April 2019

Correcting Errors and Adding Features to PMKS

David Simon Richardson
Worcester Polytechnic Institute

Mitchell Thomas Farren
Worcester Polytechnic Institute

Zhihao Xie
Worcester Polytechnic Institute

Correcting Errors and Adding Features to PMKS

Major Qualifying Project

A report submitted to the Faculty of the WORCESTER POLYTECHNIC INSTITUTE in partial fulfillment of the requirements for the Degree of Bachelor of Science.

Written by:

Mitchell Farren (CS), David Richardson (ME), Zhihao Xie (RBE)

Advisors:

Professor D. Brown (CS)

Professor P. Radhakrishnan (ME)

DATE:APRIL 26, 2019

Abstract

Planar Mechanism Kinematic Simulator (PMKS) is a web-based tool to generate the kinematics of linkages. The tool can also be used to generate joint forces and input torque in linkages with revolute joints. The team explored and formulated various additions to the tool. The additions involved implementing methods to detect singularities in four and six-bar linkages with revolute joints. The team also worked on generating interface concepts for displaying free body diagrams and simulator results to enhance student experience with the tool. Finally, aspects to the migration of the tool from the current Microsoft Silverlight platform to the Microsoft Windows Desktop were also investigated.

Acknowledgments

We would like to express our thanks to our advisors, Prof. Pradeep Radhakrishnan and Prof. David Brown for their feedback and guidance throughout the whole process. We would also like to give our gratitude to the past MQP teams that have put countless hours into PMKS to be like what it is.

Table of Contents

Authorship

Chapter 1 Introduction	
1.1: Background on PMKS	Mitchell Farren
1.2: Problem Statement	Mitchell Farren
1.3: Structure of Paper	Mitchell Farren
Chapter 2: Literature Review	
2.5: Singularity	Zhihao Xie
2.6: Silverlight	Mitchell Farren
2.7: Windows Presentation Foundation	Mitchell Farren
2.8: Universal Windows Platform	Mitchell Farren
2.9: PMKS Code Organization	Mitchell Farren
2.10: PMKS Design of Animation Engine	Mitchell Farren
2.11 Comparison of Displaying Data Between PMKS and Other Software	David Richardson
2.12: Link Creation in PMKS Compared to SolidWorks and Creo	David Richardson
2.13: Force Creation in PMKS Compared to SolidWorks and Creo	David Richardson
2.14: Methods for analysis in PMKS compared to SolidWorks and Creo	David Richardson
Chapter 3: Methodology	
3.6: Having PMKS Determine Singularity in a four-bar and six-bar	Zhihao Xie
3.7: Work on the PMKS Software	Mitchell Farren
3.8: Maximizing the UI Experience	David Richardson
Chapter 4: Implementation	
4.4: Code foundation behind Singularity Check	Zhihao Xie
4.5: Work on the PMKS Software	Mitchell Farren
4.6: Maximizing the UI experience	David Richardson

Chapter 5: Discussion	
5.1 Accomplishments	Mitchell Farren, David Richardson
5.2 Problems Faced	David Richardson
5.3 What Still Needs Work	Mitchell Farren, David Richardson
5.4: Singularity Check	Zhihao Xie
5.5: State of the Software Now and Future Insight	Mitchell Farren
5.6: User Experience	David Richardson
Chapter 6: Conclusion	
6.1: Accomplishments	Mitchell Farren, David Richardson
6.2: Major Roadblocks Faced	Mitchell Farren, David Richardson
6.3: What was Learned	David Richardson

Chapter 1 Introduction

1.1: Background on PMKS

The Planar Mechanism Kinematics Simulator (PMKS), developed by Prof. Matthew I Campbell, Professor, Mechanical Engineering, Oregon State University is an algorithmic tool that is intended to generate a variety of static and dynamic data. It models mechanical data from planar linkages such as joint forces, input torque, velocities, accelerations, mass moments of inertias, instantaneous centers, and stress at joints and links.

PMKS is currently being used by students in advanced mechanical engineering classes, as a tool to help learn different analyses on linkages and as a way to check their calculations. As an academic tool, it brings a lot of fundamental mechanical information to the user in one click. PMKS has evolved throughout years by adding features from successive Major Qualify Projects (MQP) at Worcester Polytechnic Institute (WPI). More information on PMKS, including access to the Silverlight version, can be found at https://designengrlab.github.io/PMKS/.

1.2: Problem Statement

When we first got this project, our advisor Professor Pradeep originally wanted PMKS to implement link synthesis. As we got into our project, we quickly realized the scope needed to be more focused. As we started work on the project, our team was spread thin amongst a number of different tasks. As we continued, the complexity of the software became apparent, and our goals shifted towards adding features and correcting errors on PMKS instead of renovating the software to achieve link synthesis. To tackle this project, our team divided up into two teams: the ME and CS team.

1.2.1: Mechanical Engineering Team

The ME team focused on one task:

- integrating linkage singularity detection

1.2.2: Computer Science Team

The CS team focused on three tasks:

- attempted to get the new desktop application running properly

- looked into creating a better design improvement of the UI
- create diagrams and documentation were developed to expedite the process of learning how PMKS works for future project teams

1.3: Structure of Paper

This paper is organized as follows. Chapter 2 presents some background information that is needed to more comprehensively understand our project. Chapter 3 presents the methodology, which is a collection of plans that was followed throughout this project. That is followed by Chapter 4 which contains our implementation section. This section consists of the work we did during the project and how we did it. Chapter 5 consists of a discussion section, where we discuss the project, how it went, what we accomplished, and where the project is now. The final chapter, Chapter 6, is our conclusion, where we reiterate our goal and talk about our achievements and their importance.

Chapter 2: Literature Review

2.1: Singularity

Singular positions are certain states of a linkage mechanism at which prediction of future motion is impossible. For example, in **Figure 2.1**, it is possible for the follower link to rotate in either clockwise or counterclockwise direction, which makes its successive state of movement unpredictable. Singularity restricts the mobility of a linkage and is therefore generally unfavorable (in operation of robot arms, for instance, approaching a singular position can cause extra load to the arm). Nevertheless, there exist some cases in which singularity is utilized for specific purposes (e.g. toggle position of a truck tailgate). PMKS and most other linkage simulators currently do not offer the functionality of singularity detection. Some software (e.g. SolidWorks) notify the presence of movement constraints, but it is not an accurate indication of singularity.

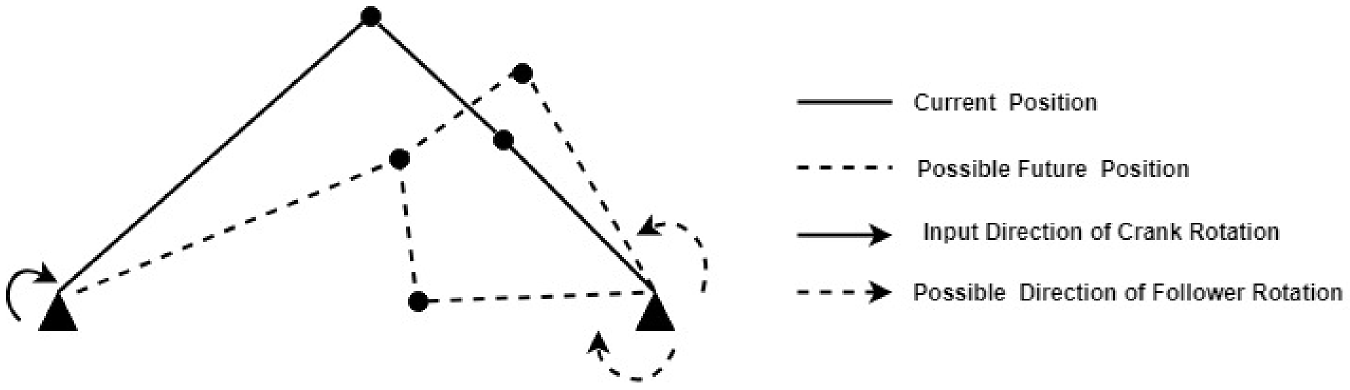

Current Position
Possible Future Position
Input Direction of Crank Rotation
Possible Direction of Follower Rotation

Figure 2.1: Internal uncertainty of singular position

As one of its main functionalities, PMKS uses matrix equation to carry out static and dynamic analysis of linkages, but matrix equation is not solvable for linkages at singular positions, at which the coefficient matrix is singular. This causes the program to crash when it tries to conduct analysis for any linkage with a singular position, since PMKS analyses all possible positions of a linkage within its range of movement.

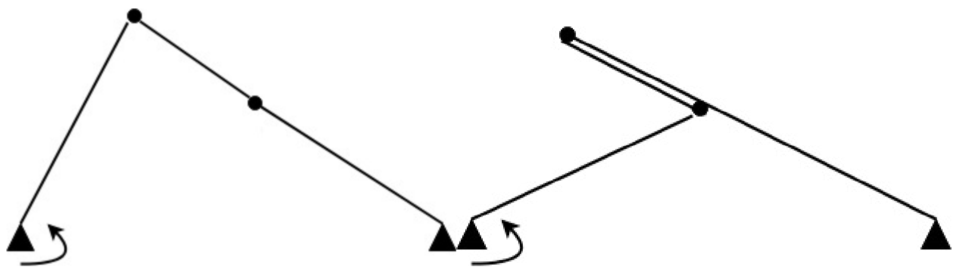


Figure 2.2: Examples of singular position in four-bar mechanism

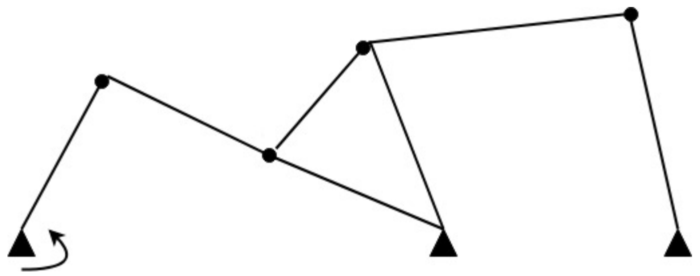


Figure 2.3: Example of singular position in six-bar mechanism

A four-bar linkage is at its singular position when its coupler aligns with follower (**Figure 2.2**). Six-bar mechanism, on the other hand, can be seen as two four-bars connected side by side ---- if either of the two four-bar sub-mechanisms is at singular position, the whole six-bar linkage will also be under a singular configuration (**Figure 2.3**). In order to avoid program failure during analysis of linkages with singularity, some means of identifying singular positions needs to be implemented.

2.2: Silverlight

Silverlight is a development tool created by Microsoft used for creating interactive user experiences for Web and mobile applications. It is powered by the .NET framework and compatible with multiple browsers, devices and operating systems.
Silverlight is no longer supported by Microsoft, as the only browsers that support it are those the Internet Explorer browser family. This lack of support by modern browsers has led to the discontinued use of Silverlight to make new applications. Silverlight applications are based on a client-server architecture, with a back-end system that talks to a light weight front-end.

2.3: Windows Presentation Foundation

Windows Presentation Foundation (WPF) is a solution that allows a user to develop applications on computers running Windows operating systems. It provides classes and a general structure to desktop application design, allowing for code to be maintainable and uniform.
WPF supports many more features and has more functionality than Silverlight. WPF is a more mature tool, providing enhanced animation functionality as well as a larger user base. It is primarily focused on the user interface portions of applications designed on the .NET framework. It has evolved from WinForms, which is more primitive front-end solution for building applications. Windows applications use WPF to keep application development consistent and to make them consistent with other applications in terms of their structure, dependencies, and format.

WPF is used to make web, desktop, and other kinds of applications across the Microsoft application development ecosystem.

2.4: Universal Windows Platform

The Universal Windows Platform (UWP) is a common application platform that developers can use to build applications that run on Windows 10 systems. Microsoft is favoring development using UWP over other, less universal development solutions. It allows for polished, professional applications to be built and run efficiently on Windows 10 devices.

By moving towards a universal platform, it will build the developer community and unite what would have been a collection of mostly unrelated software built on different systems.

2.5: PMKS Code Organization

PMKS is currently a Visual Studio Solution that contains a couple different coding project files. There is one project file that handles the linkage analyses, and two project files that handle front-end user interface code. The first one is a Silverlight web application front-end, while the second is a WPF desktop application front-end. Both project files that have front-end code reference the back-end code that handles linkage analyses.

In the WPF desktop application project file, the class MainPage is where most of the logic is in the code. This contains all code that handles animation, data input, data output, and user notifications.

2.6: PMKS Design of Animation Engine

PMKS handles animations in a couple different steps. The user first inputs their linkage data into the input matrix on the top left of the PMKS application window. This data is stored in the code handling the front end of the system. The data is then banded to variables in the back-end of the system, which uses the user-created linkage data to perform analyses on the linkage. These analyses are then available to the front-end so resultant forces and other information can be displayed to the user.

2.7 Comparison of Displaying Data Between PMKS and Other Software

PMKS displays very little data in its animation almost all data needs to be requested and saved than sorted through. Compared to systems like SolidWorks and Creo, which typically open windows with data in them for the user to save or immediately use. This makes data acquisition take longer for

PMKS than in programs like SolidWorks and Creo. This issue needs and can be resolved through various methods like free body diagrams and charting functionalities. PMKS creates all the data that the user could need to analyze a linkage. To create these functions is really just a question of how to ask for this data and which methods are best to display this data.

2.8: Link Creation in PMKS Compared to SolidWorks and Creo

PMKS is made for creating linkages so by limiting the creation options it is easier to make linkages in PMKS than in SolidWorks and Creo. SolidWorks and Creo are capable of making almost any physical object or system so they require more dimensions and inputs than PMKS does. PMKS only asks for the distance between joints to create a link, as well it only requires the type of joint and possibly its angle to create a joint. This vastly shortens the time needed to create a system when PMKS is compared to more complex software like SolidWorks and Creo.

2.9: Force Creation in PMKS Compared to SolidWorks and Creo

Applying a force in PMKS is a difficult process, however it is still is less demanding than more powerful software like SolidWorks and Creo. These programs require the user to set up simulations before allowing them to apply forces. Than to create forces the user needs to identify the area the forces will be applied the direction of the forces in three dimensions and the magnitude of the forces. This can only work properly if the assembly has been correctly constrained or the simulation will have conflicts and create no data or incorrect data for the user. Unlike PMKS which makes assumption for the user and simplifies forces creation and animation.

2.10: Methods for analysis in PMKS compared to SolidWorks and Creo

Like previously stated in section (2.8) the main difference between PMKS and software like SolidWorks and Creo is that the wide scope of SolidWorks and Creo. This makes the user need to specify more dimensions and characteristics to make a part or test that part. What makes PMKS so useful is its ability to infer from the user’s inputs and create parts with only a few dimensions. In terms of analysis methods PMKS can do just about all the analysis, for a linkage, that SolidWorks or Creo can do the difference in PMKS is that it makes assumptions and is capable of creating usable data faster than SolidWorks and Creo. Where SolidWorks and Creo make very few assumptions and require the user’s input instead. Because PMKS is so focused on its linkage capabilities' it can work faster, however SolidWorks and Creo are more specialized in custom or complex systems.

