# **Emergency Calculator**

Developers: Dylan Engle, Eric Bunch, Weston Abner

Introduction

There are many scenarios in which one can forget the tools necessary to achieve their goals which generally can end in disaster. Much like a carpenter needs a measure or a chisel, Stem students need a calculator to do the work required of them in a timely manner. This led to the creation of the Emergency Calculator that provides many key components that can increase the student's quality of life. It was designed to have the functionality to evaluate the functions given to it, share those functions with other users, graph the evaluated functions, and even print those graphs into a jpg or png format. This would solve a lot of the problems that STEM students find themselves in and help them to save time when working alone or with a group. The Emergency Calculator is a one-stop-shop for students with mathematical needs.

Technology

The need for the application to be used in emergencies and other situations in which one might not have access to their usual tools led to the design choice that the emergency calculator would be a web application in order to be used anywhere and have the ability to expand to multiple devices and browsers. With the web application in mind there are a few technologies that were necessary in order to bring this project into fruition. For the web browser side this led to using the normal technology that is utilized such as HTML, CSS, and Javascript. Each of these technologies is paramount to the core design of the application. Other needs still remained to really complete the project. To have the ability to save the functions and other such evaluations required to transfer data between students meant that the application was in need of database technology. This led to the utilization of Nodejs and mySQL to have a comprehensive ability to have a database that would manage these expectations as well the ability to interact with JavaScript using the NodeJS technology. Due to the lack of javascript experience on the team we were tentatively open to using Typescript in order to leverage our C# experience and hopefully allow for an easier transition into using javascript. We also felt that using Docker could be a way of hosting our services especially in the debugging phase as it could work as a temporary way to test functionality.

