g0b4h-278.5h-136.5i0h1c5h-280.5h-47.5i4h0g0b5h-280.5h-47.5i4h0c6h-283.5h42.5i0h1g0b6h-283.5h42.5i0h1c7h-278.5h148.5i4h0g0e
https://boglweb.github.io/?q=f1h-421.5h-258.0i0h3c2h-422.5h-156.0i0h1c3h-432.5h-55.0i0h0c4h-432.5h38.0i0h1c5h-425.5h132.0i0h0c6h-321.5h-203.0i0h5a4h-432.5h38.0i0h1c5h-425.5h132.0i0h0g0b4h-432.5h38.0i0h1c3h-432.5h-55.0i0h0g0b3h-432.5h-55.0i0h0c2h-422.5h-156.0i0h1g0b2h-422.5h-156.0i0h1c1h-421.5h-258.0i0h3g0e
https://boglweb.github.io/?q=f1h-144.5h-134.5i0h3c2h-140.5h-46.5i0h1c3h-141.5h33.3i0h0c4h-151.4h122.1i0h1c5h-144.3h282.8i4h0c8h-414.3h-164.0i0h5a3h-141.5h33.3i0h0c4h-151.4h122.1i0h1g0b4h-

151.4h122.1i0h1c5h-144.3h282.8i4h0g0b2h-140.5h-46.5i0h1c3h-141.5h33.3i0h0g0b1h-144.5h-134.5i0h3c2h-140.5h-46.5i0h1g0e
https://boglweb.github.io/?q=f10h-283.5h-233.1i0h3c11h-288.5h-137.1i0h1c12h-286.5h-44.1i0h0c13h-290.5h50.9i0h1c14h-289.5h151.9i4h0c15h-552.5h-192.1i0h5a10h-283.5h-233.1i0h3c11h-288.5h-137.1i0h1g0b11h-288.5h-137.1i0h1c12h-286.5h-44.1i0h0g0b12h-286.5h-44.1i0h0c13h-290.5h50.9i0h1g0b13h-290.5h50.9i0h1c14h-289.5h151.9i4h0g0e
https://boglweb.github.io/?q=f1h-525.8h-369.3i0h3c2h-523.3h-260.8i0h1c3h-523.3h-148.3i0h0c4h-522.1h-41.5i0h1c5h-521.0h58.5i0h0c6h-782.4h-366.5i0h5a1h-525.8h-369.3i0h3c2h-523.3h-260.8i0h1g0b2h-523.3h-260.8i0h1c3h-523.3h-148.3i0h0g0b3h-523.3h-148.3i0h0c4h-522.1h-41.5i0h1g0b4h-522.1h-41.5i0h1c5h-521.0h58.5i0h0g0e

Q29 - Generate a .bogl file from the system diagram and open that file in BoGL Web a new tab. Check off all that apply to the system diagram created from that file, using the previous diagram as a reference.



#	Answer	%	Count
1	At least one component is missing.	0.00%	0
2	At least one bond is missing.	0.00%	0
3	At least one modifier is not set correctly.	12.50%	4
4	At least one extra component was added.	0.00%	0

5	At least one extra bond was added.	0.00%	0
6	This system diagram is identical to the previous one.	71.88%	23
7	Another issue was identified.	15.63%	5
	Total	100%	32