Chapter 3: Methodology

This MQP focused on improving PMKS based on calculating analysis. Throughout the year, there were three objectives that were worked on:

1. Having PMKS determine singularity in a 4-bar and 6-bar.
2. Debugging the desktop application and developing diagrams and documentation.
3. Maximizing the UI experience.

While focusing on these objectives, we would certain on certain applications to validate the analysis. MATLAB code and Working Model replicas would be created in which certain analysis would be determined.

3.1: Having PMKS Determine Singularity in a four-bar and six-bar

In order to avoid program failure during analysis of linkages that have singular positions, PMKS needs to be able to tell whether a linkage created by user will have any singular position or not, so that if singularity is present (which is not a favorable design trait in many cases), the user can modify the linkage or inform the program on how should it treat singular positions. This can be achieved based on the relationship between length of links within the mechanism.

3.1.1: Triangular-Check Method

As shown in **Figure 3.1**, when a four-bar is at its singular position, the links (or rather, the lines between joints on the same link) enclose a triangular shape. Specifically, segment AB (length of crank), segment AD (length of ground link) and segment BD (the sum of lengths of coupler and follower, or the difference between them) form a triangle. Alternatively, they can also completely overlap into a single segment, as is the case of a parallel four-bar linkage (**Figure 3.2**). Inversely, in a given four-bar linkage, if its segments AB, AD and BD satisfy this relationship, it will reach at least one singular position during its cycle of movement.

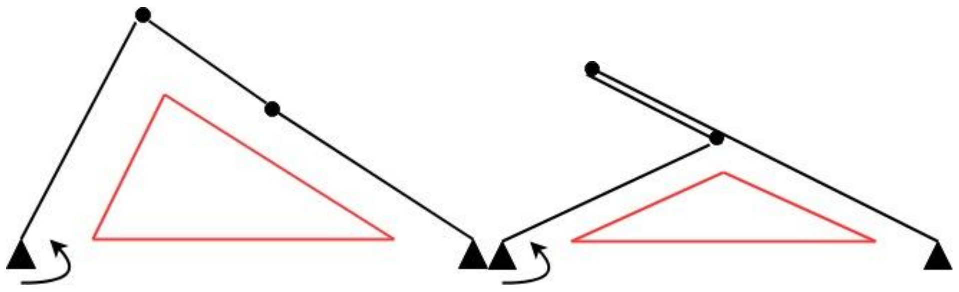


Figure 3.1: Triangle Shape Enclosed by Links in Four-Bars at their Singular Positions

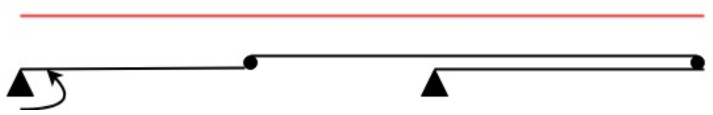


Figure 3.2: Singular position of parallel 4-bar linkage, in which all links overlap into one single line

For three segments to be able to form a triangle or completely overlap, they must satisfy the following condition:

$Longest\ segment\ <=\ Sum\ of\ the\ other\ two$

(Equation 8)

3.1.2: Why not simply check if coupler and follower have the same slope during analysis?

This alternative approach is easier to implement but has not been used for two main reasons:

1) The checking process takes place during linkage analysis, which means the user cannot know if the linkage has singular position or not before analysis and take corresponding actions to improve the design.
2) PMKS analyses the positions of linkage at a user-specified angle increment, which is 1 degree by default. Since singular position only exists when coupler exactly aligns with follower, it is possible for PMKS to miss the exact singular position and fail to detect it, which makes the user unable to know if the linkage actually has any singular position.

3.2: Work on the PMKS Software

Our initial goal for the programming side of the project was to debug the desktop version of PMKS. This section goes over the methodology for this.

3.2.1: Background on PMKS as an Application

PMKS is programmed in Silverlight, and it is currently deployed as a web application. PMKS users access the application through the use of a web browser. Silverlight is a Microsoft product, so Internet Explorer is needed to properly use the application. At the start of PMKS, Microsoft was still supporting Silverlight. Now, they have dropped support, and have instead opted to support desktop application building frameworks. As a consequence, our goal was to repurpose the old Silverlight code to create a desktop version of PMKS with the same functionality as the Silverlight web application version.

The Windows Presentation Foundation (WPF) is one of the frameworks Microsoft currently supports, and it has similar organization and function to Silverlight. Because of these similarities, much of the code in the Silverlight version of PMKS can be copied over to a new project that is using WPF, and many things continue to work even though Silverlight is not in the new project. Prior to this project, this conversion from Silverlight to WPF was begun. Our project began with this preliminary conversion of PMKS that did not run and crashed when the user clicked anywhere in the application after it started up.

3.2.2: Comparing the WPF Application with the Silverlight Application

We started with a broken desktop version of the application as well as a working web application version. To start, the team took the broken desktop version and compared it with the working web application that was being run locally in Visual Studio. The PMKS software is a large piece of software, so the first phase of the project was onboarding. In this phase, the team tried to understand the code and what is doing what. This includes investigating the working code to see how it works and then try and understand the design of the new WPF project.

3.2.3: Debugging and Error Correction

The second phase of the project was to get PMKS fixed and back to how it used to work in the web application, but now in the desktop application. This will be done by comparing WPF with Silverlight to understand the differences in the way they provide functionality to the developer, and the fixing logical errors in the software architecture.

3.2.4: Design Analysis of WPF Application

The third phase was an analysis of the new PMKS WPF software. The deliverable of this was to develop onboarding documentation to expedite the process of getting a new team up to date on PMKS, as that was the most difficult part of this project. This onboarding documentation includes information contained in this paper as well as diagrams, which are contained in Chapter 4. In addition, relevant information regarding WPF and Silverlight and the differences between the two will be synthesized and included (see **Appendix A**).

3.3: Maximizing the UI Experience

The user experience was not a priority to the creators of PMKS. The creators simply needed a workhorse to do complex calculations fast and accurately. As a result of the creators' focus on creating a workhorse they subsequently neglected the user interface which led to complex actions for the users. However, now the user experience should be improved because there is the time and manpower to do so. In order to make the appropriate updates to the user experience the program issues had to be identified. Once we had a strong understanding of the problems, they could be focused on and solved. Moreover, there was a focus on simplifying action even for those who already knew the program.

3.3.1: Force Entry, Cyclical Loading

Entering forces is a very convoluted process within PMKS. The user is required to know which force they wanted and where to apply it. Then in order to apply the force they had to know the global coordinates for the position within the link at the starting position of the animation. This is particularly difficult and time consuming because the width of the link is not factored in, the force is on the infinitely small line the link is projected around or it does not exist to the calculations. The first step in fixing this problem was to determine what data PMKS relied on to calculate the forces and other ways to input that data. The data required was the origin of the force within a link, the magnitude of the force, and the direction of the force. Through an iterative process with the project's advisors' new methods were developed that allowed the user to apply forces on links without relying on the global coordinates while still having the capability to use the global coordinates. Once this method was determined an addition of "Cyclical Loading" was discussed. Cyclical Loading would allow the user to determine when, in both the animation and calculations, the force would be applied. This was created to mimic the linkage moving an object in a repetitive process for instance from conveyor belt to conveyor belt. This was brought up because of the prevalence of linkages in repetitive processes that require moving an object to a new place or change the object's orientation.

3.3.2: Link Creation

Creating links is often the first task a user will do, and it is an overly complex process in PMKS. The complexity comes as a result of the table, called Edit Joints used to input links and joint data. The first column is populated with two link names separated by a comma then the rest of the information in that row is related to the connection between those two links. The rest of the information in that row is the start of the second link that is listed in the first column. The overlap in information, or rather its meaning, is a large source of confusion for the user. The chart confuses many users especially those with little to no experiences with the program. To combat these issues the plan was to add a column that would contain single link names and then the next column would be the joints attached to the link. This way the links and joints would have names and there would be a visual representation of the hierarchy of the order of connection, from link to joint to the next link.

3.3.3: Free Body Diagrams

After the difficulties with inputting new links, joints and forces were addressed, additions to PMKS were discussed. The intent of these changes was to give the users more information without requiring them to download and sort through all of force and position data that PMKS exports. The first addition worked on was a "Free Body Diagram" or (FBD) capability. This functionality would allow the user to get a strong understanding of the direction and magnitudes of the forces without sorting through the data points. It would allow the user to make FBD's for any link or joint they choose. All of the forces and directions are recorded in PMKS the challenge here was getting the users to know how to ask for this data. Free body diagrams are well known enough that FBD could work as a button label then. The next step was figuring out how to indicate to the user what link or joint they were selecting to avoid further confusion. The best way we saw was pausing the animation and having the links and joints react to being scrolled over. The last portion of this function was displaying the data, and this led to giving the user control over how the data should be displayed. All the forces are displayed as vectors; however the user can select x and y components or the user could ask for the resultant force and its direction.

3.3.4: Charting Links and Joints

Another addition that was intended to shorten the user's path from linkage creation to data acquisition is the charting functionality. It would allow the user to select a part within the animation and request graphs to visually depict the forces and torques over time or over angle. The first question to answer was how the user will select a part to request data for. We decided on the right click. This would simultaneously select a part and allow the user to start creating graphs. Then the next question was how to show all of the options for the graphs. It was far too many to fit in a right click box, so we elected to open a window to list the various options and controls for graph creation.

3.3.5: Section Conclusion

Over all these steps were taken to simplify the user's actions in PMKS and allow them to do more with their time. The calculations PMKS does are so in depth that it was more of a question of how all of this data can be best utilized for the end user. The creators did a lot for the calculation side however they left a lot to be desired on the user interface side.

Chapter 4: Implementation

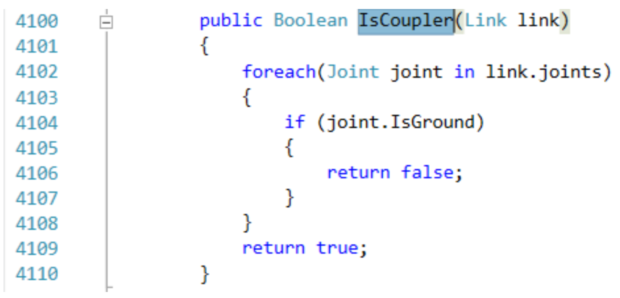
In this section, the focus is on the implementation of the analysis of the code foundation behind singularity check, work on converting PMKS to a desktop application, and the process of maximizing the UI application.

4.1: Code foundation behind Singularity Check

A simple check for whether a user-created mechanism will have any singular position has been implemented into the PMKS code for four-bar and six-bar linkages by adding code to the *Analysis()* function in file *FindFullMovement.cs.*

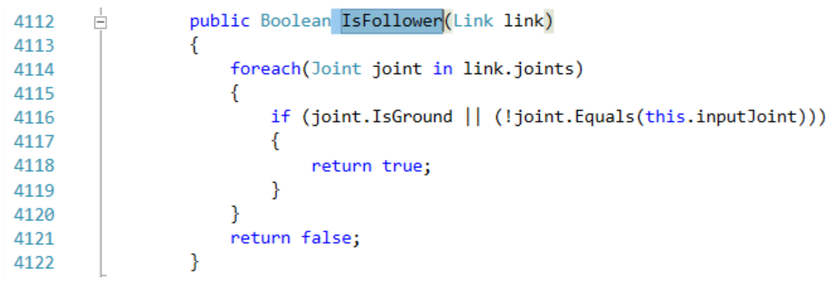
4.1.1: Singularity Check for Four-Bar

As described in section 4.1, a four-bar linkage has at least one singular position if its linkage lengths satisfy a specific relationship. Nevertheless, the check cannot be carried out if the program does not know the lengths of crank, coupler, follower and ground. PMKS already stores the information of input link (crank) and ground link, so helper functions ---- *isCoupler()* and *isFollower()* ---- are constructed to identify which link is coupler and which one is follower.



```
public Boolean IsCoupler(Link link)
{
    foreach(Joint joint in link.joints)
    {
        if (joint.IsGround)
        {
            return false;
        }
    }
    return true;
}
```

Figure 4.1: Code Structure of the Function *IsCoupler()*

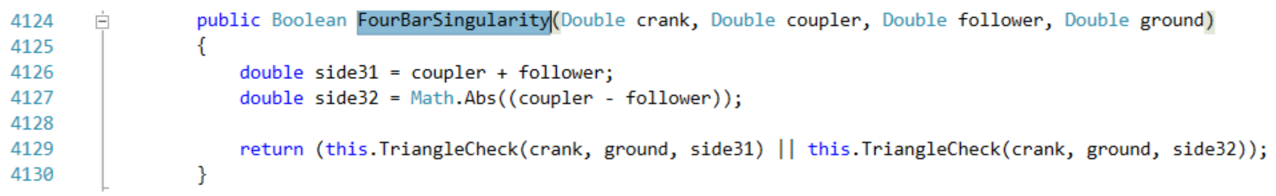


```
public Boolean IsFollower(Link link)
{
    foreach(Joint joint in link.joints)
    {
        if (joint.IsGround || (!joint.Equals(this.inputJoint)))
        {
            return true;
        }
    }
    return false;
}
```

Figure 4.2: Code Structure of the Function *IsFollower()*

- A link is a coupler if it does not have any grounded joint.
- A link is a follower if it is connected to ground but is not the input link.