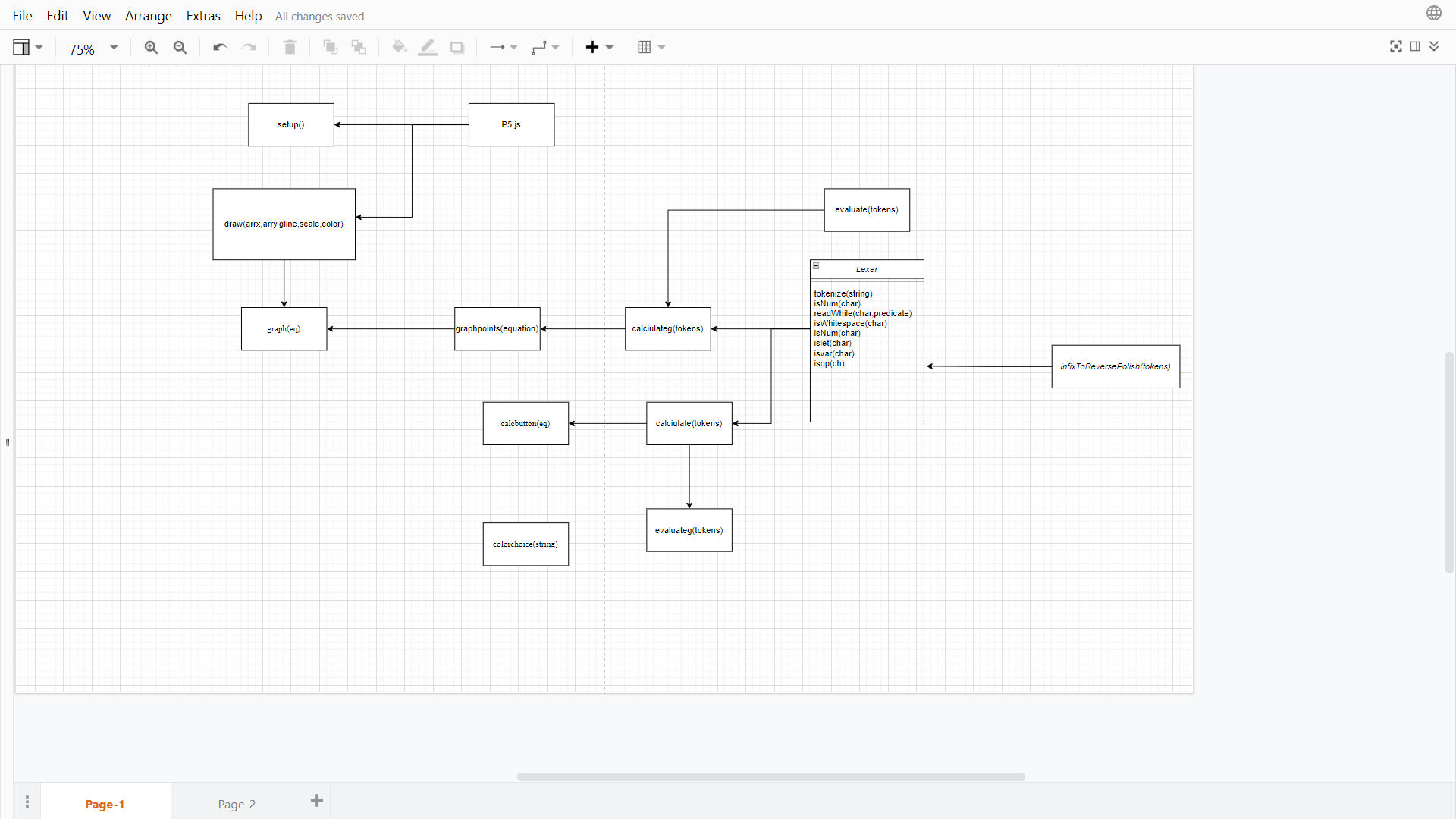
JavaScript, html5, and CSS were the correct programming languages for this project. This was a web-based calculator and html5 and CSS were used to great effect to make the project look professional. JavaScript was the correct scripting language as it allowed us to parse user input, draw on a canvas, do basic math along with trigonometry, and had libraries that reduced the amount of code our team had to write from scratch. It also allowed us to use the same language for both the server and the scripting an advantage that most languages would not give our team. The only issue with JavaScript was that our team was not very familiar with the language which led to many mistakes that would not have happened in a language that we were more familiar with. Due to us becoming more antiquated with javascript as the program evolved we noted that we had no use in using typescript as it would be redundant in that it gets translated to javascript proper when compiled. Instead we persevered with Javascript and learned enough to be able to utilize it fully within the project.Another wall from a former technology that we felt invested into was that of Docker. It was something the team was ill experienced in and we found that trying to leverage it as a way of hosting our project was going to cost precious time that we could not afford in the developmental process. Scrapping Docker allowed us to move our focus on to creating a minimally viable product instead of fighting technologies we mostly had no experience in. This turned out to be the correct choice with our time scale considering that there were features that still ended up not being functionally complete at the deadline and Docker would have been a huge hindrance in that progress.

The p5.js library was the correct scripting library for the job mostly because it is the most used one that works with JavaScript. This allowed for plenty of resources to get up to speed with this library. A different library might have also worked but the number of resources available for it would have been at least slightly smaller. This library provided any tools that our team found incredibly useful. It provided a better canvas than the default html canvas for starters that could be easily saved and exported for the users. It provided a shape function that could make lines from a series of points making drawing mathematically correct graphs simpler. It provided a stroke function which could adjust the thickness and color of the lines and the color could be in a string making it easier to adjust the color through user input. Most importantly it provided a draw function that would draw on the canvas meaning we didn’t have to assign a html canvas and use that instead. This library was slightly difficult to get up to speed on because the team wasn’t very experienced with JavaScript but the functions within the library functioned as normal in JavaScript so a team more familiar with the language would have an easier time getting up to speed with it then we did.