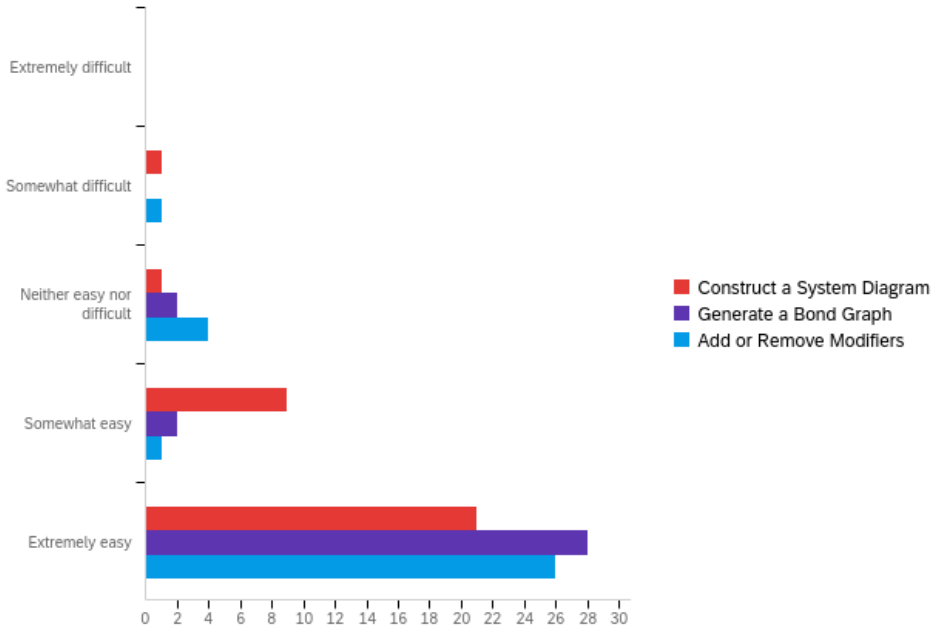
Q30 - Please upload the .bogl file you generated in the previous question.

All users submitted .bogl files.

Q32 - Please screenshot your system diagram and upload it.

All users submitted screenshots.

Q33 - How easy was each task?



#	Field	Minimum	Maximum	Mean	Std Deviation	Variance	Count
1	Construct a System Diagram	14.00	17.00	16.56	0.70	0.50	32
2	Generate a Bond Graph	15.00	17.00	16.81	0.53	0.28	32
3	Add or Remove Modifiers	14.00	17.00	16.63	0.82	0.67	32

#	Question	Extremely difficult		Somewhat difficult		Neither easy nor difficult		Somewhat easy		Extremely easy		Total
1	Construct a System Diagram	0.00%	0	3.13%	1	3.13%	1	28.13%	9	65.63%	21	32
2	Generate a Bond Graph	0.00%	0	0.00%	0	6.25%	2	6.25%	2	87.50%	28	32
3	Add or Remove Modifiers	0.00%	0	3.13%	1	12.50%	4	3.13%	1	81.25%	26	32

Appendix C – Glossary

Ant Design: A user interface framework created by Ant Financial, an affiliate company of the conglomerate Alibaba. The framework contains many commonly used user interface components, which can be easily accessed and used in a software project. (*Introduction - Ant Design*, n.d.)

Asynchronous: An operation which is asynchronous will occur independent of other operations that are occurring in the system (*Definition of Asynchronous | Dictionary.Com*, n.d.).

Asynchronous JavaScript and XML (AJAX): A development style where code is written such that actions occur asynchronously as to not disrupt the user experience (*What Is Ajax Programming - Explained - KeyCDN Support*, n.d.).

Azure: A cloud service created by Microsoft. It can be used to host a website, store data, and more (*Create Your Azure Free Account Today | Microsoft Azure*, n.d.).

Backend: All the code in an application with which the user does not directly interact (*Top Backend Developer Skills You Must Have (2023)*, 2023).

Bond Graph: A graphical technique that depicts the energy flow in a system and helps gather insight into a system's behavior (*About Bond Graphs*, n.d.).

Bootstrap: A toolkit, which is used for front end development and allow developers to build applications quickly (Thornton & Otto, n.d.).

Bulma: A free, open-source, CSS library built with Flexbox, a common CSS layout model, that allows you to easily change the theme of components with Syntactically Awesome Style Sheets (SASS), a CSS language extension. (*Bulma: Free, Open Source, and Modern CSS Framework Based on Flexbox*, n.d.)

Cascading Style Sheets (CSS): A language that specifies the presentation of documents on the web (*What Is CSS?,* 2023).

Compile Time: The point in time when a program is converted into a machine-readable form.

Client: Software downloaded from a server that receives data from the user, displays data, and requests data or services from outside sources as needed ("Client," 2022).

Data Structure: A format for organizing processing, storing, and retrieving data. They are commonly used in a software project (*What Are Data Structures?*, n.d.).

Dependency Injection: A technique used to help make classes within a software project independent of any dependencies. This is done by decoupling the usage and creation of an object (Janssen, 2018).

Deployment: The action of putting an application on a server so it can be accessed by clients looking to use the application.

Deserializer: A piece of code that takes some string of text, parses the text, and creates an object.