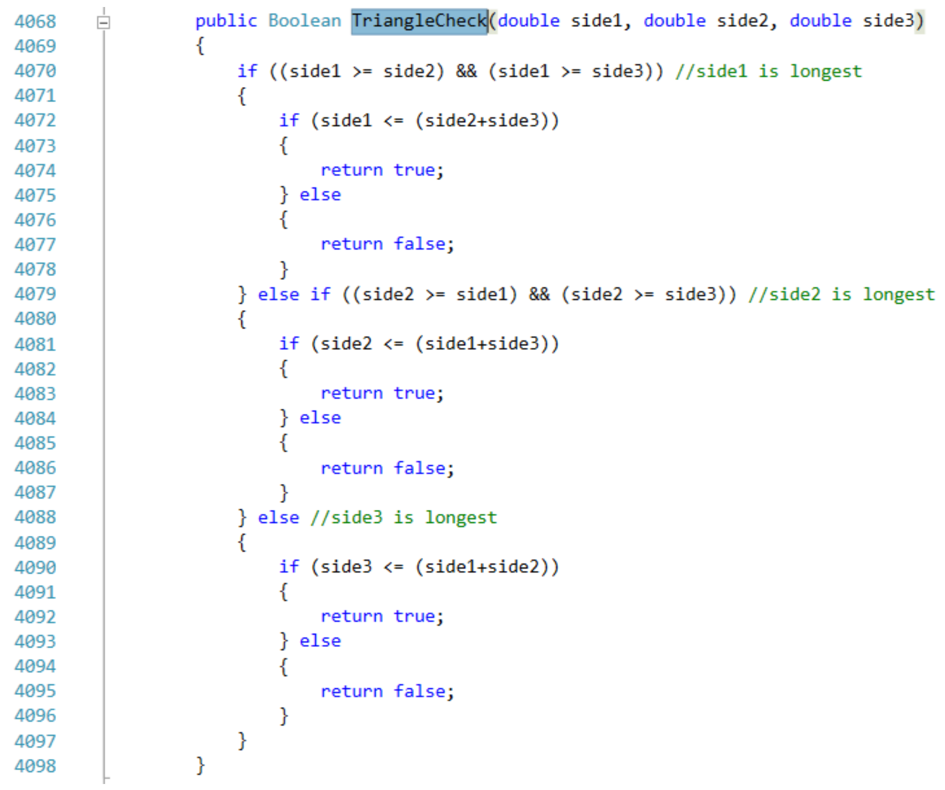
Using the logic above, these two functions find out the first two sides for the triangular-check method. After that, the function *FourBarSingularity()* calculates the length of the third side of the triangle and calls the function *TriangleCheck()*, which applies the triangular-check method and returns a boolean value to indicate the presence or absence of singular position.



```
public Boolean FourBarSingularity(Double crank, Double coupler, Double follower, Double ground)
{
    double side31 = coupler + follower;
    double side32 = Math.Abs((coupler - follower));

    return (this.TriangleCheck(crank, ground, side31) || this.TriangleCheck(crank, ground, side32));
}
```

Figure 4.3: Code Structure of the Function *FourBarSingularity()*



```
public Boolean TriangleCheck(double side1, double side2, double side3)
{
    if ((side1 >= side2) && (side1 >= side3)) //side1 is longest
    {
        if (side1 <= (side2+side3))
        {
            return true;
        } else
        {
            return false;
        }
    } else if ((side2 >= side1) && (side2 >= side3)) //side2 is longest
    {
        if (side2 <= (side1+side3))
        {
            return true;
        } else
        {
            return false;
        }
    } else //side3 is longest
    {
        if (side3 <= (side1+side2))
        {
            return true;
        } else
        {
            return false;
        }
    }
}
```

Figure 4.4: Code Structure of the Function *TriangleCheck()*

The following three lengths are calculated:

- Length of crank.
- Length of ground.
- Length of the third side of triangle, which can be either the sum of lengths of coupler and follower, or the difference between them.

The three lengths are then checked if they satisfy the relationship:

$Longest\ segment\ <=\ Sum\ of\ the\ other\ two$

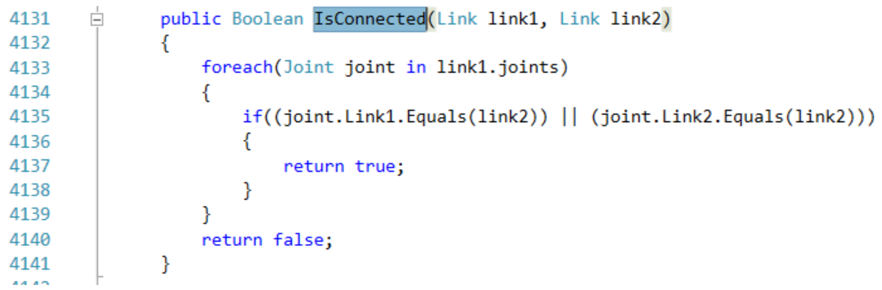
If this condition is met, the four-bar will have singular position. Currently these functions are called before PMKS runs analysis on a four-bar linkage.

4.1.2: Singularity Check for Six-Bar

The current version of singularity check for six-bar linkage follows the same approach as for four-bar. Since a six-bar can be viewed as two four-bars connected in parallel, if either of them has singular position, PMKS will report that the six-bar contains singularity (however, this logic has several limitations, which will be explained in the discussion section).

In order to carry out singularity check for each of the two four-bar sub-mechanisms, the lengths of crank, coupler, follower and ground needs to be determined for each of the two four-bar subsystems. Additional helper functions are constructed for this purpose.

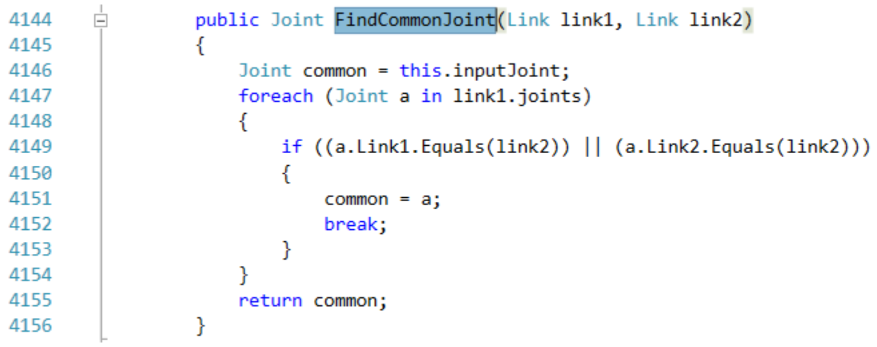
- Function is*Connected()* checks if two links are connected (i.e. share a common joint).



```
public Boolean IsConnected(Link link1, Link link2)
{
    foreach(Joint joint in link1.joints)
    {
        if((joint.Link1.Equals(link2)) || (joint.Link2.Equals(link2)))
        {
            return true;
        }
    }
    return false;
}
```

Figure 4.5: Code structure of the function *isConneccted()*

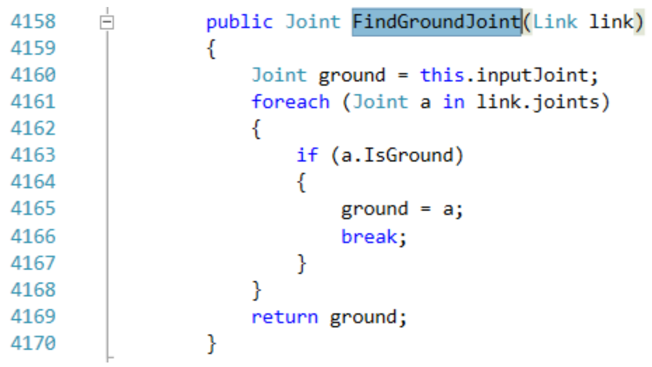
- Function *getCommonJoint()* returns the shared joint between two connected links.



```
public Joint FindCommonJoint(Link link1, Link link2)
{
    Joint common = this.inputJoint;
    foreach (Joint a in link1.joints)
    {
        if ((a.Link1.Equals(link2)) || (a.Link2.Equals(link2)))
        {
            common = a;
            break;
        }
    }
    return common;
}
```

Figure 4.6: Code structure of the function *FindCommonJoint()*

- Function *getGroundJoint()* returns the grounded joint of a link connected to ground.



```
public Joint FindGroundJoint(Link link)
{
    Joint ground = this.inputJoint;
    foreach (Joint a in link.joints)
    {
        if (a.IsGround)
        {
            ground = a;
            break;
        }
    }
    return ground;
}
```

Figure 4.7: Code Structure of the function *FindGroundJoint()*

In order to identify the links in each four-bar sub-mechanism, a specific algorithm is set up to find out the identity of each link, which is displayed in **Figure 4.8**:

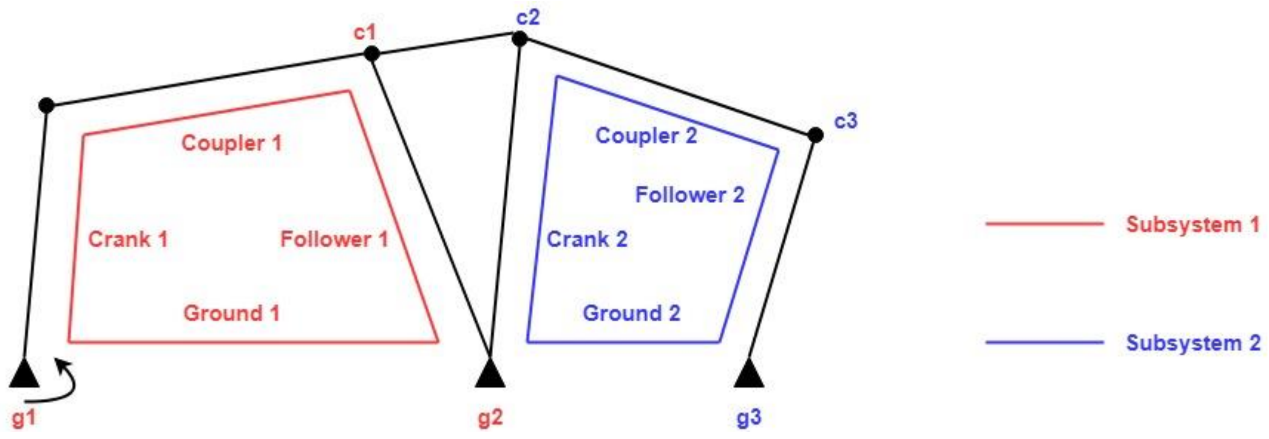


Figure 4.8: Four-bar subsystems of a six-bar linkage

- Links to identify:

Crank 1 = input link;

Coupler 1 = the link that is coupler and is connected to input link;

Follower 1 = the link that is ground link and is connected to coupler 1 (the ternary link);

Coupler 2 = the link that is coupler and is not connected to input link;

Follower 2 = the link that is ground link and is connected to coupler 2 (the ternary link);

- Joints to identify:

Joint c1 = common joint between coupler 1 and follower 1;

Joint c2 = common joint between coupler 2 and follower 1;

Joint c3 = common joint between coupler 2 and follower 2;

Joint g1 = grounded joint of follower 1;

Joint g2 = grounded joint of follower 2;

- Lengths to calculate:

Length of follower 1 = distance between c1 and g1;

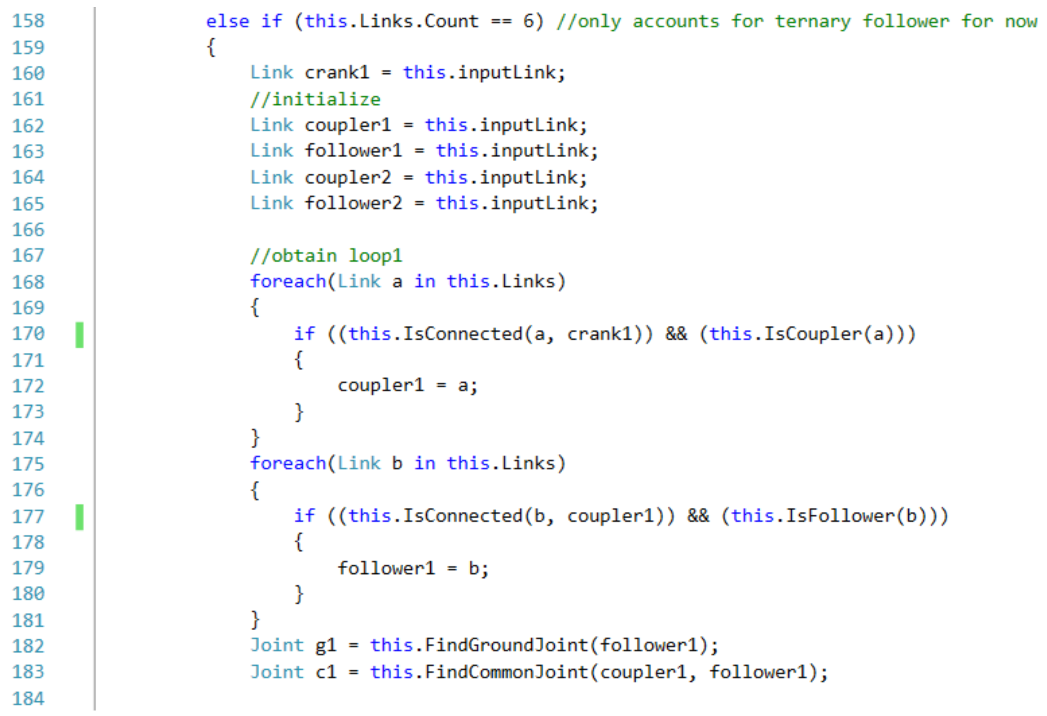
Length of ground 1 = distance between input joint and g1;

Length of crank 2 = distance between g1 and c2;

Length of follower 2 = distance between c3 and g2;

Length of ground 2 = distance between g1 and g2;

The code below shows how the algorithm described above is actually implemented in the function *Analysis()*. It is called whenever analysis is run for a six-bar linkage. Subsystem 1 was identified first through two for loops, after which subsystem 2 was found out using the same process.

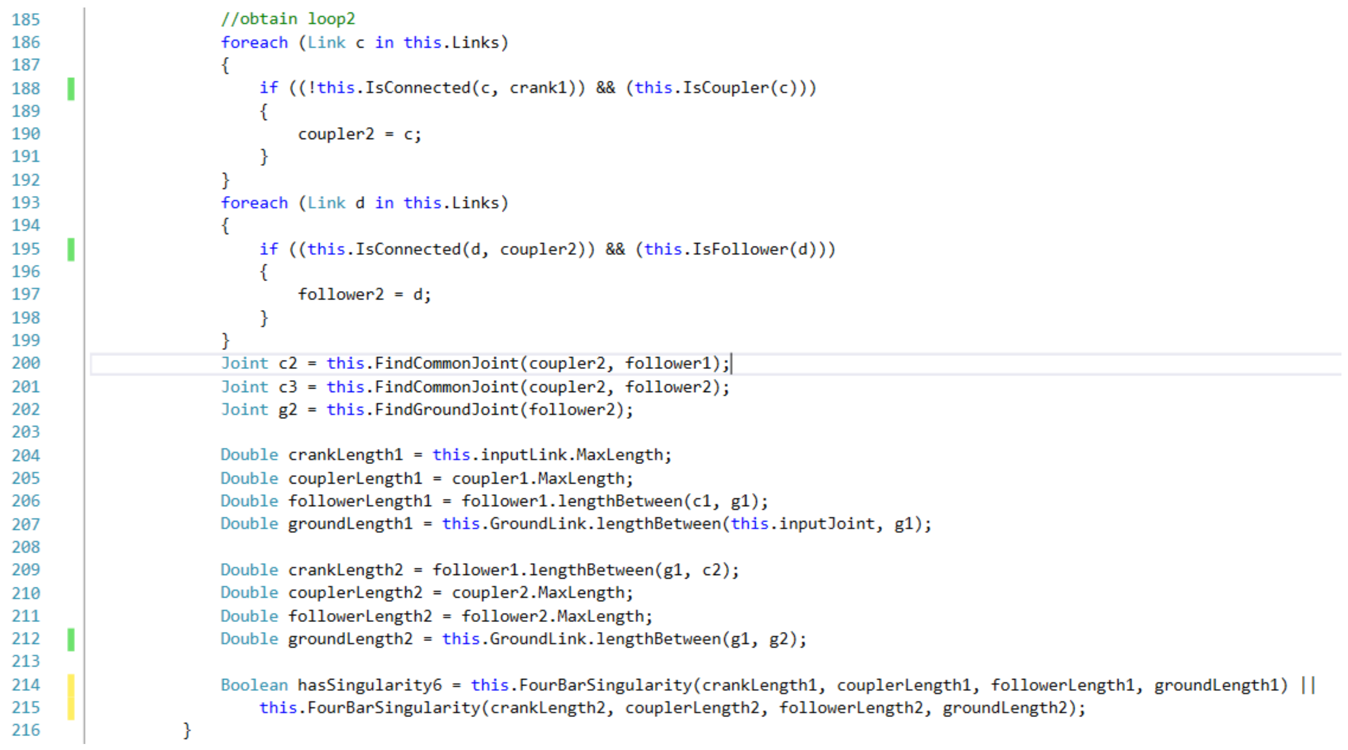


```
            else if (this.Links.Count == 6) //only accounts for ternary follower for now
            {
                Link crank1 = this.inputLink;
                //initialize
                Link coupler1 = this.inputLink;
                Link follower1 = this.inputLink;
                Link coupler2 = this.inputLink;
                Link follower2 = this.inputLink;

                //obtain loop1
                foreach(Link a in this.Links)
                {
                    if ((this.IsConnected(a, crank1)) && (this.IsCoupler(a)))
                    {
                        coupler1 = a;
                    }
                }
                foreach(Link b in this.Links)
                {
                    if ((this.IsConnected(b, coupler1)) && (this.IsFollower(b)))
                    {
                        follower1 = b;
                    }
                }
                Joint g1 = this.FindGroundJoint(follower1);
                Joint c1 = this.FindCommonJoint(coupler1, follower1);