Overall technology quick list:

* Javascript
* Html
* CSS
* Xampp
* MySql Database
* VSCode
* Apache Server
* Node.js Express server
* P5 Library

Design



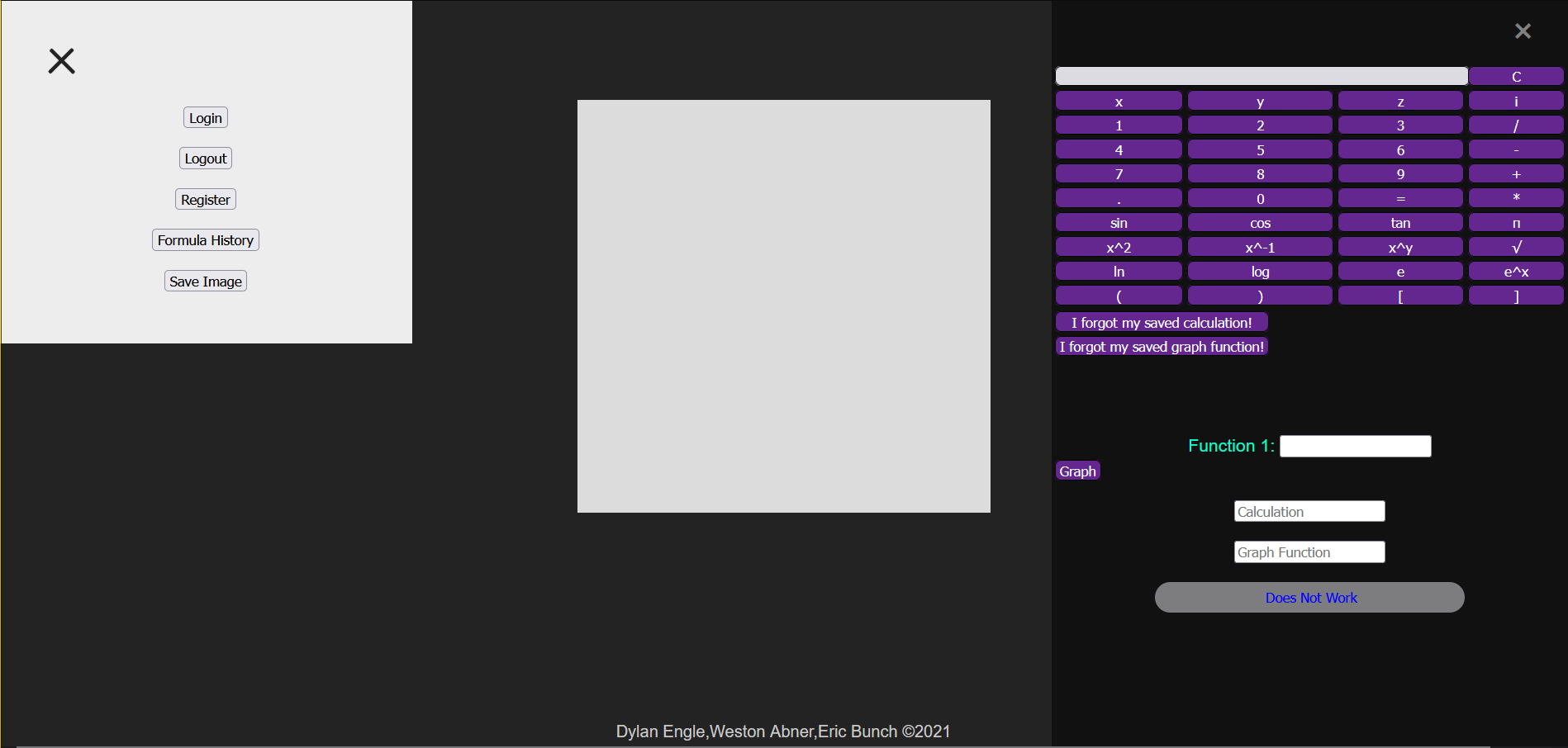
This function diagram can also be found <https://github.com/ebunch99/EmergencyCalc/blob/main/docs/emergancy%20calc%20function%20diagram.drawio>