Design Pattern: Common ways to describe a system and discuss solutions to problems that occur often within a software project. They are used to help structure code in a way that is seen as clean and readable (*Design Patterns*, n.d.).

Dictionary: A data structure which maps one piece of data, a key, to another piece of data, a value.

Document Object Model (DOM): A representation of the structure of the website and hold all of the information needed to display a website to a user (*Introduction to the DOM - Web APIs | MDN*, 2023).

Extensible Markup Language (XML): A text-based format that can be used to represent structured information (*XML Essentials - W3C*, n.d.).

Factory Pattern: A design pattern that takes in object specifications from a client and returns an object matching those specifications using an interface, which is common to all instances of an object. This factory pattern hides object creation logic from the client. ("Factory Method for Designing Pattern," 2015)

Frontend: The part of an application with which the user interacts (*What Does a Front-End Developer Do?*, 2023).

Git: A version control tool, which provides a standard for how repositories are set up and accessed.

GitHub: A platform, which hosts repositories that can be accessed using Git.

GitHub Actions: A tool used to automate tasks for specific GitHub repositories.

GitHub Pages: A platform, which can be used to host static web pages.

Hypertext Transfer Protocol (HTTP): The protocol used to transfer web pages to a client (*What Is HTTP?*, n.d.).

Integrated Development Environment (IDE): A tool, which is used by developers to write code. These environments also include many tools that can be used to test, run, deploy, and refactor code.

Jira: A platform which is used to manage tasks. Tasks can be organized using tickets, which can be sorted into categories including To Do, In Progress, etc.

jQuery: A feature-rich JavaScript library, which can be used to aid in creating websites. jQuery helps developers create features including animation, event handling, etc. (js.foundation, n.d.).

Linked List: A data structure, which stores a list of elements, with each element connected to the next element in the list and sometimes the previous element in the list (*Linked List in a Data Structure*, n.d.).

Material Design: A set of guidelines, which are used to create good looking, practical user interfaces (*Material Design*, n.d.-b).

One-directional Binding: The ability to bind data from a UI component to the DOM or from the DOM to a UI component, but not both. (Goudar, 2019)

Parse: To extract the structure and meaning from a string of text so that it can be used by an application.

Progressive Web App (PWA): Web applications, which use web-platform features to give the user an experience like a native application (*Introduction to Progressive Web Apps - Progressive Web Apps (PWAs) | MDN*, n.d.).

Repository: A location where data or other information can be stored. This is often done in the cloud (*Definition of Repository | Dictionary.Com*, n.d.).

Rider: An IDE developed by JetBrains specifically designed for C# development (*JetBrains*, n.d.).

Runtime: Time during which the user runs the application.

Service worker: Part of an application that sits between the browser and the network. It is intended to catch outgoing web traffic and substitute information to allow for developers to allow for the website to be used when the user does not have an internet connection (*Service Worker API - Web APIs | MDN*, 2023).

Singleton: A design pattern used to ensure that only one instead of a class is ever created and used.

Stack: A data structure which follows the last in first out principle (*Implementing Stacks in Data Structures [Updated]*, n.d.).

State Equation: A differential equation derived from the flow and effort relationships in a bond graph which represents the values of those flows and efforts over time.

Substring: A continuous sequence of characters within a larger string.

System Diagram: A method of intuitively representing a mechanical or electrical network as a web of connected pictures of components including springs, resistors, masses, etc.

User Interface (UI): What a user uses to interact with an application (*What Is User Interface (UI)?*, n.d.).

Version Control: Software that enables tracking and managing of changes made within a project (Atlassian, n.d.-b).

Vim: A text editor which allows for a lot of customization and efficiency (*Welcome Home : Vim Online*, n.d.).

Visual Studio: An IDE which can be used to develop .NET and C++ applications. It additionally includes many tools to aid the developer when they are working on these applications (*Visual Studio*, n.d.).

Web Manifest File: Files that contain information about the application. This is needed for progressive web applications (*Web App Manifests | MDN*, 2023).

Appendix D – Graph Drawing Discussion

The graph drawing algorithm we implemented for BoGL Web was a force directed algorithm. We chose this algorithm since it works on graphs, which are non-planar, and only requires that the graph be connected. The implementation for this algorithm was inspired by (Kindermann, 2021). The rest of this section assumes a knowledge of a first course in linear algebra (which could be obtained from Linear Algebra Done Right (Axler, 2014)). The concepts behind the algorithms can be interpreted without this background, but the mathematical underpinnings will require this knowledge.