```

Figure 4.9(a): Code section in the function Analysis () for singularity check of six-bar linkages



```
                //obtain loop2
                foreach (Link c in this.Links)
                {
                    if ((!this.IsConnected(c, crank1)) && (this.IsCoupler(c)))
                    {
                        coupler2 = c;
                    }
                }
                foreach (Link d in this.Links)
                {
                    if ((this.IsConnected(d, coupler2)) && (this.IsFollower(d)))
                    {
                        follower2 = d;
                    }
                }
                Joint c2 = this.FindCommonJoint(coupler2, follower1);
                Joint c3 = this.FindCommonJoint(coupler2, follower2);
                Joint g2 = this.FindGroundJoint(follower2);

                Double crankLength1 = this.inputLink.MaxLength;
                Double couplerLength1 = coupler1.MaxLength;
                Double followerLength1 = follower1.lengthBetween(c1, g1);
                Double groundLength1 = this.GroundLink.lengthBetween(this.inputJoint, g1);

                Double crankLength2 = follower1.lengthBetween(g1, c2);
                Double couplerLength2 = coupler2.MaxLength;
                Double followerLength2 = follower2.MaxLength;
                Double groundLength2 = this.GroundLink.lengthBetween(g1, g2);

                Boolean hasSingularity6 = this.FourBarSingularity(crankLength1, couplerLength1, followerLength1, groundLength1) ||
                    this.FourBarSingularity(crankLength2, couplerLength2, followerLength2, groundLength2);
            }
```

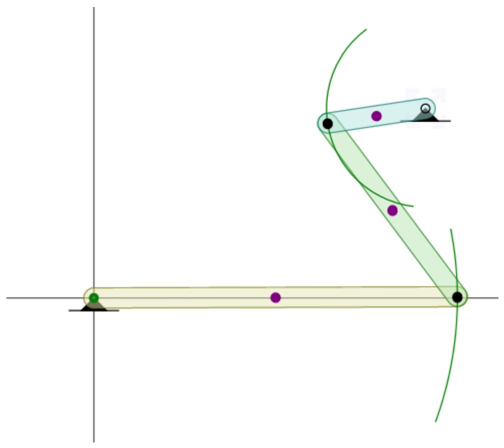
Figure 4.9(b): Code section in the function Analysis () for singularity check of six-bar linkages (resumed)

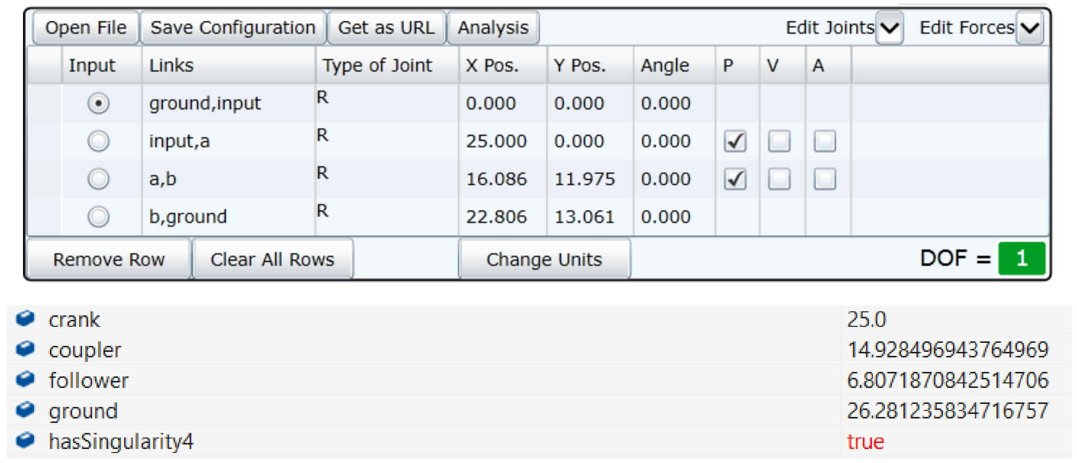
After the links for each four-bar are identified, singularity check is conducted for each of the two sub-mechanisms. If either of them is found to have singular position, it will be possible for the six-bar linkage to have singular position as well.

4.1.3: Testing Singularity Check

In order to check if the singularity identification functionality was operating as designed, a number of tests were conducted to check different portions of the algorithm. Shown below are a few examples of the test cases used.

- Four-bar with singularity





Input	Links	Type of Joint	X Pos.	Y Pos.	Angle	P	V	A
●	ground,input	R	0.000	0.000	0.000			
○	input,a	R	25.000	0.000	0.000	✓	☐	☐
○	a,b	R	16.086	11.975	0.000	✓	☐	☐
○	b,ground	R	22.806	13.061	0.000			

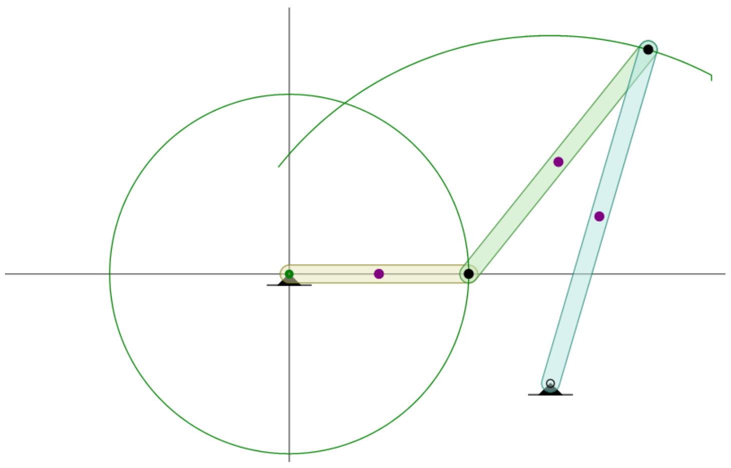
DOF = 1

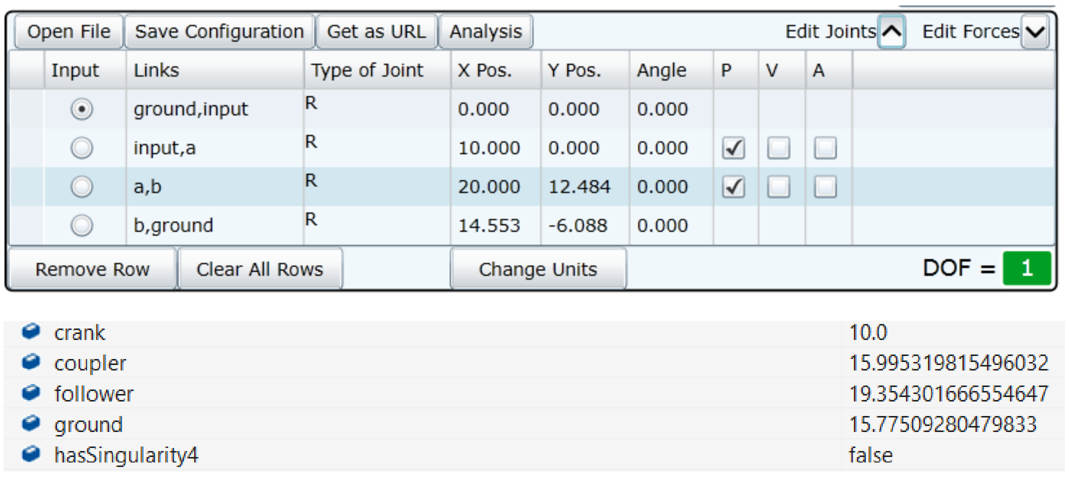
crank	25.0
coupler	14.928496943764969
follower	6.8071870842514706
ground	26.281235834716757
hasSingularity4	true

Figure 4.10: Test case for singularity check of four-bar linkage when singularity is present, and the results

This four-bar linkage had singular positions as its coupler can either line up with the follower or overlap with it. The code not only succeeded in retrieving the lengths of all links but also generated a boolean value (*hasSingularity4*) of "True", which predicted the presence of singular position.

- Four-bar without singularity





Input	Links	Type of Joint	X Pos.	Y Pos.	Angle	P	V	A
●	ground,input	R	0.000	0.000	0.000			
○	input,a	R	10.000	0.000	0.000	✓	☐	☐
○	a,b	R	20.000	12.484	0.000	✓	☐	☐
○	b,ground	R	14.553	-6.088	0.000			

DOF = 1

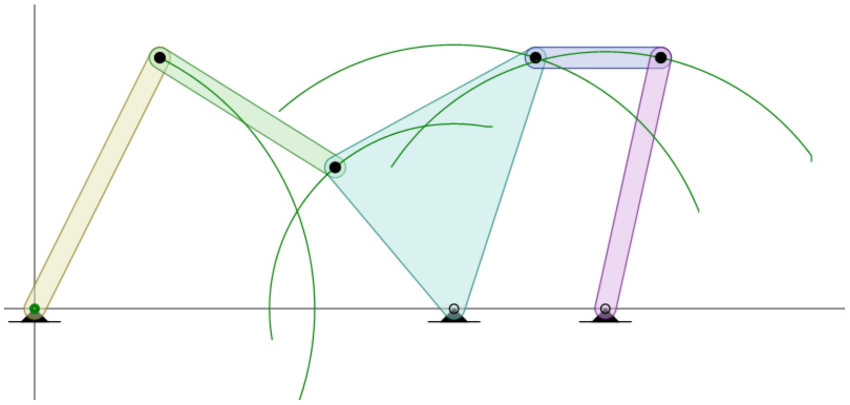
crank	10.0
coupler	15.995319815496032
follower	19.354301666554647
ground	15.77509280479833
hasSingularity4	false

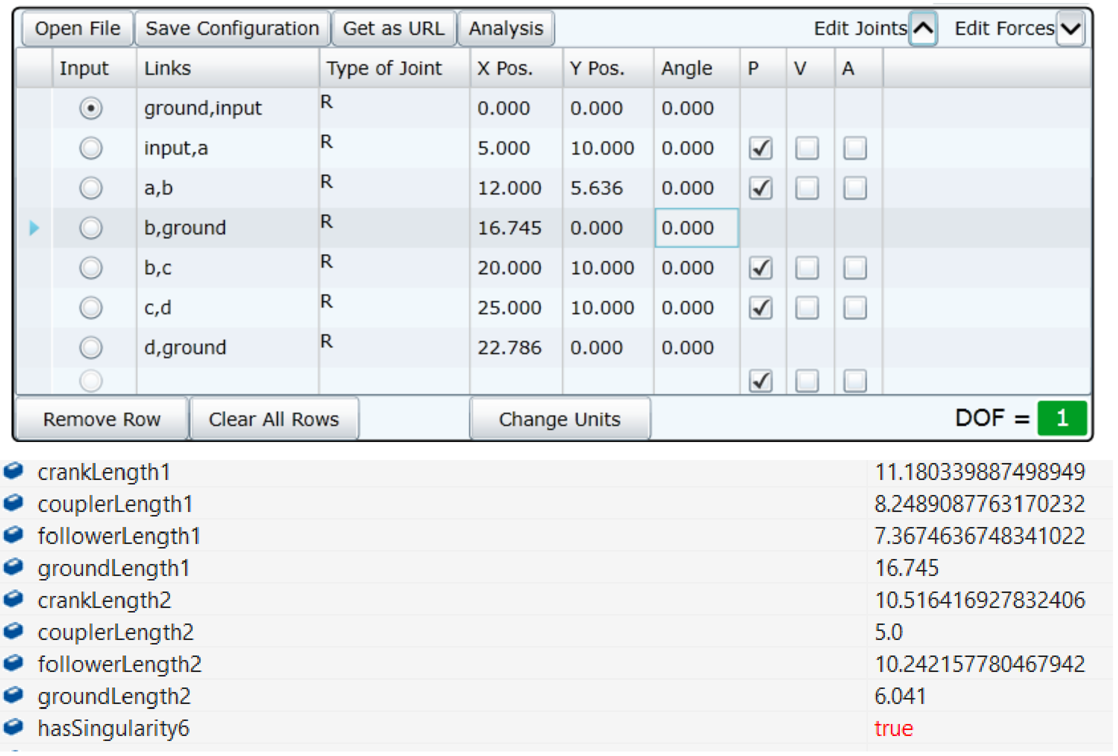
Figure 4.11 Test case for singularity check of four-bar linkage when singularity is absent, and the results

This four-bar linkage is an example of four-bar with no singular position, as its coupler and follower can never perfectly line up. The code returned a boolean value of "False", indicating the absence of singular position

- Identifying four-bar subsystems in a six-bar

The algorithm for detecting the presence of singular position in a six-bar linkage, as described in section 4.7, is based on the singularity check for four-bar, and the main difference is the way for PMKS to identify the links for each four-bar subsystem individually. Shown below is one of the test cases for checking if the code was functioning properly.





Input	Links	Type of Joint	X Pos.	Y Pos.	Angle	P	V	A
●	ground,input	R	0.000	0.000	0.000			
○	input,a	R	5.000	10.000	0.000	✓	☐	☐
○	a,b	R	12.000	5.636	0.000	✓	☐	☐
○	b,ground	R	16.745	0.000	0.000			
○	b,c	R	20.000	10.000	0.000	✓	☐	☐
○	c,d	R	25.000	10.000	0.000	✓	☐	☐
○	d,ground	R	22.786	0.000	0.000			
○						✓	☐	☐

DOF = 1

crankLength1	11.180339887498949
couplerLength1	8.2489087763170232
followerLength1	7.3674636748341022
groundLength1	16.745
crankLength2	10.516416927832406
couplerLength2	5.0
followerLength2	10.242157780467942
groundLength2	6.041
hasSingularity6	true

Figure 4.12: Test case for singularity check of six-bar linkage when singularity is present, and the results

The code successfully identified the corresponding crank, coupler, follower and ground links for each four-bar sub-mechanism and obtained their lengths. It also returned a boolean value of “True” (*hasSingularity6*), indicating the presence of singular position in this six-bar.