The calculator requires a myriad of functions to function properly. It also requires one class lexer. Lexer is composed of one main function tokenize which tokenizes user input for later use, and some support functions that allow tokenizer to tokenize. Most of these functions are predicate functions. The first of these predicates is isWhitespace which takes a char checks if its white space or not and then returns true or false based on the results of the check. The next is isNum which checks if the char is a number which is defined as 0-9 or “.” with “.” being used to allow the calculator to accept numbers with decimals in them. Along with isNum there is islet which checks for the letters a-z. This function is used to check for non basic math functions like ln or cos along with another function that will be discussed a little later. After islet we have isvar which checks for one letter variables that we decided to include. Currently there are only three variables it is checking for those being x, e, and π. The next function is isOp which checks for the basic math operators of (), \, -, +, and \* standing for parenthesis, division, subtraction, addition, and multiplication respectively. It also checks for ^,%, and √ standing for power, modulus, and square root respectively. The only non-predicate helper function defined in the Lexer class is readwhile. Readwhile is like the readwhile functions in other languages except it is tuned for chars and predicates. It takes an array of chars and a predicate, sets a string to an empty string and then runs through a while loop that goes through the array until the predicate is no longer true or the char array reaches the end of the array. As long as this while loop is running it adds chars from the chars array and then when the while loop ends it returns the new string. The final function in the lexer class is the tokenize function. This function uses all of the support functions described earlier. It takes user input in the form of a string and turns it into an array of chars, and creates an empty array of tokens to store the tokens that will be created. Then it starts a while loop that runs until it's past the length of chars. It then uses the readWhile function stated earlier to go through all the white spaces. It uses an if statement to make sure it doesn't go past the length of the array with the readWhile function. It then assignment the first char from the array to a variable and runs the variable through a multitude of if statements checking if its a number using isNum and if it is it uses string addition and the readWhile function to get all of the numbers after it to make sure it gets full numbers and nt just part of the number. It then pushes this number plus the type NUM to the tokens array to be used at the end of the function. Next it checks if the value is an operator with the isOp function and then if it is an operator it pushes it to the tokens array with type OP. The next if statement checks if it's a variable and if it is it pushes it to tokens with type NUM. The final set of the if statements checks for letters with islet, uses the readWhile function to combine all of the letters into a variable the same way the numbers are, and then pushes it to the array with the type OP. This if statement is added specifically to work with functions that use words like ln or cos. The final part of the function applies the infixToReversePolish function to the tokens array and returns the array. This function converts a string from infix notation to reverse polish notation which is the function that will be discussed next.

The next function to be discussed as stated is the infixToReversePolish function. This function takes an array of tokens. The beginning of this function creates two empty arrays called stack and queue. These arrays will be used like a stack and queue respectively. The next thing this function does is set precedence to be used later. The precedence is ( followed by + and -, followed by /, \*, and %, followed by ^, followed by the other functions with the final priority operators in sqrt, tan, cos, ln, log, and √. Note these all have the lowest priority as they are expected to be used (sqrt 10) instead of sqrt(10) to make it easier to program. This is stated in the user's guide. After we set up the precedence we run through a shunting yard algorithm. A shunting yard algorithm can be described in a few easy steps. First you look at the next token in the array, or first token if it's at the start of the array. If there are no more tokens empty the stack into the queue. If the stack is not empty check what the next token is. If the next token is ) empty the stack into the que until you reach a ( and then discard both of the parenthesis. This is done because if parentheses are put into the array out of order it will mess everything in the algorithm up as parenthesis are very important in mathematical equations. If the stack is both not empty and the top of the stack is not ) check to see if it's a number. If it is a number, push it onto the queue and then continue the loop from the beginning. For our algorithm number means anything with the NUM type so this includes variables that we have set as we treat them the same since they will be turned into numbers in the next function that will be used. If it is not a number then it is an operator and we have to do some checks before we can put it into the stack. The function first checks if the stack is empty. If the stack is empty we don't have to worry about it and can push it to the stack and then continue the loop. If the operator is parenthesis, since we already have a check for parenthesis we can also immediately push it into the stack and continue the loop. If it is something else we have to check what is currently in the stack. If the current operator has a higher precedence value than the one currently on the stack, push it on the stack and continue the loop. If the precedence is lower than the precedence of the operator on the top of the stack, then pop the stack and push the value into the queue, then continue checking until you can put them onto the stack in precedence order. After you go all the way through the loop, return the stack.