The idea of a force directed algorithm is to create some physical analog for a system and find an optimal solution by minimizing or maximizing the energy in the system. Eades presents springs as one physical analog we can use to generate graph layouts (Eades, 1984). To create a physical analogy, we consider each vertex to be a ring, and each edge to be a spring. The ends of the springs are connected to the rings and the rings are places somewhere in space. We can then let all the rings go and the system will reach a relaxed state, which will be visually appealing.

The graph drawing algorithm uses this concept: vertices will be placed somewhere in a coordinate plane and are then "let go", allowing the algorithm to modify the position of the vertices until they reach a minimum energy state. Our intent with this algorithm was to create a visually appealing embedding (drawing of the graph).

There are a few issues with the force directed algorithm that an improved system should address. The first is that the algorithm only finds a locally optimal solution. This means that the embedding created by the algorithm may have unnecessary edge crossings. Another issue with this drawing algorithm is that it is computationally slow. Each iteration of the algorithm takes

$\mathcal{O}(V^2)$, where V is the number of vertices in the graph, and the algorithm must run for some number of iterations, until it reaches some stopping criteria. This stopping criterion can either be a number of iterations, or until the maximum force computed in the graph is below some epsilon. It is hard to predict how fast it will converge to this solution and, in practice, on many different graph (including graphs other than bond graphs), this takes a lot of time to arrive at a solution (e.g., several seconds for a graph with six vertices and six edges).

The spectral formulation of the force directed layout algorithm uses properties of matrices relating to the graph to create embeddings of the graph. The most important matrix for creating these drawings is the Laplacian matrix. There are several different ideas behind using matrices to draw graphs. We will focus on an energy-based method using eigenprojection that is presented in (Koren, 2005). The basic idea of this method was inspired by the ideas of Eades and similar spring/energy approaches. We use the eigenvalues of either the adjacency matrix or the Laplacian matrix. For regular graphs, previous studies have shown that the choice of using the adjacency matrix, or the Laplacian does not affect the drawing since the eigenvectors are equal. However, for non-regular graphs it has been shown that the Laplacian matrix produced better drawings.

To explain this algorithm, we must first discuss some notations. We consider a graph $G = (V, E)$ where V is the vertex set and E is the edge set. We have a p-dimensional layout defined by p vectors

$$x^1, \dots, x^p \in \mathbb{R}^n$$

where,

$$x^1(i), \dots, x^p(i)$$

are the coordinates of node i and $\mathbb{R}^n$ is an *n*-dimensional vector in the reals. We have a distance d_{ij} between vertices i and j defined by

$$d_{ij} = \sqrt{\sum_{k=1}^{p} \left(x^k(i) - x^k(j)\right)^2}$$

We use the adjacency matrix A is defined as

$$A_{ij} = \begin{cases} 0, & \langle i,j \rangle \notin E \\ 1, & \langle i,j \rangle \in E \end{cases}$$

and the Laplacian matrix L defined as

$$L_{ij} = \begin{cases} \deg(i), & i = j \\ -1, & \langle i,j \rangle \in E \\ 0 & \text{otherwise} \end{cases}$$

Where $\deg(i)$ is the degree of the i^{th} vertex.

There are several important properties of the Laplacian matrix, which are important to graph drawing:

- L is a real symmetric and hence, its n eigenvalues are real, and its eigenvectors are orthogonal.
- L is positive semidefinite and hence, all eigenvalues of L are nonnegative.
- $1_n \stackrel{\text{def}}{=} (1,1,\ldots,1) \in \mathbb{R}^n$ is an eigenvector of L, with associated eigenvalue 0.
- The multiplicity of the zero eigenvalue is equal to the number of connected components of a graph G. If G is connected, then, 1_n is the only eigenvector associated with the eigenvalue 0.

We next present a lemma, theorem, and corollary, which will be needed for creating the spectral layout. Proofs for these can be found in (Koren, 2005).