4.2: Work on the PMKS Software

In this section, we go over work done debugging the WPF desktop application version of PMKS as well as flaws found in the software design.

4.2.1: Debugging the Desktop Application

The PMKS application was programmed in C# using Microsoft Visual Studio 2015. The 2015 version was used because it has no compatibility issues with running applications built on Silverlight. In the Visual Studio Solution titled "PMKS", there are a number of projects, as seen in Figure 4.5.1. The "PlanarMechanismKinematicSimulator" project contains the code that does the mathematical analyses on the linkages. This is then used by the "PMKS_Web_MQP" project, which is the code for the Silverlight web application, to compute the various analyses on the linkage in the application. The "MQP_Convert" project contains the code for the WPF desktop application. The "MQP_Convert" project uses the "PlanarMechanismKinematicSimulator" project for the mathematical analyses on the linkages, just like the "PMKS_Web_MQP" project does.

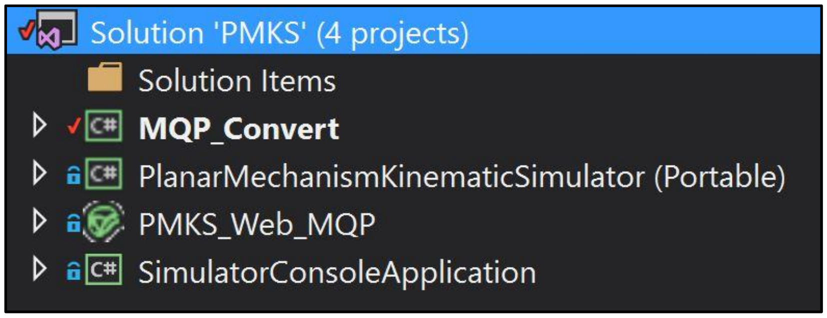
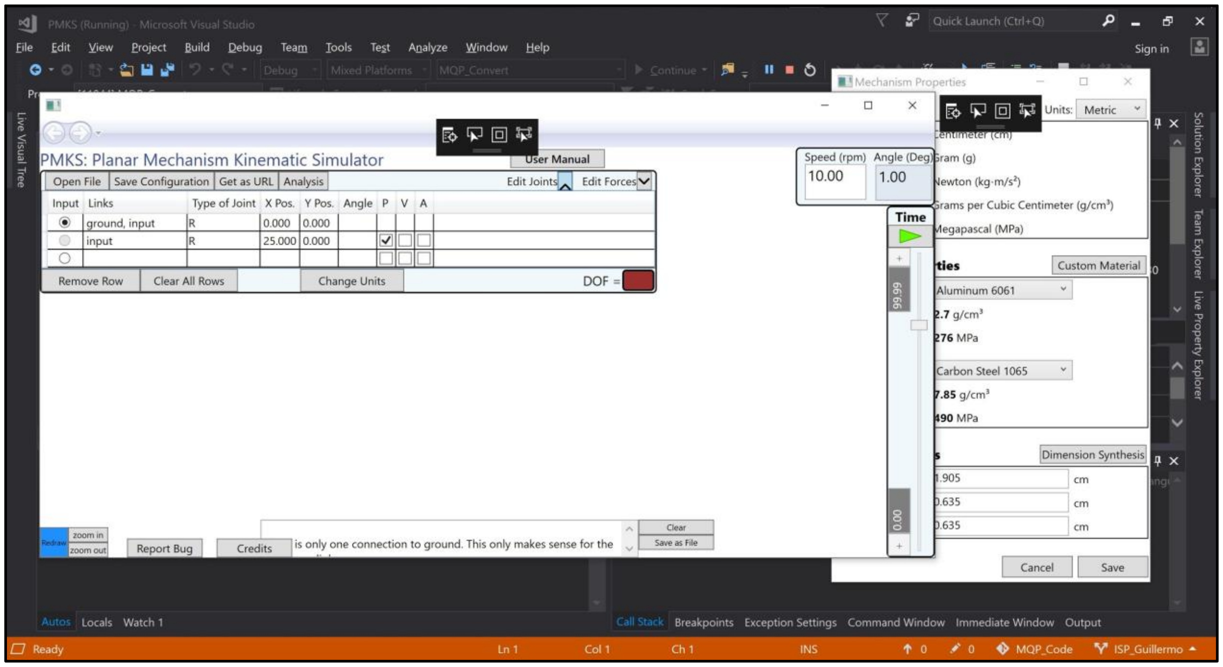

Solution 'PMKS' (4 projects)
Solution Items
MQP_Convert
PlanarMechanismKinematicSimulator (Portable)
PMKS_Web_MQP
SimulatorConsoleApplication

Figure 4.13: Visual Studio solution hierarchy

Upon starting the desktop application code, two windows open up. One of them being the main PMKS window, seen on the left in **Figure 4.13.** To the right of the main PMKS window is the link properties window. When the link properties were saved with the lower right hand "Save" button, the whole software would crash, and it gave errors related to all the functions from Silverlight not being completely supported by WPF. Additionally, if the user tried to click anywhere in the main PMKS window, the whole software would crash. Without understanding how the code was designed, and without knowing where the errors were, we had to spend time in the debugger figuring out how functions and classes work together.

We first temporarily removed the link properties window, so upon startup, only the main PMKS window would open. The one thing that wasn't visibly working was the link animation. On startup, there should be a default link that is animated to give the user a starting point, as seen in **Figure 4.14**.



Input	Links	Type of Joint	X Pos.	Y Pos.	Angle	P	V	A
●	ground, input	R	0.000	0.000				
○	input	R	25.000	0.000		✓	☐	☐
○						☐	☐	☐

DOF = 🟥

Speed (rpm) 10.00 Angle (Deg) 1.00

Units: Metric

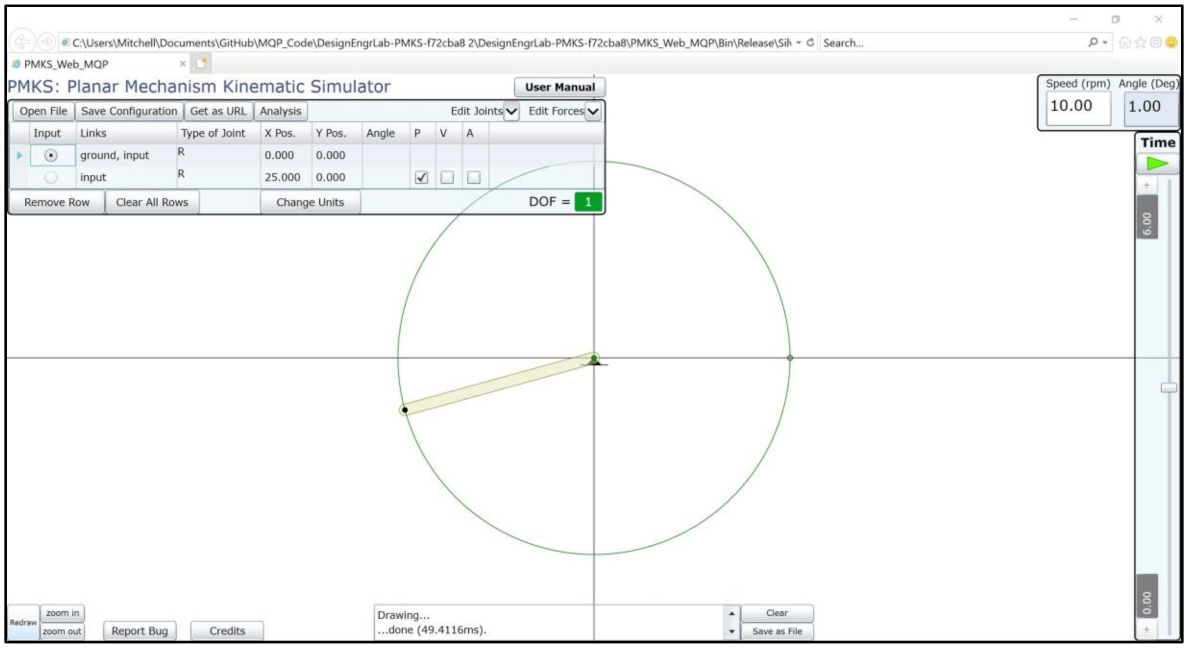
entimeter (cm)
Gram (g)
Newton (kg·m/s²)
Grams per Cubic Centimeter (g/cm³)
Megapascal (MPa)

Aluminum 6061
2.7 g/cm³
276 MPa
Carbon Steel 1065
7.85 g/cm³
490 MPa

1.905 cm
0.635 cm
0.635 cm

is only one connection to ground. This only makes sense for the

Figure 4.14: PMKS desktop application on startup



Input	Links	Type of Joint	X Pos.	Y Pos.	Angle	P	V	A
●	ground, input	R	0.000	0.000				
○	input	R	25.000	0.000		✓	☐	☐

DOF = 1

Speed (rpm) 10.00 Angle (Deg) 1.00

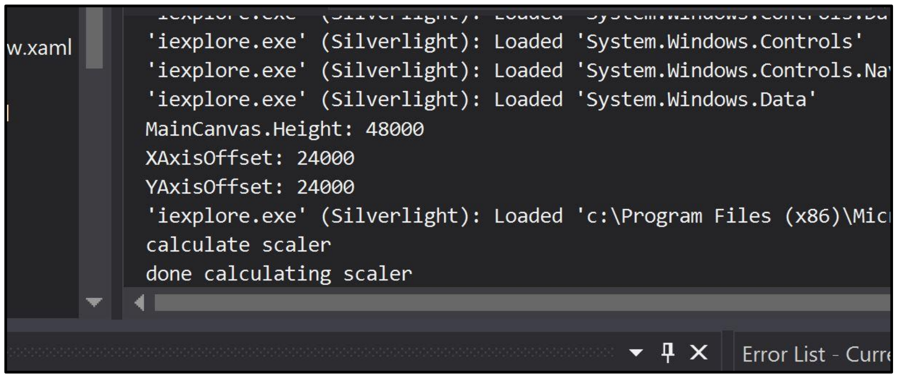
Drawing...
...done (49.4116ms).

Figure 4.15: PMKS web application on startup

The default linkage has two links; a ground link and a link named “input”, as seen in the table at the top left of the main PMKS window. All the other buttons and UI subsections were being shown in the new, WPF version, albeit they were not working as expected.

After reading Microsoft’s documentation and debugging the software, it turned out the application is made up of elements WPF refers to as “Panels”. PMKS was designed with one main panel that contains a lot of different sub-panels. One thing that was difficult to understand and fix was the multiple namespaces in the software. There were two namespaces that seemed to follow no pattern as to which classes used what. The reason this happened was two people were working on the project in different areas, and as to not mess up the other person’s code, they both used different, similarly named namespaces. This is a bad design decision and could not be easily remedied due to a number of variables having the same names just in different namespaces. Because these variables are used in a number of different files, it is very difficult to accurately change the variable names in an effort to have one master namespace for the project. After trying to fix the aforementioned issue, the easiest way to get rid of some of the errors related to the animation was to bring the sub-panel that contained the animation into the main PMKS window.

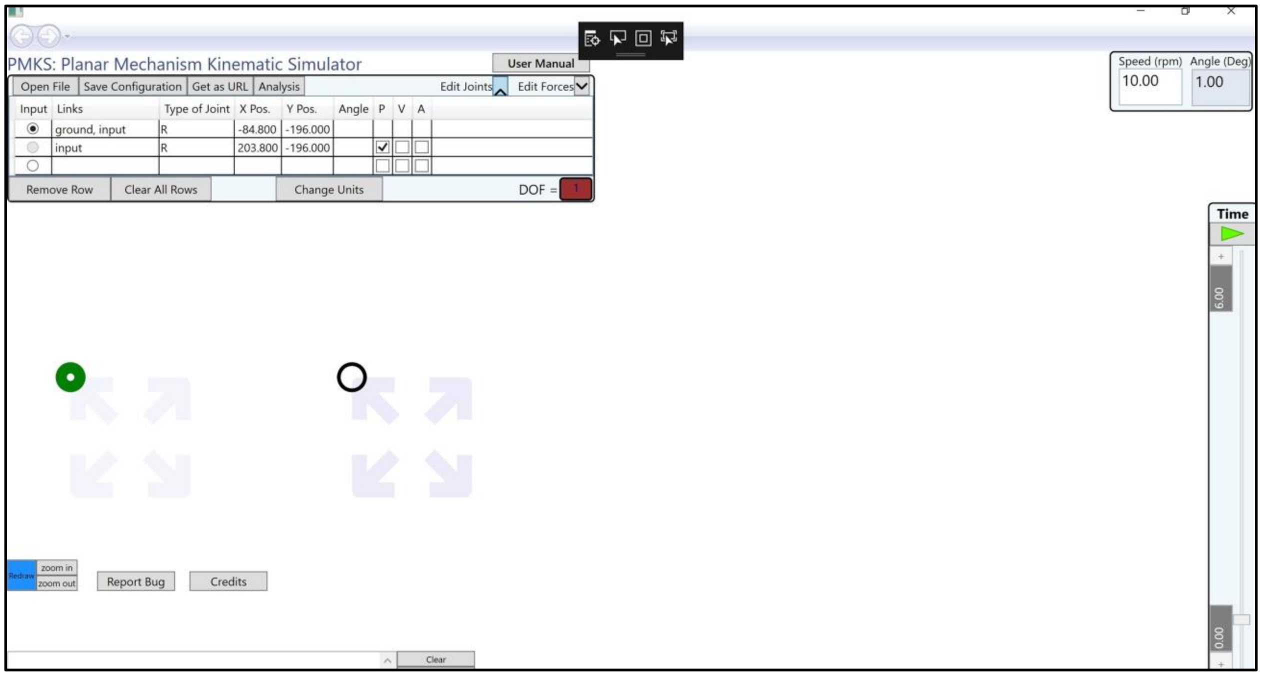
The animation was on a special kind of panel known as a “Canvas” element. This element is used for animations, and one of its features is that the animation uses pixel values as coordinates. Through debugging the software, we came to find the linkage joints were using very high values for coordinates, as seen in **Figure 4.16**.



```
'iexplore.exe' (Silverlight): Loaded 'System.Windows.Controls'
'iexplore.exe' (Silverlight): Loaded 'System.Windows.Controls.Na
'iexplore.exe' (Silverlight): Loaded 'System.Windows.Data'
MainCanvas.Height: 48000
XAxisOffset: 24000
YAxisOffset: 24000
'iexplore.exe' (Silverlight): Loaded 'c:\Program Files (x86)\Mic
calculate scaler
done calculating scaler
```

Figure 4.16: Visual Studio console out containing variables and values

The “XAxisOffset” and “YAxisOffset” are used to locate the middle of the canvas element. As shown above, the software was trying to display the linkage animation thousands of pixels off screen. After managing to change these values, the static linkage joints were displayed, as seen in **Figure 4.17**.