The next function required for the calculator is the evaluate and the evaluateg functions. These functions also take an array as input. They will be almost the exact same. The two differences between these two is that evaluateg also takes a variable v and it has 1 more if statement that will be discussed when we reach that point of the function. At the beginning of the function we create 3 variables for use in the function. A stack which is an empty array that will be treated like a stack like the stack from earlier, and rhs and lhs which are both set to zero. We start another loop that goes until the array is empty. We first check if the token is a NUM. If it is a NUM we check what kind of NUM it is. If it's the pi symbol then we push Math.PI into the stack and continue the loop. The Math function calls onto the Math object in JavaScript which is used in multiple ways in this function. If it is e then we push Math.E into the stack and continue the loop. There is a final if statement in the evaluateg function where if the NUM is x then the variable given to evaluateg will be pushed into the stack to allow the function to solve for y when being used by the draw functions which will be discussed later. If the token in the top of the array is an OP instead of a num we check what type of OP it is. If it is tan, cos, sin, sqrt, ln, or log then we pop the stack into rhs. If it is not one of these OPs then we pop the top of the stack into rhs and pop it again into lhs. The next part of this function is run by a switch statement. This switch statement has a lot of cases that we will go through next in a very set way. We will state the OP and then what happens from that OP and then it breaks out of the switch and continues the loop. + adds lhs to rhs and then pushes it into the stack. - subtracts lhs from rhs and then pushes it into the stack. \* multiplies lhs by rhs and then pushes it into the stack. / divides lhs by rhs and then pushes it into the stack. % moduluses lhs by rhs and then pushes it into the stack. ^ puts lhs to the power of rhs and pushes it into the stack. This is the first of the switch cases that uses the Math object from JavaScript in this case Math.pow. The next set will be the trig function in which the number rhs is multiplied by Math.PI and divided by 180 to get the answer in degrees. This will not be stated each time and will just be stated as Math.function on rhs to make it simpler to state. Tan uses Math.tan on rhs and then pushes that into the stack. Sin uses Math.sin on rhs and then pushes that onto the stack. Cos uses Math.cos on the rhs and pushes it into the stack. Sqrt or the sqrt symbol √ both do Math.sqrt on the rhs and then pushes the answer onto the stack. Ln does Math.log rhs and then pushes it onto the stack. Log takes Math.log and divides it by Math.log 10 to find the log of a number on base 10. To adjust this function for other log bases would replace the 10 with whatever base it would need instead of 10. After the loop is done we pop the stack and return that.

Calculate and calculateg are like evaluate and evaluateg as they are basically the same outside of one function call. Calculate takes a string as an input and then runs that through the lexer.tokenize function. It then takes the result and puts it into the evaluate function and then returns that result. Calculateg takes a string and a variable in this case the x for y= functions so that things can be graphed. It then follows through the same steps except it uses evaluateg instead of evaluate.



Everything mentioned in the previous paragraphs are shown on the right side of the screen with the calculator. So all of the previous technical things are only ever called and evaluated into the very top white box in the calculator. The box with function 1 does not have the same functionality as what has been talked about. The information provided below is what is called upon typing a function into the box that says function 1 and hitting the graph button.