Lemma 1. Let L be an $n \, x \, n$ Laplacian matrix and let $x \in R^n$. Then,

$$x^\top Lx = \sum_{i,j \in E} w_{ij}\big(x(i) - x(j)\big)^2$$

More generally, for p vectors $x^1, \dots, x^p \in \mathbb{R}^n$, we have

$$\sum_{k=1}^{p} (x^k)^\top Lx^k = \sum_{i,j \in E} w_{ij} d_{ij}^2.$$

This lemma is used to show a relationship between the distance between vertices and the Laplacian matrix. We will later use the lemma to reform the minimization problem using the Laplacian matrix. Next, we present an important theorem, which will provide solutions to the minimization formulation of this problem.

Theorem 1.

Given a symmetric matrix $A_{n\times n}$, denote by $v^1, \dots, v^n$ its eigenvectors, with corresponding eigenvalues $\lambda_1 \le \cdots \le \lambda_n$. Then, $v^1, \dots, v^p$ are an optimal solution of the constrained minimization problem,

$$\min_{x^1,\dots,x^p} \sum_{k=1}^{p} (x^k)^\top Ax^k,$$

$$subject\ to\colon\ (x^k)^\top x^l = \delta_{kl},\ k, l = 1, \dots, p.$$

Where δ_{kl} is the Kronecker Delta. We assume that $p < n$.

This theorem will later be used to provide solutions to our minimization problem. We will be able to form the minimization problem as it is here, and then use the eigenvectors as solutions.

Corollary 1.

Given a symmetric matrix $A_{n\times n}$ and a positive definite matrix $B_{n\times n}$, denoted by $v^1, \dots, v^n$, the generalized eigenvectors of (A, B), with corresponding eigenvalues $\lambda_1 \le \cdots \le \lambda_n$, then $v^{k+1}, \dots, v^{k+p}$ are an optimal solution of the constrained minimization problem,

$$\min_{x^1,\ldots,x^p} \frac{\sum_{i=1}^{p}(x^i)^\top A x^i}{\sum_{\mathrm{i}=1}^{p}(x^i)^\top B x^i}$$

$$subject\ to\colon\ (x^1)^\top B x^1 \ = \ (x^2)^\top B x^2 \ = \cdots = \ (x^p)^\top B x^p$$

$$\left(x^i\right)^\top B x^j \ = \ 0, 1 \leq \ \mathrm{i} \neq \ \mathrm{j} \leq \mathrm{p}$$

$$\left(x^i\right)^\top B v^j \ = \ 0, \ \ i = 1, \ldots, p, \ j \ = \ 1, \ldots, k.$$

Like the previous theorem, we can use this corollary to show that solutions to the minimization problem are the eigenvectors of the Laplacian matrix.

We can now set up a minimization problem, which will be used to create a drawing.

$$\min_{x^1,\ldots,x^p} E(x^1, \ldots, x^p) \stackrel{\text{def}}{=} \frac{\sum w_{ij} d_{ij}^2}{\sum_{i<j} d_{ij}^2}$$

$$subject\ to\colon\ \mathrm{Var}(x^1) = \mathrm{Var}(x^2) = \cdots = \mathrm{Var}(x^p)$$

$$\mathrm{Cov}(x^k, \ x^l) = 0, \ 1 \leq k \neq \ l \leq p$$

We define $\mathrm{Var}(x)$ as the variance of x defined as:

$$\mathrm{Var}(x) = \frac{1}{n}\sum_{i=1}^{n}(x(i) - \bar{x})^2$$

where $\bar{x}$ is the mean of x. We also define $\mathrm{Cov}(x^k, x^l)$ as the covariance of x^k and x^l defined as:

$$\frac{1}{n}\sum_{i=1}^{n}\left(x^k(i) - \overline{x^k}\right)\left(x^l(i) - \overline{x^l}\right)$$

For this problem, we rewrite d_{ij} as:

$$d_{ij}^2 = \sum_{k=1}^{p}\left(x^k(i) - x^k(j)\right)^2$$

The goal of this minimization problem is to draw the graph such that the edge lengths are short, but vertices are not crowded together. This is like what the Eades force-directed algorithm

does. Since the sum in the numerator incorporates edge weights, edges with many neighbors will have smaller edge lengths. The constraint requiring that all variances be equal makes it, so that vertices are scattered equally along each axis of the layout. This strives to make the graph symmetric, and "balances" the placement of the vertices. The second constraint requiring the covariance to be zero limits the correlation between axes, which makes it so that each additional axis has as much information as possible.

