

1. Installation instructions and userguide, documenting the functionality of your system (20 points)

To install:

1. On your command line, navigate to the main directory of this project
 - a. The main directory should be: "CS4530_team_405-final-submission"
2. Once you are in this directory, follow the basic same steps as would be followed for any basic npm app:
 - a. "npm i"
 - b. "npm start"
3. The webpage should open automatically once loading is done. If not, navigate to:
 - a. "localhost:3000" on your web browser of choice.
4. To run tests, run "npm test" in the terminal (make sure you are still in the right directory)

User Guide:

Simply click a cell and type to set the cell's content. Pressing enter or clicking outside of the cell will set its content to be what you typed. You can see the selected cell's raw content in the formula bar, and it's evaluated content whenever you select out of the cell. To add a reference to another cell, type "`=REF(*insert cell position here*)`". If the cell you are referencing is blank, the value you are referencing will be 0. To perform any arithmetic function, first type "`=`" and then the function you want to perform (for example: "`= 1 * 2 + 2 + 4 / 5^2`"). Exponents are written simply as `base^exponent`. Addition is indicated with a "`+`", subtraction with a "`-`", multiplication with "`*`", and division with "`/`". To perform a range function, similarly start the function with "`=`" and type either "`SUM(*insert cell position(s) here*)`" or "`AVG(*insert cell position(s) here*)`". You can give one cell position for SUM or AVG as well (ex: "`=SUM(B1)`"). You can give a list of cells for either function by separating each cell you want to reference with commas (for example: "`=SUM(A1, B1, C1)`"). You can also give a range of cells, with the start cell and end cell separated by "`:`" (for example: "`=SUM(A1:C3)`"). If any of the cells you are referencing are blank, the value you are referencing for that cell will be 0.

If a user tries to type in any type of reference function (REF, SUM, or AVG) and mistypes a cell position, (ex: "`=REF(1B)`" or "`=REF(12)`" or "`=REF(!B1)`"), an error will be


displayed in the cell “ERROR: Invalid cell position.” and an error pop-up box will appear, which the user can click out of. If the user types in a circular reference (ex: A1 has a reference to B1, B1 has a reference to C1, and C1 has a reference to A1), an error will be displayed in the cell saying “ERROR: circular reference failed to set content.”


If a user tries to perform a SUM or AVG function with a cell reference pointing to a cell that has a string for content (ex: A1 is “= SUM(B1)” and B1 is “= hello”), then that cell’s content will display “ERROR: SUM/AVG cannot perform on non-numbers.” as its content. Also, an error pop-up box will appear displaying that error, which the user can click out of. If the user misspelled or mistyped a function name (for example if the user typed something like “= RED(B1)” instead of “=REF(B1)”), the cell's content will display “ERROR: Invalid function.” Again, the error pop-up box will appear displaying that error, which the user can click out of.


If the user types in just an “=” into a cell, or an “=” with any number of spaces around it, the cell will display “ERROR: No function given after =” as its content. Also, an error pop-up box will appear displaying that error, which the user can click out of. If the user tries to evaluate a function containing both strings and numbers (for example: “= 1 + hello”) then the cell will display “ERROR: Type mismatch” as its content. If the user tries to divide anything by 0, then the cell will display “ERROR: Divide by Zero” as its content. If the user tries to divide 0 by 0, then the cell will display “ERROR: Zero divided By Zero” as its content. Again, in all of these cases, an error pop-up box will appear displaying these errors, which the user can click out of.

You can clear a cell by selecting the cell and then hitting the clear button. This will remove both the raw content and evaluated content of the cell.