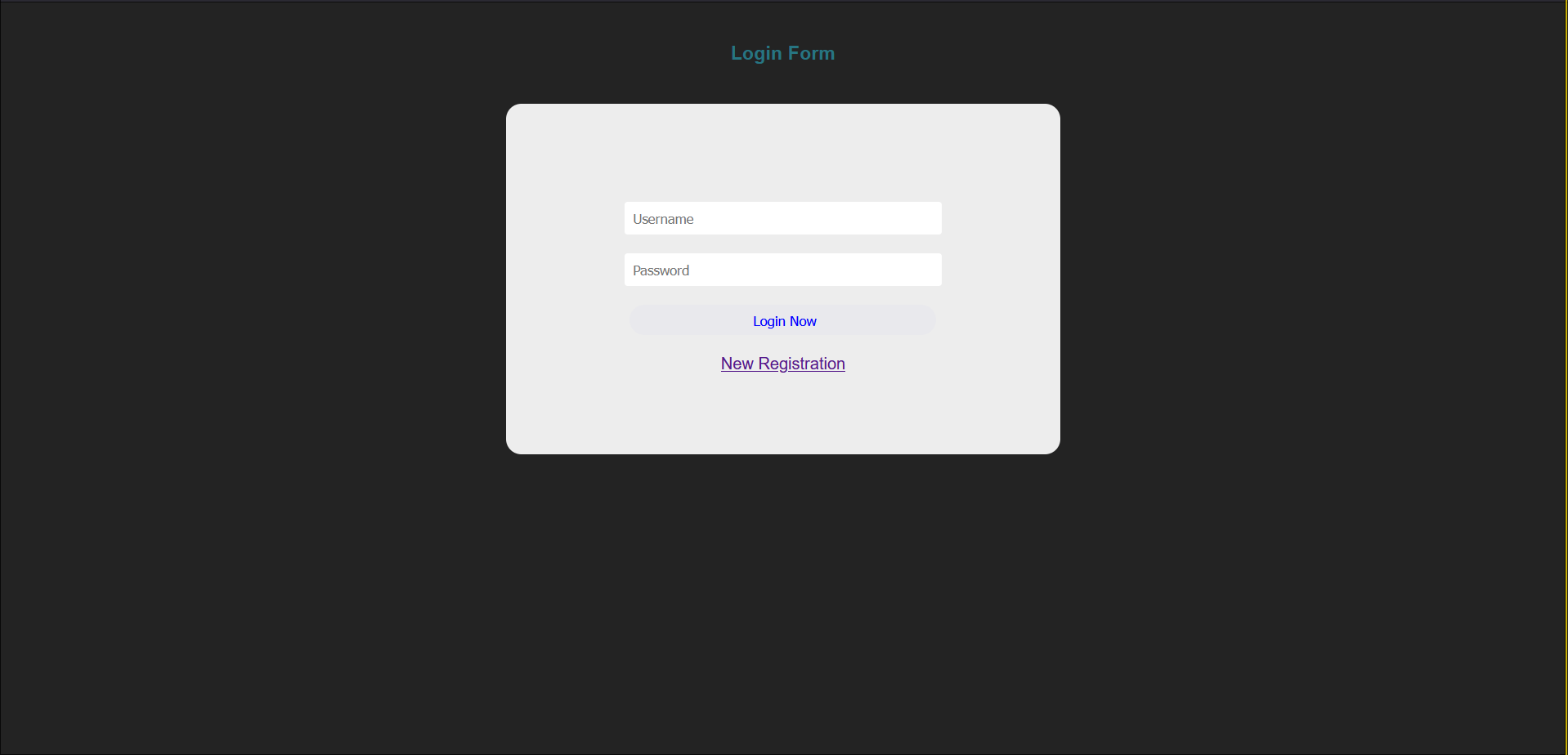
That is all of the calculator functions so we will move on to the graph functions. There are three graph functions, two of which are p5 functions. These two are the set up and draw function. The third graph function is graphpoints which does the math for the x and y spots for the graph. The first one we will discuss is the set up function. This function is very simple. It creates a canvas through the createCanvas function provided by the p5 library. We set this to 400,400. We set this equal to c so that the canvas can be saved through a save canvas function though it isn't working currently. I then set the strokeWeight to one for default which is what will stay throughout all of the functions. We also set the background color to 220 which is off white.