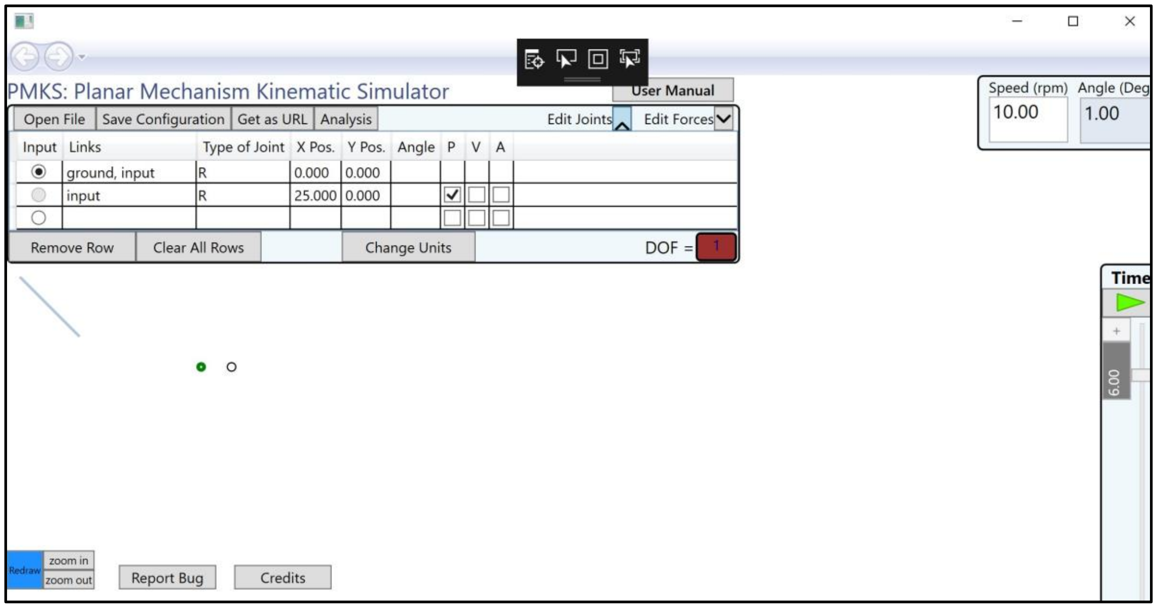


Input	Links	Type of Joint	X Pos.	Y Pos.	Angle	P	V	A
●	ground, input	R	-84.800	-196.000				
	input	R	203.800	-196.000		✓	☐	☐
○						☐	☐	☐

Figure 4.17: PMKS desktop application with joints displayed

The two circles under the table are enlarged joint points. At this point, we knew that there was some functionality in the code that ran the linkage animation, however there was no moving linkage. In addition, the Y axis was mirrored, so the lower the value, the higher on screen the joint would go.

To see why only static shapes were being displayed, the team tried to draw simple lines in different parts of the code, to see what was functioning properly. In the function that displays static shapes, the team was able to display a line as seen in **Figure 4.18**.



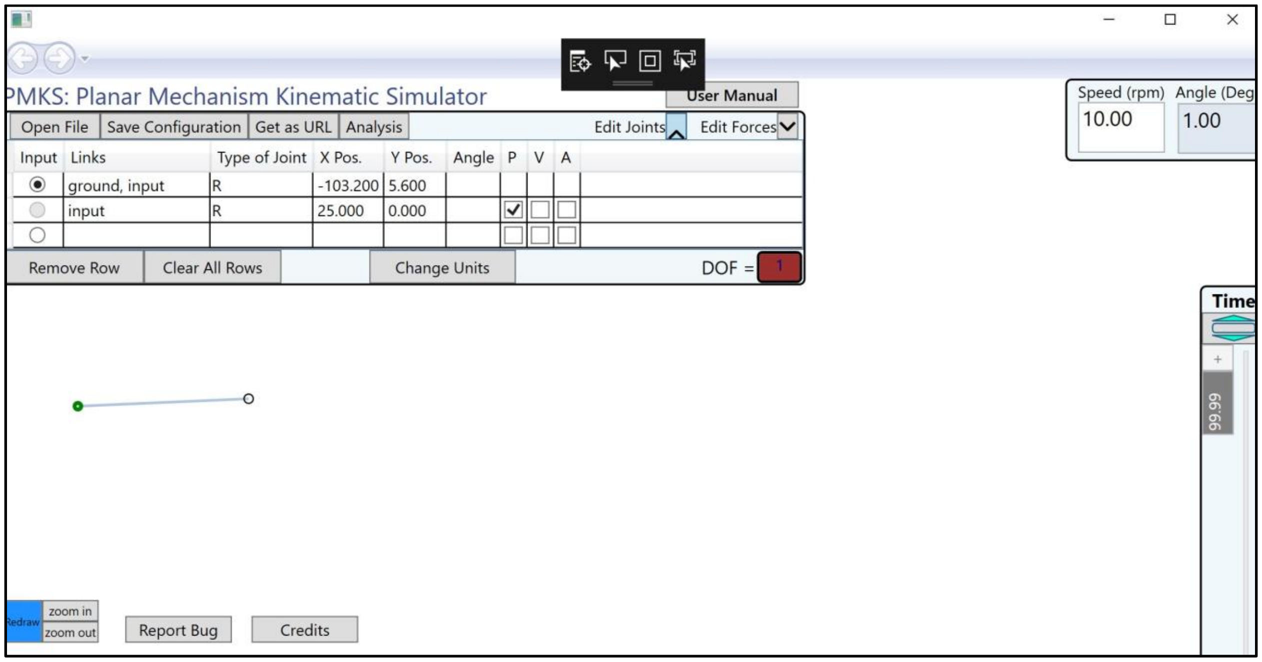
Input	Links	Type of Joint	X Pos.	Y Pos.	Angle	P	V	A
●	ground, input	R	0.000	0.000				
○	input	R	25.000	0.000		✓	☐	☐
○						☐	☐	☐

DOF = 1

Speed (rpm)	Angle (Deg
10.00	1.00

Figure 4.18: PMKS desktop application with joints and a drawn line

I was then able to make the line connect the joints and respond to moving the joints, as seen in **Figure 4.19** and **Figure 4.20**.

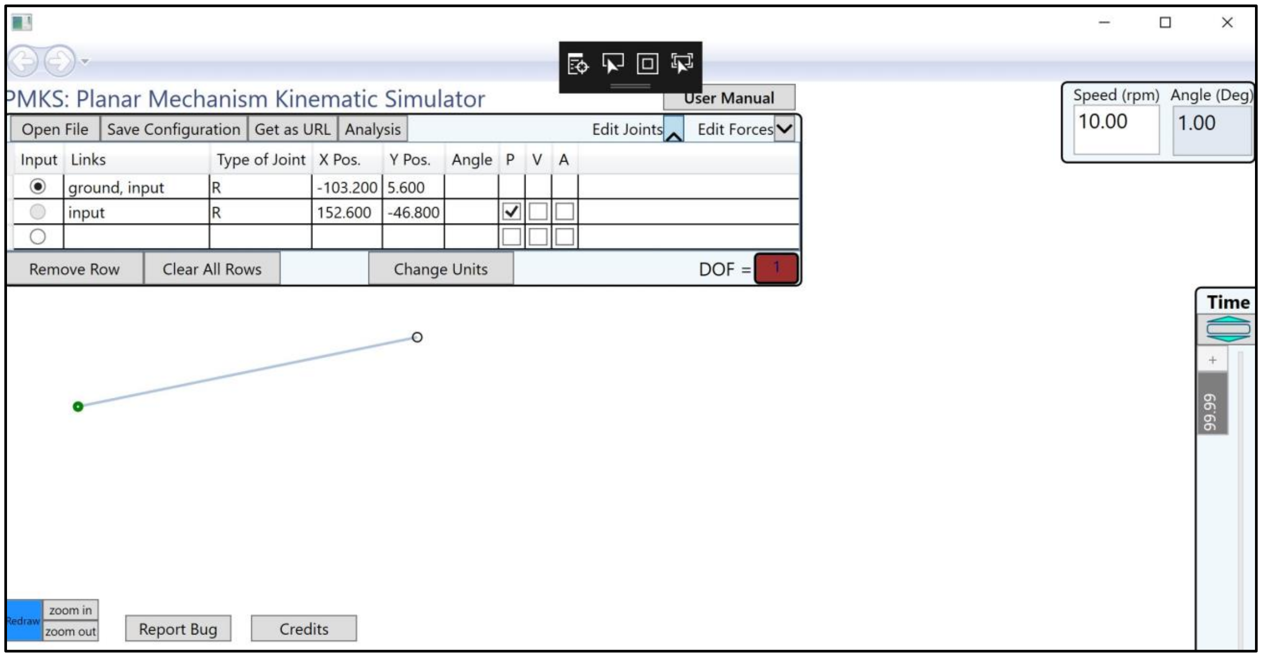


Input	Links	Type of Joint	X Pos.	Y Pos.	Angle	P	V	A
●	ground, input	R	-103.200	5.600				
○	input	R	25.000	0.000		✓	☐	☐
○						☐	☐	☐

DOF = 1

Speed (rpm)	Angle (Deg
10.00	1.00

Figure 4.19: PMKS desktop application with the line between the joints (before moving joint)



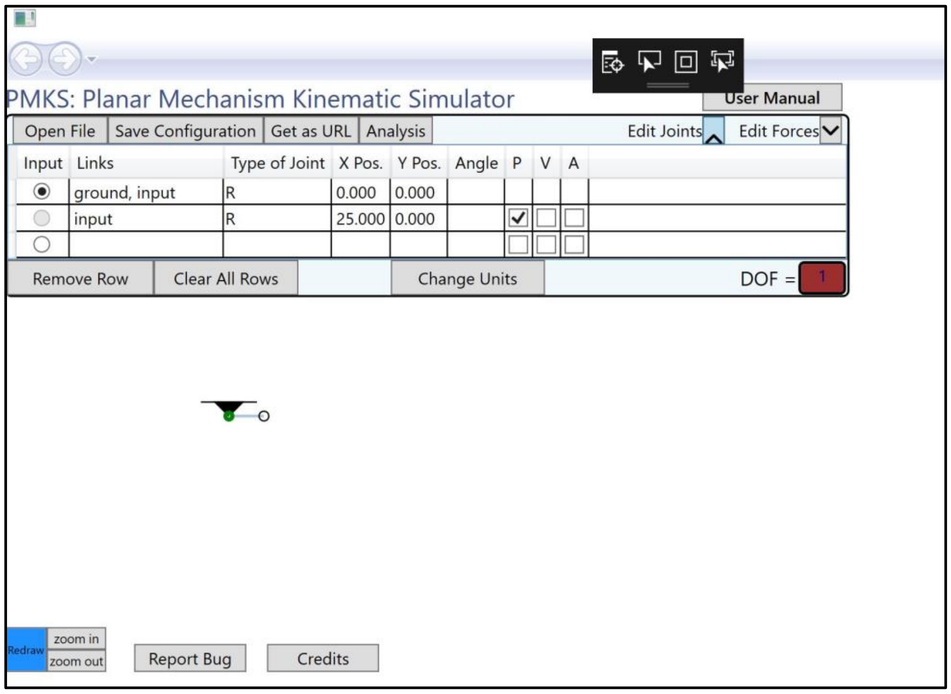
Speed (rpm)	Angle (Deg)
10.00	1.00

Input	Links	Type of Joint	X Pos.	Y Pos.	Angle	P	V	A
●	ground, input	R	-103.200	5.600				
○	input	R	152.600	-46.800		✓	☐	☐
○						☐	☐	☐

DOF = 1

Figure 4.20: PMKS desktop application with the line redrawn between joints (after moving joint)

In addition to this, we were able to get the static ground link shape to display, albeit upside down due to the Y axis being flipped, as seen in **Figure 4.21**.



Input	Links	Type of Joint	X Pos.	Y Pos.	Angle	P	V	A
●	ground, input	R	0.000	0.000				
○	input	R	25.000	0.000		✓	☐	☐
○						☐	☐	☐

DOF = 1

Figure 4.21: PMKS desktop application with ground link shown upside down

Further debugging revealed that the problem with the dynamic shapes not being displayed was related to how animation in WPF was done differently than in Silverlight.

In the Silverlight version of PMKS, a built-in class called Path was being used. Animation in WPF is done without this. When PMKS was being briefly converted last year, instead of rebuilding the animation the WPF way, the previous group created their own Path class in an attempt to reuse as much of the Silverlight code as possible. This is the real root of the problem as to why the animation isn't displaying some things like link shapes and joint paths and why it doesn't move.

The changes needed to get the code working extends the time allocated for this project. In order to best help the next team work on it, onboarding documentation and diagrams needed to be made to bring a new group up to speed on PMKS. The beginning of the project where we were ramping up, trying to understand what our project was and how to do it was the hardest part. With improved documentation, future teams will have an easier time understanding the project and what they need to do to fix it.

4.2.2: Documentation and Design Proposal

As seen in **Figure 4.22**, PMKS currently handles animations in the MainPage class. There are two different functions in this class, one handles the animation of static shapes, while the other handles the animation of dynamic shapes. Right now, the function that handles static shapes is working, but the function for dynamic shapes is not.

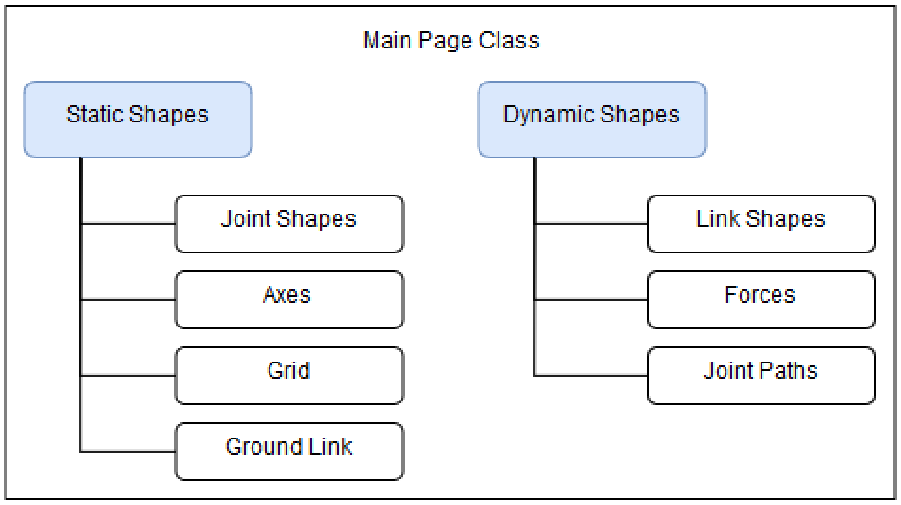


Figure 4.22: Organization of how the animations are handled

Most of the functionality for PMKS is handled in this aforementioned MainPage class. This class handles displaying UI elements, displaying and animating the linkage, binding user input data to variables in the code that handles linkage analyses, exporting linkage data to a spreadsheet, and more. This makes the MainPage class verbose and hard to maintain. After analyzing the software and its design, we recommend many of these software functions be abstracted and reorganized to make it easier to understand and modify in the future.