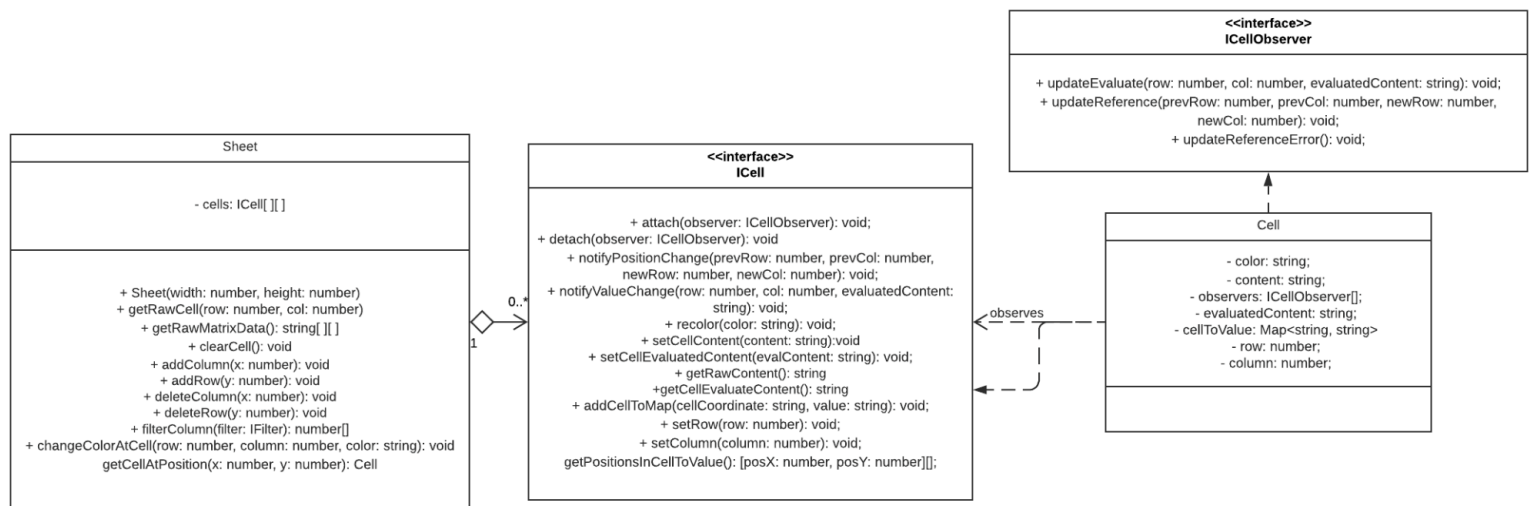
You can add/delete rows and columns with the buttons on the top of the page. A row will be added above whatever cell is currently selected. A column will be added to the left of whatever cell is currently selected. When deleting, the row/column of the selected cell will be deleted. If there is a REF, SUM, or AVG function referencing any deleted cell, this will set the content of any cells with that reference to be “ERROR: invalid reference.”. A pop-up box will also show the error message. The user can click anywhere to get rid of the pop-up box.

For our additional functionality, we added a set color button. To use this feature, the user selects a cell and then selects the color button(). This then shows a drop-down box with a palette of colors that the user can select from. All the user has to do is click on the color they want from the palette and then click outside of the pop-up box for it to go away.

We also added functionality to filter a column. This functionality only is available on cells that are not empty. The user will select a non-empty cell, and then the filter button () will be available to click. Once clicked, the filter pop-up box will appear, with a list of all unique values in the cells under the selected cell (NOTE: Filter only looks at values under the selected cell until it hits an empty cell. The same functionality exists in Excel.) The user can select/deselect values that they want to show/hide from the spreadsheet and then hit apply to view the changed spreadsheet. They can also click outside of the pop-up box to close it. To clear a filter, the user hits the “Clear Filter” button, which will remove the current filter and close the pop-up box. A filter is cleared automatically if the user attempts to filter a different column.

Our last added functionality is export to a CSV. To use this, the user will click the export button (). Depending on what browser the user is running from, it may allow the user to rename the CSV file that they are saving. On most browsers, it will download as “nu-cell-data.csv” automatically. The csv will contain the raw data of each cell in the sheet. We chose to keep the raw data so that references will work if you want to open the csv in Excel.

2. High level architecture diagrams and discussion of the overall infrastructure that you designed to implement the project (20 points).



Our most basic class in the back-end is the Cell class. It implements both the ICell interface and the ICellObserver interface. This is the class where most of the functionality is implemented. Each cell contains variables that are mostly basic information about the cell, such as Color (the color of the cell), Content (the raw content of a cell, which is whatever the user inputs into the cell), row position, and column position. It contains another variable for evaluated content, which is what the raw content of the cell evaluates to. It also contains a variable called Observers, which is a list of ICellObservers that observe the cell, and a variable called cellToValue, which is a Map containing a key of cell position and then its correlating evaluated content value. The observers use this map to keep track of the evaluated content of all the cells that it watches. The key of the map is the string representation of the position of the referenced cell; the value of the map is the evaluated content of the referenced cell. If an observed cell's content is changed, it notifies all its observers of the change and updates the value of the map. If an observed cell's position is changed, it notifies all its observers of the change and updates the key of the map. Further information about the observer pattern is described below in question 3.