The second graph function we will cover is the graphpoints function. The graphpoints function requires two parameters: an equation and scale. Scale sets the total amount of lines while equation is the equation that will be graphed. At the beginning of the function we set up all the variables required for the function to work. Both x and y are set to 0, both arrx ,arry, and values are set as empty arrays. Limit is set to scale/2 limit being the total off to each side the graph will represent. So if the user wants it to display 10- -10 the scale will be 20 and the limit will be 10. The default and only usable version of this currently is 20 with the limit being as stated earlier. It is set equal to -limit as the loop will go from -limit to limit, and gscale is set to 400/scale as that is the distance between each line on the graph. A loop is then set up from i or -limit to limit to generate all of the values that will be on the graph canvas. X is set as i and y is set to calculateg the graph equation and x. X and y are then scaled to canvas by setting the variable equal to itself time gscale and then adding it to the scale number. This number is 200 for x and -200 for y. It is like this since positive y is up the canvas so the higher the number the closer to 0 it needs to be. These values are then pushed into arrx and arry for x and y respectively. These are then pushed into the values array and then returned since javascript can't have a function that returns multiple values so it returns an array of arrays to get around that issue.

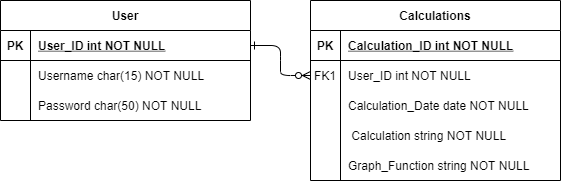
The final of the graph functions is the draw function. The draw function requires the most parameters out of any of the functions discussed so far. It requires arrx, arry, gline, scale, and color. The first line checks if gline equals true or not. If it is true then it draws the gridlines using two sets of for loops both going from 0 to the height of the canvas as its a square canvas so the height and width is the same, i being incremented by the scale value which should be the same scale value used in the graphpoints function. They both use the line function from the p5 library with the parameters 0, i, then width for the x lines and height for the y lines. The default for this function is true but the gridlines can be removed using user input. After the grid line loops we have a set of nested for loops that draw the actual lines on the graphs. The first is from zero to arrx.length. It uses several p5 functions, those being noFill which makes the graph line a line and not a filled in shape, strokeWeight which is set to one which makes the graph lines smale, stroke with the value of color i. This gives the line whatever color the user will set it to be. The final p5 functions are beginShape and endShape. Beginshape goes before the nested for loop and end shape goes after the nested for loop. These functions combine all of the curve vertices that are created in the nested for loop into one curve. The nested loop goes from 0 to arrx[i].length or the length of the array inside of the arrx. Inside this for loop it only runs one function which is curveVertex from the p5 library. This function requires a point, this point being arrx[i][j], arry[i][j] or the current x and y of the current array inside of arrx, and arry.

There are also a few button functions meant to make the buttons easier to write. The first of these functions is calcbutton, which takes user input runs it through the calculate function from earlier and returns the result. There is also colorchoice that takes user input and pushes it into a color array to be passed to the draw function. There is also a dscale and iscale that reduce and increase the scale by 2 respectively. Finally there is a graph button that runs all of the graphing functions. It takes an equation, runs it through the graphpoints function, pushes the result into an array for the xs and an array for the ys and then passes those along with the gline and scale variables and an array of all the colors.

For the backend of the project, there are quite a few things going on behind the scenes. There are no classes, however there are many different files for each page of the website. So the routes folder is responsible for the routing of the website, and each route calls upon the view (html) in order to display what is written in the html. Inside each of the route files we have the SQL code written. The http get function is what is called if we do not need to submit any information to the database, but the http post is used upon submitting information to the database, such as the registration and the index routes. Below, the login form can be seen, which is using the http get in order to retrieve information from the database to check to see if the user trying to log in is a user, and if they are a user, they are using the correct password to login. The registration page is different because it uses the post in order to send information to the database. The data that is being passed around from browser to server is getting sent through the use of forms and it does it directly.