This problem is invariant under translation meaning that it can be moved anywhere in a plane and still produces a similar result. Koren eliminates the degree of freedom by requiring that the average x- and y-coordinates of the vertices be 0 (Koren, 2005). This can be framed mathematically as

$$\sum_{i=1}^{n} x^k(i) = (x^k)^\top \cdot 1_n = 0$$

for each $1 \leq k \leq p$. This also makes it, so the covariance constraint is equivalent to requiring that the vectors be pairwise orthogonal $(x^k)^\top x^l = 0$. We also now have that $\mathrm{Var}(x^k) = \frac{1}{n}(x^k)^\top x^k$, which allows the uniform variance constraint to be rewritten as

$$(x^1)^\top x^1 = (x^2)^\top x^2 = \cdots = (x^p)^\top x^p.$$

Using Lemma 1, we can begin to reform the minimization problem using the Laplacian. We can replace $\sum_{i,j\in E} w_{ij} d_{ij}^2$ with the sum,

$$\sum_{k=1}^{p} (x^k)^\top L x^k.$$

This leads us to the equivalent minimization problem.

$$\min_{x^1,\ldots,x^p} \frac{\sum_{k=1}^{p} (x^k)^\top L x^k}{\sum_{i<j} d_{ij}^2}$$

$$subject\ to: (x^k)^\top (x^l) = \delta_{kl},\ k, l = 1, \dots, p$$

$$(x^k)^\top \cdot 1_n = 0,\ k = 1, \dots, p.$$

The next step will be to rewrite the denominator of the minimization problem. To do this we first present the following lemma.

Lemma 2.

Let $x \in R^n$ such that $x^\top 1_n = 0$, then,

$$\sum_{i<j} \big(x(i) - x(j)\big)^2 = n \sum_{i=1}^{n} x(i)^2 (= n \cdot x^\top x).$$

We want to show $\Sigma_{i<j} \big(x(i) - x(j)\big)^2 = n \cdot \Sigma_{i=1}^{n} x(i)^2$. The proof for this is direct. We start with

$$\sum_{i<j} \big(x(i) - x(j)\big)^2 = \frac{1}{2} \sum_{i,j=1}^{n} \big(x(i) - x(j)\big)^2,$$

which is obtained using properties of sums. We can now expand the binomial inside the sum to obtain,

$$\frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} x(i)^2 - 2\big(x(i)x(j)\big) + x(j)^2$$

We are then able to distribute the sums across the inside terms to get,

$$\frac{1}{2} \left[\sum_{i=1}^{n} \sum_{j=1}^{n} x(i)^2 - 2 \sum_{i=1}^{n} \sum_{j=1}^{n} x(i)x(j) + \sum_{i=1}^{n} \sum_{j=1}^{n} x(j)^2 \right].$$

We can then rewrite these sums as follows,

$$\frac{1}{2} \left[\left[\sum_{i=1}^{n} x(i)^2 \right] \left[\sum_{j=1}^{n} 1 \right] - 2 \sum_{i=1}^{n} \sum_{j=1}^{n} x(i)x(j) + \left[\sum_{j=1}^{n} x(j)^2 \right] \left[\sum_{i=1}^{n} 1 \right] \right]$$

allowing us to reduce two of the terms to just be one sum.

$$\frac{1}{2}\left[n\sum_{i=1}^{n}x(i)^2-2\sum_{i=1}^{n}\sum_{j=1}^{n}x(i)x(j)+n\sum_{j=1}^{n}x(j)^2\right]$$

Since $x(i)^2$ and $x(j)^2$ are both just positions with relation to the variable being used to index the sum, we can combine both sums to simplify the expression,

$$\frac{1}{2}\left[2n\sum_{i=1}^{n}x(i)^2-2\sum_{i=1}^{n}\sum_{j=1}^{n}x(i)x(j)\right].$$

We can finally simplify the expression by distributing the $\frac{1}{2}$ term to get,

$$n\sum_{i=1}^{n}x(i)^2-\sum_{i=1}^{n}\sum_{j=1}^{n}x(i)x(j).$$

Since x is centered at 0, $\Sigma_{j=1}^{n}=0$ allowing us to further simplify this expression to

$$n\sum_{i=1}^{n}x(i)^2.$$

This is the expression that we wanted to show.

