CS 3307A - Object Oriented Design And Analysis

Group 29

December 6, 2021

Calculator PI

Project Postmortem Report

Michael Dias, Nieve Wong, Mihir Kadiya, Michael Scantlebury, Brandon Howe

Project Summary

We have created a “smart” calculator that accepts and solves various math problems. The calculator recognizes the “type” of the question being asked and performs the appropriate calculations. Our program has a graphical interface that allows the user to input symbols and representations beyond what a keyboard allows. We made our program modular and scalable, starting by allowing it to recognize and solve simple math equations before continuously adding more functionality, increasing the types of questions it can solve. This included trigonometric equations, including sin, cos, trigs, and other functions and operators such as logs and factorials. All these would ultimately adhere to a defined grammar, from which a tree would be created to define the order of which equations would be solved. From there, another class of objects would be used to solve the problem using the format established by the tree before displaying it with a user interface. From there, we added more functionality in the form of graphing, solving linear systems of equations, matrix calculations, and determinants, using the past code as a base to build from.

Key Accomplishments

The main design of the project was split into three parts: Model, Controller, and View. Dividing the project into three parts helped with collaborating more effectively by individually working on separate parts of the project in parallel and then connecting all the components in the end.

The model, or the backend, was the main computational part of the project, which processed all the user input and gave back a valid output. From a design perspective, the classes followed a very linear relationship which made the software flow very clear to understand and increased code readability. All the function classes (Constant, Log, Trig, Polynomial, Fraction, and base Function) took advantage of inheritance remarkably well with dynamic casting. The lexer did not have any problems, and it was easily extensible to support more tokens whenever needed, which worked out very well for the parser. The parser was responsible for creating a binary tree from the tokens it received from the lexer. This was an essential class in the project as many other parts of the model used the binary tree to carry out various operations. So, this class was thoroughly tested since many other classes relied on it, and it was a huge accomplishment to make this class fully functioning. The interpreter is also considered one of the crucial classes in the model since this class used the set of Function classes to group all the tokens and then solve the binary tree. Since this class heavily relied on all the Function classes, two group members worked in a pair to implement both the Interpreter and the Function classes together, resulting in more effective communication and time management. The matrix, set, and factorial classes make up the other parts of the model which did the computation for other parts of the project. These classes worked very well with object-oriented design principles and accomplished their functionality precisely as required.

The controller's purpose was to allow the model and the view to interact with each other. Managing the middle ground of both components, it was responsible for initializing the process of computing certain equations before returning the results to the view where it was ultimately displayed to the user. It accomplished this by using a button responder, which is activated when the user pushes a “solve” or “graph” button in the view after entering an equation to solve/graph. Once this happens, it establishes the compute functions within its area, interacting with the back end, otherwise known as the model part of our project, to perform and calculate the necessary solution to the problem. It then returns a solution to the button responder class catching any errors that may have occurred within the process of solving the equation. From there, the button responder sends the acquired information to the view where it is displayed.

The view or front-end of our project was the main interface for users to interact with our program. Through this interface, users can input various mathematical problems and obtain their solutions and steps. We used a main tabbed window with tabs for solving expressions, matrices, sets, and graphs. Using a tabbed window allowed us to split the types of problems up into distinct visual areas and have separate input windows for each type of problem. This allowed us to create different ways for users to input their questions and made it easier to display the output in various formats such as a matrix or as a single-line expression. Using the singleton design pattern, we were able to keep the history of all calculated expressions in the expressions tab, which users could reference at any point in time. Having separate classes for the expression, matrices, sets, and graph GUIs made it easier to debug the front-end of our program and identify which class problems were coming from. A problem in one tab of the window did not affect the function of the others. Using separate classes also made the front end easily extensible as no previous code would need to be altered. If we wanted to handle new types of problems in the future, all that would need to be added is a new GUI class for that problem and a new tab for that GUI in the main tabbed window. Using the QErrorMessage class gave us a simple method to catch all user input errors and allowed us to display an error message on the main window to the user.

Key Problem Areas

One major problem we faced was how we dealt with storing the functions. When we started planning to have our program handle multiple types of functions, we knew we wanted to store the data relating to each type of function in their specific classes, but somehow group them mutually under a parent function class. Moreso, we wanted to have the functions be able to interact with each other without always needing to know what type the other function was, to macOS with doing operations between them. Implementing this, however, proved to be a large challenge, as we went between many different c++ data structures to get this to happen in a way that suited our needs. We tried using inheritance, templates, abstract classes, virtual methods, and different combinations of those data structures. In the end, simple inheritance with virtual overloaded methods did the job, and we were able to have other classes, namely the interpreter, use the different types of functions without always needing to know what type of function it was dealing with, which saved on a lot of redundancy.

Another problem we faced was the impact of our incorrect story point estimates. For many of our program's functionalities, we underestimated how long it would take us to implement them. As a result, we tried to include more functionality than we could implement in the time we had. To mitigate this problem, We shifted our focus away from our “wish-list” items and towards the essential areas of our program. Ultimately, we cut out certain functionalities of our program such as detecting mathematical expressions from an image, graphing key features, and simple differentiation.

An additional issue we had was getting the button responder and output formatter to retrieve information from and display information to the GUIs. We ran into a problem where a GUI needed a reference to an output formatter, but an output formatter also required a reference to a GUI, which caused compilation errors due to circular includes (we think). This problem was exacerbated by the fact that we didn’t get to coding this part until the last couple of days of the project. Due to time pressure, we decided to move the functionality that the output formatter was supposed to provide into the GUI’s. Some functionality of the button responder was also moved to the GUI’s for the same reason.

The controller classes followed the strategy design pattern to connect the front end and the back end in a meaningful and readable manner. However, the design pattern involved working with header files unique to c++. The use of “virtual” was tough to understand and apply properly in the project. Working together on this problem and debugging through all the possible errors resolved this problem.

All the group members worked with different IDEs for this project and were also on different Operating Systems. This caused many issues when the project was not running on all of our computers during the development. For example, many errors were caused by qmake and how it worked on Windows and macOS. Eventually, towards the end of the project, when it was time to put all the code together, we started to codeshare with CLion and work from the same computer.

Lessons Learned

One hard lesson we learned was about postponing critical functionality. We chose first to develop the backend and frontend, then develop the controller to connect the two. This allowed each of us to work on different parts of the code without interfering with each other. However, this came at a cost: when it came time to implement the controller, there were many unexpected and time-consuming setbacks which ultimately led to an incomplete product at the time of the deadline. If we could do the project differently, we would implement the essential controller functionality as soon as the front and backend of a calculator were developed, rather than starting development on the next type of calculator (more agile oriented approach). Having all the group members on the same IDE is very important to mitigate many of the crashes that might occur.

Throughout the project, we gained very specific insights into software development and general insights about collaborative projects. One of these insights was a realization of how code can be applied to very specialized problems and the niche techniques and practices that are out there for solving these problems. For example, to implement the functionality desired, we needed to familiarize ourselves with concepts such as grammars, parsing, and evaluating parse trees, as well as study and implement algorithms for matrix reductions, inversions, and determinants. We also took away lessons about collaborative coding projects. For example, we used techniques like paired-programming in the more technical parts to avoid mistakes and save time. Also, we learned how to communicate as a team by messaging each other with questions we had and having meetings to discuss the present project state and our future goals and limitations. Towards the end, we had more voice meetings rather than text messaging because at that point our program was interconnected, so it was helpful to have the authors of each part of the code together to complete the remaining work.