There are only two database tables with our design. These are the user table and the calculation table. Within the user table, we have the userid, username, and password rows. Within the calculation table there is the calcid, calculationdate, calculation, and graphfunction. The file that connects with the database is the database.js file. Each route in the route folder reads or writes information to and from the database depending on if the route is a http get or http post.



Shown above is the database structure. The user table is so that the user can have information that they can log in to in order to save the calculations. The calculations that they save are based on their user id and not their username. This specifically was annoying when it came to inserting the dummy data, as I had to remember the ids of the accounts that I wanted to add information to and not cause any issues within the database.

How to deploy

In order to get this code up and running there are a few things that need to be done on the developers end for this. Firstly, you are going to need your own MySql database. In the GitHub we have provided the code to get the database tables set up exactly like ours, as well as the dummy data that is inside it. There are a few different ways of setting up a MySql database so that is up to the developer on how this can be done, but we did it using an application called xampp. Upon downloading this application you will be greeted with a screen that has buttons for a database, apache server, etc.. All that needs to be run on this is the apache server and the MySql database, as the database is being run on the apache server. In the database.js file, the developer will need to get their database information and plug that into where the database information is inside the database.js file. All of this code is in the node app folder of the github, so the database.js will be inside this folder. In this folder, under public/javascripts the developer will need to change the name of the desktop links based on the name of their computer. This will make it so when the registration and other buttons are hit, it will link to the correct page. It should say in the functions "desktop--------:3000/register" or login. DO NOT TOUCH the ":3000/register", only what the name of the desktop is that this is being hosted on. Once this is done, to get this up and running, run a command line and make the current directory the nodeapp folder. The libraries needed should be installed on the github, but just to be sure the developer can run the npm install command. After this just run npm start and the website will be deployed locally. Just go to localhost:3000 and the website should be working. To allow others on the network to access the website, just give them {the name of your desktop:3000"}. To compile all of this code the only code that should be needed is command line. The SQL code for the database is inside the github.

Known Bugs

The website has a few known bugs. The graph and calculator both share one which comes from the lack of user input validation. This leads to the evaluate functions taking anything the user puts into it. The normal evaluate function just pushes nan as it can't figure out what to do with these values. The graph function has more issues with this bug. It completely crashes when it takes a word as it can't graph nan. This also gives it issues with tan x. Normally to do tan in this program you just need to (tan) however because it reads tan x as tanx you have to input it as (tan(x)) to get it to read that correctly. These errors will both be fixed by adding user input validation in the tokenize input function.

Another bug that we have has to do with logging in. A user is able to register as many accounts as they want and log in to all of them at the same time. When the insert function worked properly, this would cause major problems because the program did not know which user to input the data into in the database. This would cause the server to crash, but because we have disabled the inserting of data it does not matter how many users are logged in.

Future Work

For version 2 of the calculator a lot of little things would be added. The first would be user input validation in the tokenize input function as this would remove bugs from both the calculator and the graph. The second is the addition of physics variables. This is something that we had wanted from the beginning but a bunch of little things got in the way of this feature. These would be added as part of the tokenize function along with the user input. Finally more normal functions would be added :logs that aren't base ten, and the inverse trigonometric functions being the top of the list.

The graph canvas is also missing a bunch of features that are mostly complete. The ones that are working from a graph standpoint but missing buttons to use are; graph colors for each graph, scale, and gridlines. Things that are not currently implemented that would be easy to create is a cleargraphs function. This function would just set the x, y, and color arrays to empty arrays to clear the canvas. The function that was working but broke is the savecanvas buttons. These buttons allow the user to save the graph as either a jpg or a png but did not work correctly with the server we were using so they ended up not being fixed in time. The final piece of functionality that would be implemented is the zoom function. This would probably have to be set as a number that the user types in to zoom in at and then use the scale functions to adjust the scale around this point. Finally, the save function still needs to be implemented.