We are now able to rewrite $\Sigma_{i<j}d_{ij}^2$ as follows. We first can say,

$$\sum_{i<j}d_{ij}^2=\sum_{i<j}\sum_{k=1}^{p}\left(x^k(i)-x^k(j)\right)^2$$

since we previously defined d_{ij} as

$$d_{ij}=\sqrt{\sum_{k=1}^{p}\left(x^k(i)-x^k(j)\right)^2}$$

We can then switch the order of the sums and get,

$$\sum_{k=1}^{p}\sum_{i<j}\left(x^k(i)-x^k(j)\right)^2.$$

Lastly, we can use Lemma 2 to get,

$$\sum_{k=1}^{p} n \cdot (x^k)^\top x^k.$$

We are now able to rewrite the bottom of the above minimization problem as

$$\min_{x^1,\ldots,x^p} \frac{\sum_{k=1}^{p} (x^k)^\top L x^k}{\sum_{k=1}^{p} (x^k)^\top x^k}$$

$$subject\ to\colon\ (x^k)^\top\ (x^l) = \delta_{kl}, k, l\ =\ 1, \ldots, p$$

$$(x^k)^\top \cdot\ 1_n\ =\ 0,\ k = 1, \ldots, p.$$

Using Corollary 1, we can say that the solution to this optimization problem will be the lowest positive eigenvectors. We can use these vectors as coordinates to obtain a layout.

We conducted several tests using this algorithm to establish whether the spectral layout formulation would be faster than the force directed method. These tests produce reasonable results for smaller bond graphs with three vertices, however, the layouts were no longer visually appealing as the size of the bond graphs grew. We present an example of a system diagram and its corresponding bond graph which the layout algorithm did not work well on. This system diagram can be seen below in Figure 120Figure 120: The System Diagram, which we used to Conduct Experiments..

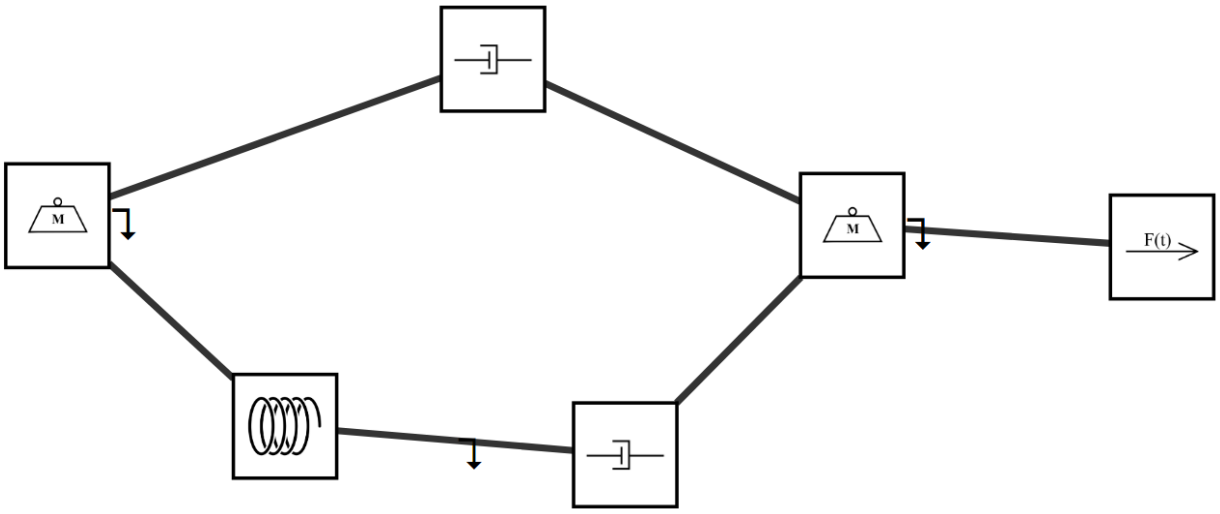


Figure 120: The System Diagram, which we used to Conduct Experiments.

This system diagram produces a bond graph with the structure in Figure 121.