The Cell class also contains many functions. First, it contains basic getters and setters. For example, getters and setters for row and column position exist so these values can be updated when a column or row is added or deleted. It also contains the evaluate() method, which is the

main function of Cell. It takes whatever value is input into a cell and either sets the evaluated content to that value if it's a simple input (such as a basic string or number) or performs a function (ex: "= 1+1" or "=SUM(B1,B2)" and sets the evaluated content equal to the value of the result of that function.

The cell also contains notify methods that are used in the observer design pattern, such as `notifyValueChange()`. This function notifies all the observers of the cell that the evaluated value of the cell has changed by calling `updateEvaluate()` on each observer. `updateEvaluate()` updates the content of the observer cell. `notifyPositionChange()` notifies all the observers of the cell that its position changed and it calls the `updateReference()` function on each observer.

`updateReference()` updates a cell's raw content to replace the previous row/column of a referenced cell with the new ones. It also updates the cell's map to have a key of the new coordinate (of the observed cell) with the value that of the old coordinate in the map.

`notifyDeletion()` notifies all the observers of the cell that the cell has been deleted and it calls `updateReferenceError()` on each observer. This function sets the content of the cell to "ERROR: invalid reference" and re-evaluates. It also notifies deletion of a cell to all observers of this cell (recursively).

The Sheet class is responsible for holding all the cells, which are stored as a variable in a matrix array. Its functions include basic getters (no setters). Most getters in this class are to get some data from a cell at a certain position, such as `getRowCell()` (gets the raw content of a cell at a given x,y coordinate). It also contains `getRowMatrixData()`, which gets the entire matrix of cells. It also contains all functions for add/delete row/columns. `addColumn()` is responsible for adding new cells and triggering the notification of position change to all affected cells. `addRow()` is functionally the same as `addColumn`, just with rows instead. The delete row/column methods do the same thing but the opposite, and instead trigger the notification of all deleted cells to all observers. Sheet also contains `FilterColumn()`, which is one of our added features. This function, given a string to filter out, returns the row indices for any cells in the column that should be hidden. The front-end uses this to know which rows should be hidden after the filter is applied.

We also have a Util class in the backend. It contains functionality that was needed all over the place, so we condensed it all into a single class, to prevent duplicated code. It contains `UIPositionToModelPosition()`, which converts from an excel-style position (like B3) to what its coordinate would be (like [2,1] for B3). It also contains `loopThroughRangeOfPositions()`, which

finds all cell positions between two given positions. This is used in formulas for inputs like (A1:B4). It contains `letterToNum()`, which converts a letter representing a column value (such as B) to its corresponding index (in B's case, 2). It also contains `numToLetter()`, which is the same as `letterToNum()` but the other way around. Lastly, it contains `posEqual()`, which checks that two given coordinate points are the same.

In the front-end, we have `App`, which is the main display of the webpage that holds all sub-objects. For `App` we use a combination of `useState` and `useReducer`. We have two modals for this program, which are for filtering and for error messages. They take advantage of `useState` in order to say whether they're showing or not. Every other state variable in `App` is stored in something called a reducer. Functionally, it just acts as one massive `useState`. `useReducers` are used when a gigantic amount of `useStates` would be needed in order to track all the variables of the function as states. In our reducer we store a bunch of informational variables, such as the currently selected coordinate and the instance of the backend's sheet class, which is referenced for every implemented functionality. The actual reducer function is just a giant switch case that allows us to change the variables stored in the state of the reducer. The `applyFilter` function here is run by the filter modal to hide rows in the spreadsheet. The HTML in `App` is basically just instantiating all the other classes. First are the `NavBar` class and formula bar elements, which both have spacers since they're fixed elements. Then, the table is instantiated. Finally, the modals for filtering and for displaying error messages are created.

Moving to the `NavBar` class file, this class is for handling the bar that appears at the top of the screen and all the buttons on it. The boolean variables defined at the top are used by the implemented buttons to make sure that they're disabled when an action on them wouldn't be valid. Each function in this program is very simple, it calls the corresponding function that was made in the backend's `Sheet` class and makes sure that any relevant variables in the app's reducer state are updated. For example, when `addColumn` is run it also updates the `AppState` "numColumns" variable. The HTML for `NavBar` uses several libraries to achieve its current look. First, the "bootstrap" library is used for the main navbar and buttons. The "CSVLink" element is pulled from the "react-csv" library. Finally, the color selector is from the "react-colorful" library.