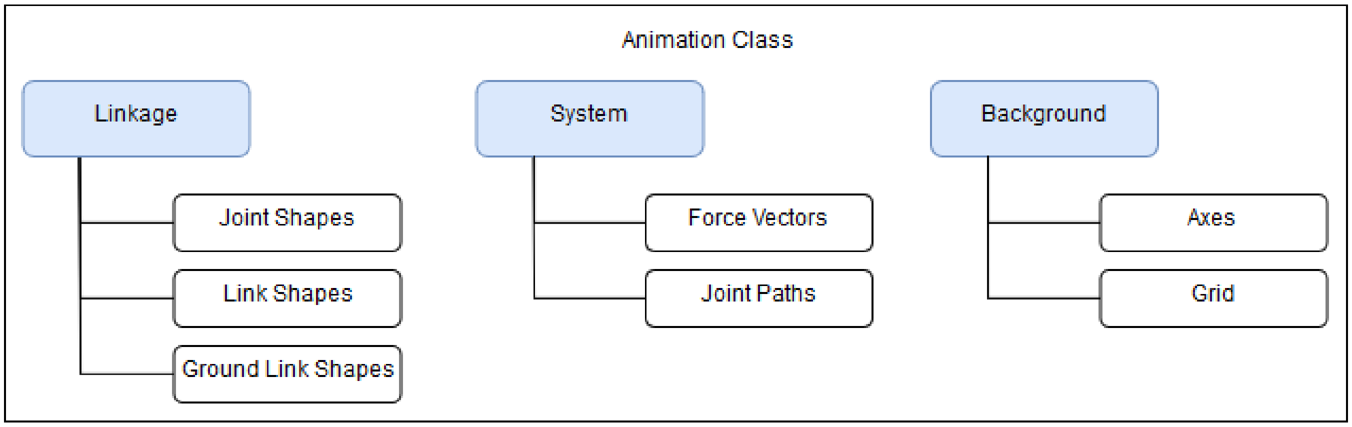


Figure 4.23: Proposed structure of code that handles linkage animation

Seen in **Figure 4.23**, this proposed structure includes the addition of an animation class that handles all animations in PMKS and is located outside of the MainPage class. Instead of having two functions animating static and dynamic shapes, we have decided that it would be better to separate the animation into functional parts. This would allow for easier cross talk between linkage elements, and structure the software based on the actual parts as opposed to the behavior of those parts.

4.3: Maximizing the UI experience

As a team we did not reach the point where we could implement these types of changes instead there is documentation of these features and how they should be implemented in the future for the next MQP team. The focus is on streamlining and simplifying existing tasks as well as adding features that give the user greater capabilities from the same data.

4.3.1: Force Entry on Cyclical Loading

As mentioned previously above, the methods for applying forces was very complex and virtually unusable to a new user. The goal was to make a simple system that would hold the users hand and help them to understand what they can do with this program. The main plan is to help the user with the first step of applying a force, which is positioning the force. When the user goes to position the force, the display will change. The change will let them know that they are selecting a position for the force. Additionally, it will help them with applying that force correctly. Once the user selects the position fields in the edit forces tab the display will darken. The links and joints will then enlarge as a reaction to being scrolled over. When the user selects a link the force's origin will snap to the middle of the link and the global coordinates will be displayed in the edit forces position fields. This simplifies the users input method as they need to only select the link in question rather than find its global coordinates. The origin of the force will be a small star like symbol, distinctly different from a joint. It can be moved with the global coordinates as well as with a click and drag. When using the click and drag function the cursor position indication box will change to show three decimal places instead of the one decimal places it usually displays; allowing for finer controls of the force placement.

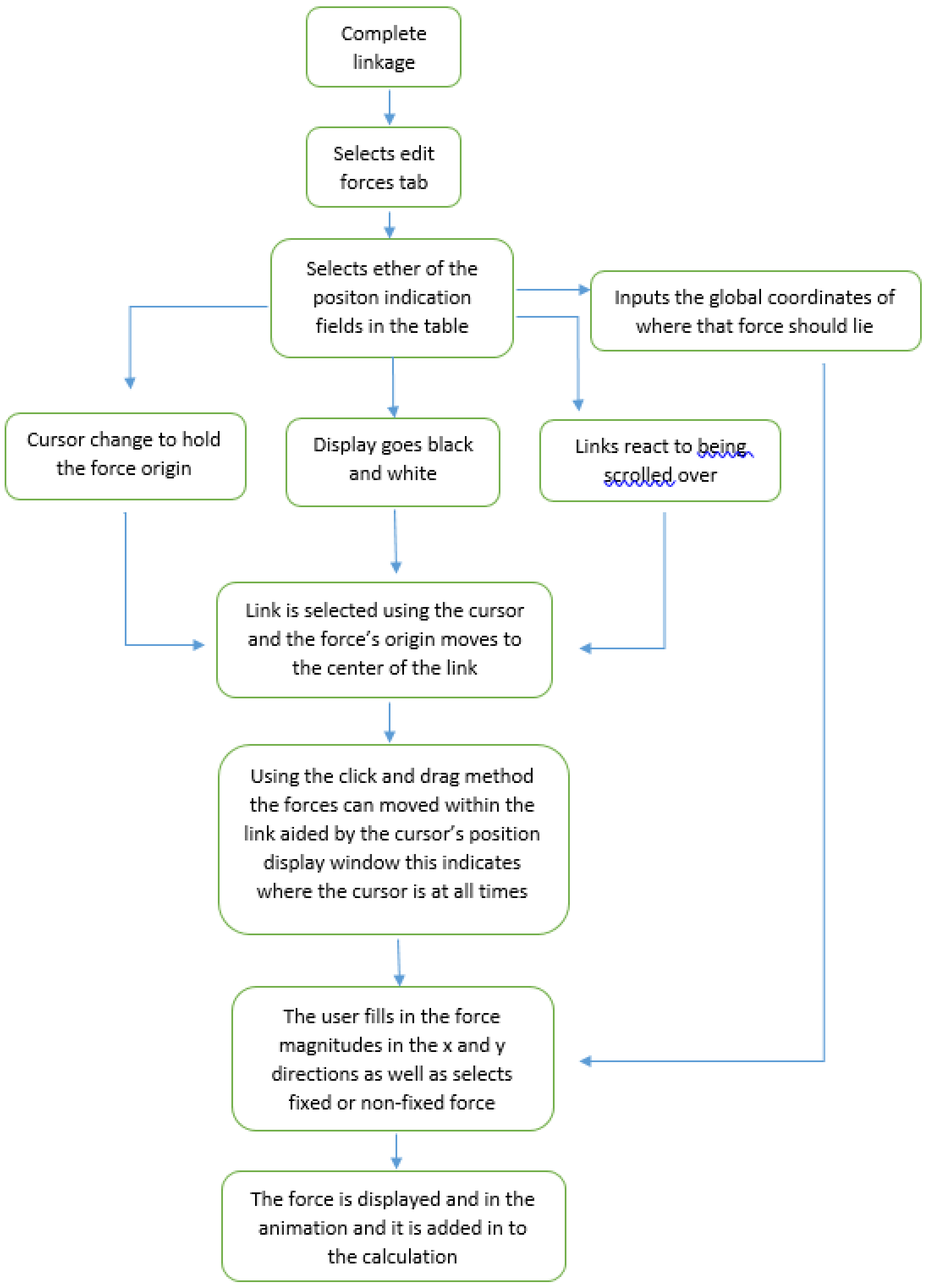
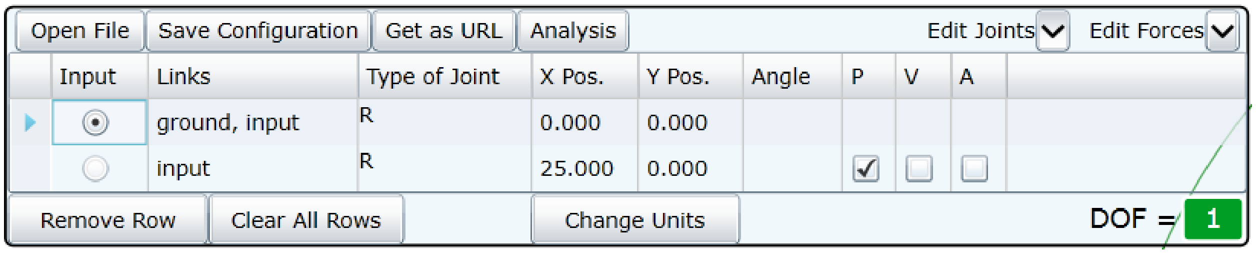

Complete linkage
Selects edit forces tab
Selects ether of the positon indication fields in the table
Inputs the global coordinates of where that force should lie
Cursor change to hold the force origin
Display goes black and white
Links react to being scrolled over
Link is selected using the cursor and the force's origin moves to the center of the link
Using the click and drag method the forces can moved within the link aided by the cursor's position display window this indicates where the cursor is at all times
The user fills in the force magnitudes in the x and y directions as well as selects fixed or non-fixed force
The force is displayed and in the animation and it is added in to the calculation

Figure 4.24: Flow chart showing all the steps required of the user to apply a force on a link.

4.3.2: Link Creation

Link creation is not difficult however, the biggest issue is understanding the edit joints tab. To fix this issue it was decided that a small reorganization is all that is needed. Which requires adding a column that shows the name of the link, with the next column over showing all the joints connected to that link. This update will fix a few things; (1) it will increase user's understanding of what they are editing. (2) It allows the user to label all of the links and joints so when the animation is paused the display can label all of the links and joints and help the user get a better understanding of what is going on.



Input	Links	Type of Joint	X Pos.	Y Pos.	Angle	P	V	A
●	ground, input	R	0.000	0.000				
○	input	R	25.000	0.000		✓	☐	☐

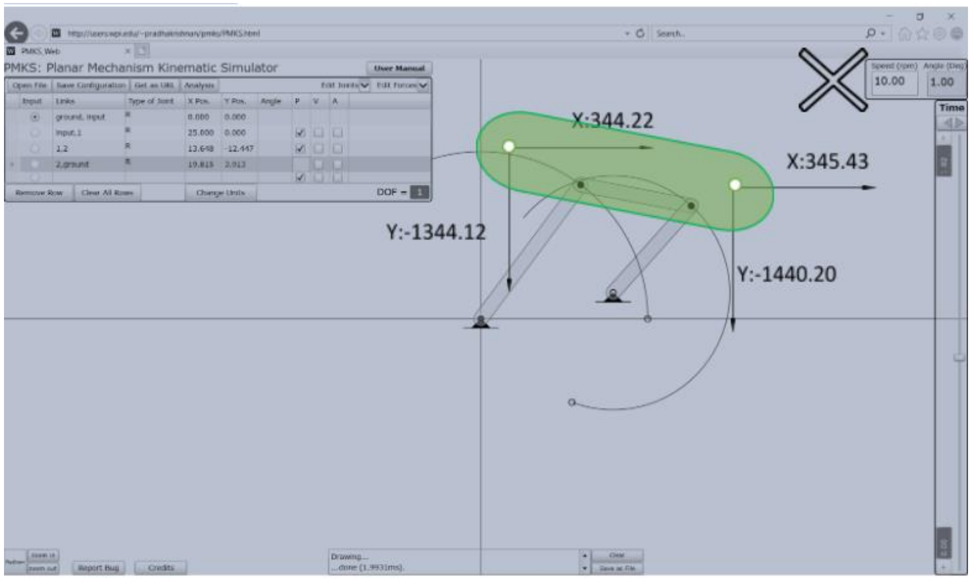
Figure 4.25: The original spreadsheet for entering data

Input	Link	Joint	Type of joint	X Pos.	Y Pos.	Angle	P	V	A
	ground	1	R	0.0	0.0				
	input	1	R	0.0	0.0				
		2	R	25.0	25.0				

Figure 4.26: A representation of the new edit joints table.

4.3.3: Free Body Diagrams

Free Body Diagrams (FBD) are going to show the users the magnitudes of forces throughout the linkage at whatever point they are requested. One functionality that has not been mentioned yet is the updated play/pause buttons. Currently if the play/pause button is selected it resets the animation to first position, then with the second click it pauses the animation. However, this is not very useful for the user. Breaking these two functionalities apart and making a button for each will be considerably more effective, (*i.e.*, having one reset button and another play/pause for the animation). With two separate buttons the user could pause the animation wherever they want if they are in the wrong place they can use the time slider to be more precise. With a paused animation the user can select FBD button from the main menu where the tabs are for the edit joints and edit forces. The display will change causing the colors to fade to black and white and the links and joints to react to being scrolled over by enlarging. This allows the user to know exactly what they are selecting and once they have selected the forces acting on those links or joints they will be displayed as vectors at the points where the forces are acting with the vector's magnitudes displayed in numbers. The user can only select one link or joint at a time so if they select one link and then another the firsts FBD will disappear; this is to keep the display clear. To unselect the user only needs to select anywhere else on the page. The user can choose between the resultant forces acting or to have the x and y forces displayed.



Input	Links	Type of Joint	X Pos.	Y Pos.	Angle	P	V	A
●	ground, input	R	0.000	0.000				
	input,1	R	25.000	0.000		✓	☐	☐
	1,2	R	13.648	-12.447		✓	☐	☐
	2,ground	R	19.815	2.913			☐	☐
						✓	☐	☐

Figure 4.27: Mock up for a Free Body Diagram Functionality

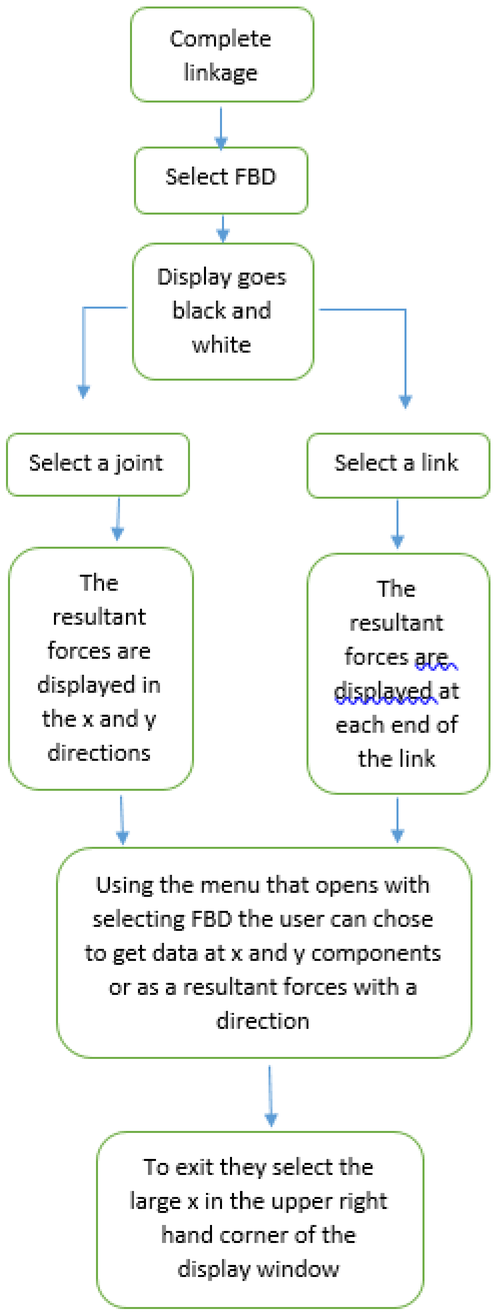

Complete linkage
Select FBD
Display goes black and white
Select a joint
Select a link
The resultant forces are displayed in the x and y directions
The resultant forces are displayed at each end of the link
Using the menu that opens with selecting FBD the user can chose to get data at x and y components or as a resultant forces with a direction
To exit they select the large x in the upper right hand corner of the display window

Figure 4.28: Flowchart of the Free Body Diagram for that linkage

4.3.4: Charting Links and Joints

The charting functionality is dependent on the new edit joints table because it allows the user to name all the parts on the screen. With labels for everything, the user can have PMKS created graphs and charts of whatever they may need. This functionality is all to minimize the user's need to export the large volumes of data PMKS creates. To activate this the user will pause the animation and select the link or the joint in question. Using the right click on either the animation or within the table they will select charting that will open a new window that controls the various options of chart or graph type and the data they want for the x and y axis. Creating these graphs will visually represent the forces acting over the full rotation of the animation.