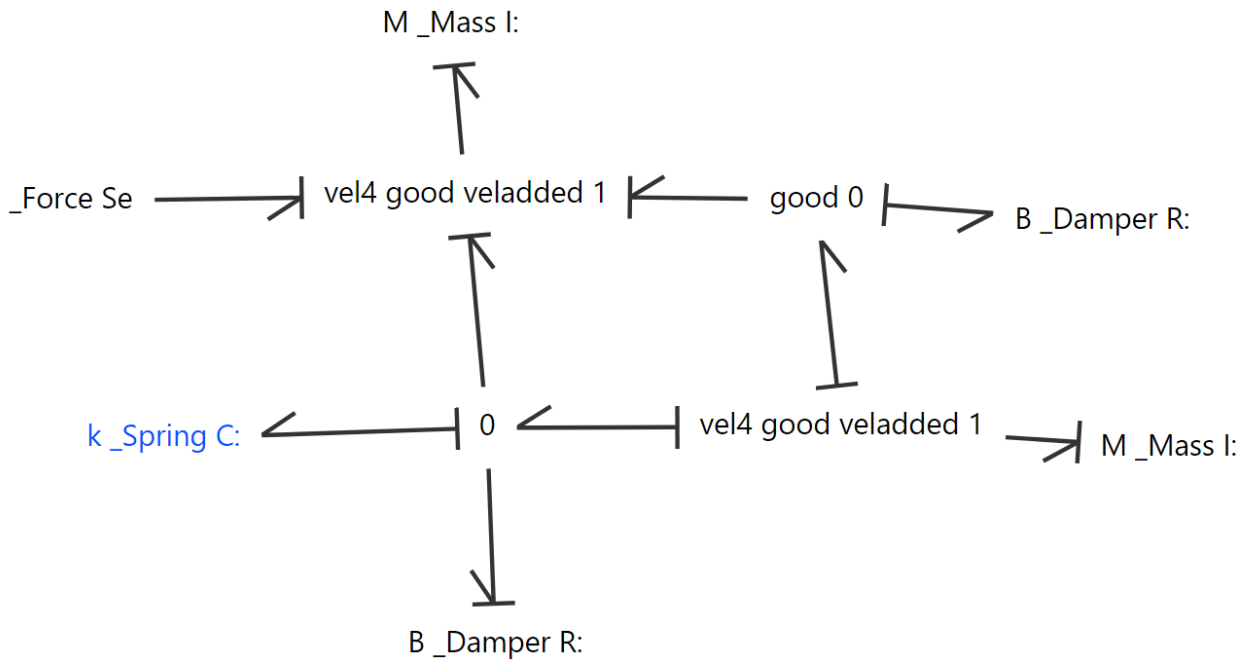


Figure 121: The Bond Graph, which was Generated by the System Diagram.

From this bond graph, we can construct the following Laplacian Matrix. This is done by creating some ordering of the vertices, placing the degrees of the vertices on the diagonal, and a -1 where there is an edge between two vertices.

$$L = \begin{bmatrix} 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & -1 & 4 & -1 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 3 & -1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 3 & -1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & -1 & 0 & 4 & -1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 \end{bmatrix}$$

This matrix will have ten eigenvalues, and ten associated eigenvectors. We can take the lowest two positive eigenvalues and their associated eigenvectors to be the following:

$$\lambda_1 \approx 0.485$$

$$v_1 \approx (-1 \quad -1 \quad -0.515 \quad -0.327 \quad -0.635 \quad 0.327 \quad 0.635 \quad 0.515 \quad 1 \quad 1)^{\mathsf{T}}$$

$$\lambda_2 \approx 0.519$$

$$v_2 \approx (1 \quad 1 \quad 0.481 \quad -0.806 \quad -1.675 \quad -0.806 \quad -1.675 \quad 0.481 \quad 1 \quad 1)^{\mathsf{T}}$$

It is clear from these vectors that if we constructed a layout using these vectors, we would have two pairs of elements, which would overlap since we can see that v_1 has two -1s, which line up with two ones in v_2. This would obviously be a poor layout as it would be difficult for the user to see elements that are on top of each other. Due to this issue, we did not investigate this algorithm further.

Koren's paper provides several other algorithms, which could be investigated (Koren, 2005). These algorithms are more complex, which is why we do not investigate them. There are many other algorithms, which can be used to draw graphs. Further exploration is required to find a layout algorithm that works best for bond graphs.