The table class file is by far the simplest of the front-end. Its only real purpose is to actually display all of the cells in a HTML table. This is done using map loops, which here we just use as actual for loops, to set up our matrix of cells.

Finally, with the VisualCell file, we get our visual implementation of the backend's Cell class which is what's displayed over and over in the table class. VisualCell also has a reducer, as there are a lot of variables that need to be tracked for each cell. With VisualCells, we get into using `useEffects()`, which allows us to run a script every time a certain variable is updated. Since cells can update without the user typing anything in them, we implement a series of `useEffects` to watch for changes. For example, we look at the `appState`'s currently selected cell to determine whether a `visualCell` is currently highlighted by the user and update the cell based on that. We also check if the passed-in cell changed (which indicates an adding or deleting of rows) and make sure we have the most up to date Cell class object for this `visualCell`. The HTML for this class is also very simple; it is just an HTML cell with an HTML input inside of it. The style on both elements is to allow the cell to get longer as the amount of things typed in the cell gets longer.

3. Describe the approach that you used to track dependencies between spreadsheet cells (20 points)

To track dependencies between spreadsheet cells, we used the observer pattern. Cells are both the observers and the subjects. The Cell class implements the ICellObserver interface which contains the three update methods (updateEvaluate, updateReference, and updateReferenceError). Observers are used during any event where a cell that references another cell needs to change its value without being edited directly. If one of the cells referenced in a formula is updated, the cell holding the formula is told to change its value. There are three main cases for observers and three different notify/update methods for each of these: (1) if a cell being observed has a position changed (in the case of add/delete row/column), (2) if a cell being observed is deleted through delete row/column, and (3) if a cell being observed has a change of content.

When a cell's content is set to an expression containing one of the three reference functions(AVG, SUM, REF), we add the cell to the lists of observers of all cells that are referenced in the function. A map of the string representation of the position of the referenced cells to their corresponding evaluated value is kept in the cell. When one of the referenced cells has its value or position changed, it will notify the observer of the change and update the observer's map. When one of the referenced cells is deleted, it will notify the observer and set an error message to all the cells that refer to the deleted cell (directly or indirectly through another reference.) Descriptions of all functions used are described above in question 2.

4. Discussion of how the design and functionality of your system evolved since the phase B plan. Include a discussion of how you came to your final implementation of your spreadsheet. (20 points)

We changed our use cases a bit. One example is that we were planning to have functionality to export as a PDF, as well as a CSV. We just went with CSV since that makes more sense in the context of exporting a table. We also slightly changed how we first described how the filter function would work. Instead of a small drop-down box appearing to the right of the cell that filter was applied to, a larger pop-up box appears whenever you hit the filter button. This seems more intuitive for users. For our color cell function, we were planning on allowing the user to be able to select multiple cells to color, but we did not implement this “select multiple cells” functionality. Instead, we focused our time on implementing our more important features.

Similarly, our use cases described the SUM and AVG functionality to work by the user typing “= SUM(” or “=AVG(” into a cell and then selecting one or multiple cells by clicking and/or dragging. However, we decided not to implement this, and instead have the user just type in what cells they want to select (ex: “B1” for an individual cell , “B1, C1, D1” for a list of cells, or “B1:D2” for a range of cells) inside of the SUM or AVG function, and then closing the function with parenthesis and hitting enter.

The way we described exponent to work in the use cases is only slightly different. We said that a user would perform exponentiation by typing “*number*^(*exponent*)”. However, parenthesis are not needed around the exponent, which is more intuitive for the user. The Error messages that we described in the use cases are also slightly different, with messages that are clearer and more intuitive for the user. Also, a few more error cases have been added that we missed in the use cases (for example: “ERROR: invalid function.” if the user typed something like “= RED(B1)” instead of “=REF(B1)”).