4.3.5 Section Conclusion

By updating the users experience, PMKS can be better utilized in the education sector as well as in industry. Having a good user interaction will change how people think and feel about this program, which can have a profound effect on usage.

Chapter 5: Discussion

5.1 Accomplishments

There were three tasks that were accomplished: (1) determining linkage singularity detection, (2) debugging the desktop application, and (3), researching and designing new user interface features.

5.1.1: Determining Linkage Singularity Detection

Another accomplishment was creating a program that will determine if there is a singularity in the linkage.

This is so important because at a singularity point the matrix equation commonly used will no longer be usable. With this need program PMKS will be capable of accounting for the singularity point.

5.1.2: Debugging the WPF Desktop Application Version of PMKS

Another accomplishment was the progress made on converting the Silverlight web application version of PMKS to the WPF desktop application version. At the beginning of this project, the user could not click anywhere or do anything without crashing the application and throwing a ton of errors. Now, the user can see and move joints and interact with much of the software. Through debugging, we were able to figure out what needed to be changed to recreate all the functionality that the Silverlight web application version of PMKS provides.

5.1.3: Researching and Designing new User Interface Features

The final major accomplishment for this project was the research and design of new user interface features. These features include new force creation link creation, Free Body Diagram functionality and charting capability.

5.2 Problems Faced

PMKS possessed many problems for the computer science teams. There was a lack of documentation that made working on the code very difficult. The past project team had been using the same names for different parts of the code by having more than one namespace. On the user interface side, the problems came from trying to satisfy the data requirements PMKS needed while simplifying the users experience. Similarly trying to display more data without making the user need to search through every bit PMKS created.

5.3 What Still Needs Work

- Further debugging the desktop application
- Implementation of user interface changes

There is still much to be done before the new WPF desktop application works as the old Silverlight web application does. In addition to that, the new UI features still need to be implemented after the desktop application is working properly.

5.4: Singularity Check

The modified version of PMKS runs respective versions of singularity check for user-created four-bar and six-bar linkages when the analysis button is pressed. The algorithm was proved by testing to be functioning as expected, but several issues still need to be addressed or require successive work.

5.4.1: Limitations of Singularity Check for Six-Bar Linkage

As far as we are aware, the implemented method for checking the presence of singularity in six-bar linkage has two limitations:

1) This approach checks the presence of singular position for each four-bar subsystem individually but does not take into account the interference between them. Since the two subsystems will affect each other's motion, which adds additional limit to their range of movement, the singular positions predicted for one subsystem may not be actually reached by the six-bar moving as a whole.

Nevertheless, the incentive for identifying singularity is to prevent program from crashing, which will happen when it tries to analyze a singular position, so over-predicting does not hurt this purpose.

2) Currently, this algorithm only works for six-bars with a ternary follower (i.e. for the singularity check to be correct, the six-bars must have its ternary link to be one of the followers); it does not work for six-bar with a ternary crank (**Figure 5.1**) or a ternary coupler (**Figure 5.2**).

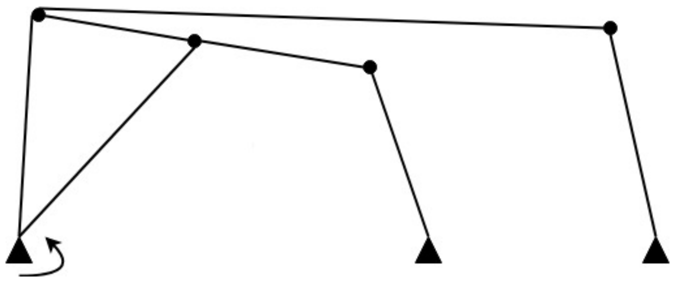


Figure 5.1: Six-bar linkage with a ternary crank

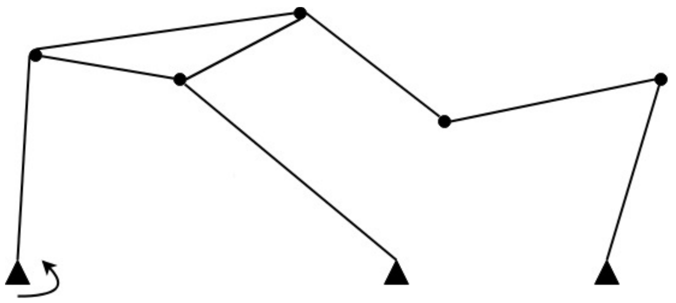


Figure 5.2: Six-bar linkage with a ternary coupler

In the case of ternary crank, the six-bar still consists of two four-bar subsystems, so the algorithm for identifying the corresponding links for each subsystems and conducting singularity check will be similar to the current version, Maybe there exists a more general algorithm for identifying the links, which will greatly improve the extensibility of the code.

On the other hand, when the six-bar has a ternary coupler, one of the subsystems will be an single-input five-bar, which has two degrees of freedom. Identifying the presence of singular position for five-bar linkages turned out to be a tricky task, and by the time of this report, we had yet figured out any solution for it.

5.4.2: Identifying singular positions during analysis and taking corresponding actions

The next step for solving the singularity problem, which has not been implemented yet, is to check if the linkage position being analyzed is singular. This can be achieved in two ways:

1) PMKS can check if the coupler and follower have the same slope before each linkage position is analyzed. As described in section 4.7.2, this approach is straightforward but limits the range of action a user can take in advance.
2) PMKS can calculate the crank angles at singular positions after a linkage with singularity is created and check if the actual crank angle is equal to that value during analysis. Crank angles can be calculated by using the cosine rule. Although PMKS stores the current angles for links, this functionality was flawed at the time of this report. Nevertheless, link angles can be easily calculated by using coordinates of joints.

In the future, additional functionalities can be added to allow the user to take a range of actions after PMKS detects the existence of singular position in the linkage and warns about it. Besides modifying the linkage, user may, for instance, choose to specify a range of angles around the singular positions for PMKS to skip during linkage analysis, since static and kinematic data gets less representative as linkage approaches its singular positions. If the user has given no instruction, PMKS should take certain action against singular positions by default (e.g. by skipping them during analysis).

5.5: State of the Software Now and Future Insight

As it stands, the WPF desktop version of PMKS has no dynamic animations working. Static shapes, like joints and the ground link are being displayed in the application. The code that does the mathematical analyses on the user created linkages was corrected and expanded upon. The Silverlight web application continues to work when ran with Visual Studio 2015 and the web browser used is Internet Explorer. The WPF desktop application no longer crashes on first interaction, and joints are now displayed and movable. Static shapes have been fixed, which leaves dynamic shapes to need future work. The animation in PMKS needs to be rebuilt from the original Silverlight version to the new, more feature rich WPF version. There are a lot of changes between how Silverlight handles animations and how WPF handles animations. The way Silverlight handles it is much less complex, while WPF has many more features and is purpose built to be able to support complex animations. To make the codebase more maintainable and easier to understand, a redesign of the way PMKS animates dynamic shapes is necessary.

5.6: User Experience

Throughout this project we have had numerous tasks and several roadblocks. Our first focus was taking an existing web application and making a desktop version of the application. The next focus was to redesign that program so that it could create mechanical systems on its own. Those designs would be sent to a 3D printer and be rapidly prototyped. These tasks were broken into smaller segments: creating the desktop application, correcting calculation errors and fixing code errors. Within creating the desktop application there was a smaller focus of addressing the user interface problems.

For a new user this program had a very steep learning curve as most programs like this do. This program was modified with several roles in mind: it was intended as both a teaching software and an industry application to design systems. Having this dual role made it more difficult to design for the end user. It was determined that simplicity would be the most important aspect of the user interface redesign. There are two major problem areas within the user experience: data input and data retrieval. On the data input side, there were problems with lack of data specificity to assist in the link creation. Additionally, there was great difficulty locating the correct point to identify the force creation. In addition, on the data retrieval side the major issue was the massive quantity of data that PMKS would create an that needed to shorten the time and action from having data to finding the key data in question. Therefore, redesigns were created for all of these issues however, without successfully creating the desktop application those redesigns were never implemented. Instead, they have been vetted, detailed and recorded for the next team to work on and this way the next team does not need to reinvent the wheel**.**

Chapter 6: Conclusion

6.1: Accomplishments

While the team ran it to its share of road blocks, there were a lot of accomplishments. Work was done on creating a program that can determine if there is a singularity and that give PMKS the capacity to prepare for the matrix equations to fail. The CS team worked on the user experience and troubleshooting the desktop application. The CS team also made significant progress on converting the Silverlight web application version of PMKS to a WPF desktop application. In this field four user interface changes were documented for the next team to implement. Moreover, the desktop application troubleshooting helped in cleaning up the code and creating documentation for the next team.

6.2: Major Roadblocks Faced

The CS team ran in a multitude of roadblocks as a result of the lack of documentation and communication from the last team. Working on the desktop version PMKS turned into a one step forward five steps back as more and more was understood about the software that was handed to the team. The CS team also ran into issues with the way animations were handled in Silverlight compared to WPF. One of the hardest things we had to overcome was having only one CS major on the team for a project that is entirely code.

6.3: What was Learned

The CS team learned about the importance of best practices when writing code. The uses of best practices, documentation and diagrams allows code to improve in iterations. Without these tools every new sent of eye needs to start from scratch to creates a well running program. Moreover, the importance of creating simple and easy to understand user interface. The designer needs to take into account the users and what they can be expected to know then create an interface that works with their understanding to hold their hand as they learn how the program works.

Working on PMKS has been an enriching experience that helped all of the three team members in different ways. The computer science major needed to widen his depth and breadth in statics, dynamics, and kinematics. All of the team members learned how to better function in a team and how to better communicate their ideas to one another. Moreover, it was a teaching experiences in perseverance in handling the many tasks and the problems they presented. The team ended this project more humbly than they had started, and with a greater appreciation for the programs and tools used daily on the world’s stage.

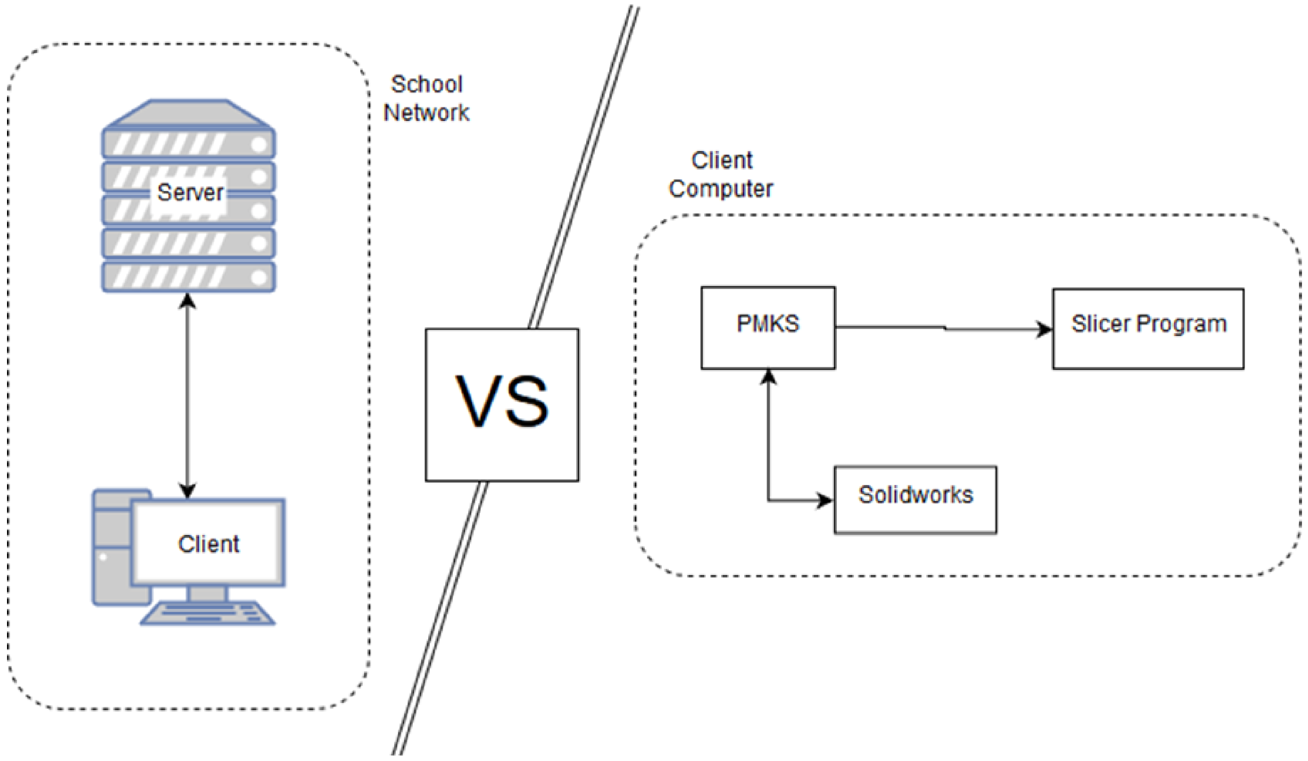
References

[1]: Saldana. “Technical Documentation, API, and Code Examples.” *Technical Documentation, API, and Code Examples*, docs.microsoft.com/en-us/.

[2]: “Planar Mechanism Kinematic Simulator.” *PMKS: Planar Mechanism Kinematic Simulator by DesignEngrLab*, designengrlab.github.io/PMKS/.

Appendices

Appendix A: Silverlight version of PMKS vs WPF version of PMKS



- The Silverlight version of PMKS runs on a client-server architecture
    - Allows mass distribution and use
    - Limits software complexity
- The WPF version of PMKS runs as a local desktop application
    - More overhead
        - Need to download program
        - Local computer needs to be fast enough to run it
    - Increased software complexity
        - Allows for cross application communication
        - Can be more graphically intense with animations