The way we described the concatenation function in the use cases also changed in our implementation. In the use cases, we said that the user must type a concatenation function with the strings wrapped in quotation marks (for example: “= “zip” + “zap””). However, we allow users to concatenate strings even if they are not wrapped in quotation marks since it is more intuitive this way (for example: “= zip + zap”). Both examples given will result in “zipzap”. We also were planning to throw errors in the case that a user types “= “zip” “zap”” or “= “zip” - “zap”, but we decided that we would not throw an error, and instead set the cell to equal whatever is after the equals sign. We are treating the entire content after the equal sign as a

string. There could be a case where the user just wants to set the content equal to a string, and that they were not meaning to apply any sort of concatenation function, so we don't want to throw an error.

For add/delete row/column, in our use cases we planned to have the user click directly on the row/column number/letter (Ex: hit row "2" directly instead of a cell) that they wanted to add to/delete and then hit the corresponding button. However, we allow the user to simply click on a cell and then hit add/delete row/column from there. We found this to be more intuitive for the user. We also were planning to gray out the add/delete row/column buttons if a row/column wasn't selected, however this no longer implies. Instead, the buttons will only be unable to be clicked if no cell is selected.

Overall, our skeleton code stayed mainly the same, except there were a lot of classes we didn't use. Any classes for buttons were not used, and neither was the IFilter interface. We decided to just make functions for filtering inside of the Sheet class that the front-end could then handle. Buttons were also handled by the front-end.

5. Reflection on software engineering processes that your team applied in the project, including design patterns, development process, version control, code review, and testing (20 points)

When we first started, we decided to split our team into a front-end and back-end team. Since all of us have worked on a co-op with an agile process, we decided to try that. We used JIRA to organize the work that needed to be done. This was done by dividing the work into small tasks. These small tasks would be written as tickets and then placed on the JIRA board. The JIRA board consisted of five categories: To-Do, In-Progress, In Review, In Verification, and Done. The JIRA board would visualize what everyone in the team was working on. It would also visualize timelines on when everything needed to get done. We also thought about using the recommended milestone dates as the end date for our sprints. However, using an external software to organize ourselves proved to be more time consuming for a project of this size. This is because the amount of time spent learning and using software that would not add a noticeable benefit to our project efficiency did not seem worth it. If the team was bigger or the project needed to scale more, then we would have used JIRA to make sure what we were building was robust. We instead switched to having frequent updates between the front-end and the back-end. This allowed us to work in parallel without having to do redundant work.

The design pattern we decided to use for the spreadsheet cells was the observer pattern. It made the most sense to use this pattern because there would be a lot of objects that depend on each other. This allowed us to create custom behavior for alerting cells of changes too. Different changes in the cells cause different behaviors to occur so this proves extremely useful.

We used Git for our version control and had two main working branches in addition to our master branch ("Front-end-setup" and "backend"). To ensure that each branch did not get out of date, we did a weekly merge with master. The front-end and back-end mostly pair programmed on their respective branch. However, whenever a major refactor of code was attempted, a new branch was created. For example, August attempted to optimize our spreadsheet late into the project which required a major refactor in the code. Since what we had for the spreadsheet was already working, he created a separate branch to work on in case his changes didn't end up working. Even though we could have just stayed on our original branch and reverted to a previous commit if something went wrong, having a separate branch allowed work on the front-end to be done in parallel. This way, August could work on optimization while

Mitch could work on styling. Without separate branches, doing parallel work would cause merge conflicts and thus waste a lot of time.

When it came time to do our weekly merge for master, each team would review the other team's code. This would allow a fresh pair of eyes to look at the code as opposed to having each team review their own code. Doing these code reviews in the beginning of the project when most of what we had was just skeleton code actually proved super useful. This was due to each team finding unnecessary functions that were going to be implemented. This saved us a lot of time by ensuring that we were only developing the functionality that was needed.

Our philosophy for this project was to employ test driven development. We did this by first writing a bunch of tests that mirrored the use cases we wrote for Phase B. Then we filled in the gaps in our testing that our use cases didn't cover. Once we had our initial list of tests, we used it as a list of high-level requirements that needed to be completed in order to finish the project. However, this was not the only time during the project that we created tests. We used describe statements to separate our tests into different categories. To determine which tests needed to be written, we followed the Software Unit Test Smells guidelines. All of our tests were written using the Jest testing suite. We also had plans to use Selenium as our front-end testing framework. However, front-end testing proved to be much more complex and time consuming than creating back-end tests. Additionally, we could not create the tests until after we created a working front-end which went against our test driven development philosophy. Because of this, we decided to forego writing front-end tests and would just manually test it ourselves. We also had the back-end team test the front-end.