## 0.1 Strategy

In order to develop a working software calculator prototype in the coming weeks, our team has come up with a development strategy to be prepared for challenges that we may face. This strategy must take into account a plan for writing requirements for features implemented, the technologies selected to develop the calculator, ideas of algorithms for numerical computation of selected functions, and tasks to be allocated to each team member based each of our strengths. With a well advised plan of action, we are much more likely to collaborate efficiently as a team and ultimately meet the deadline of the project.

## 0.2 Choice of technology

Our team needed to select technologies to meet our software needs while also complementing the team members' development expertise. The necessary technologies included: • a main programming language with unit testing libraries, a code documentation interface and a graphical user interface library • communication tools • version control software

For our primary programming language we decided to pick the language in which all of our team members had experience writing code in, Java. This allowed each team member to jump right into coding when the time came since no one was blocked having to learn a new language. Java is also advantageous since it has many libraries that we would be able to use for unit testing (Junit), GUI development (JavaFX) and code documentation (Javadoc). Additionally, its object-oriented design fit with how we envisioned designing our calculator program.

In order to coordinate with each other outside of regular meeting hours, we decided to use Discord as our communication tool. Discord can be accessed on almost any device, is very reliable and is easy to use. It also allowed us to create different chat channels for different subjects in order to keep our discussions on topic. For example, one channel could be for scheduling and one could be for brainstorming ideas about algorithms.

We decided to choose git as our version control software since it is simple enough to use, it has tons of documentation to support our needs and some of our team had already used it. Furthermore, they would allow us to work on the same files at different times and easily keep track of changes made to any of the files.

## 0.3 Algorithms evaluation

Based on our experience and with the information gleaned from the interview, we decided to implement at least the following transcendental functions: $x^y$, $e^x$, $sin(x)$ and $cos(x)$, $tan(x)$, $ln(x)$ and $sqrt(x)$. Currently, optional functions are $arcsin(x)$, $arccos(x)$, $arctan(x)$. In order to develop solutions for our calculator without help from Java's math library, we would have to do research on numerical methods to compute these functions. We would have to find the sites with mathematical references such as Wolfram-Alpha, Wikipedia and YouTube in order to find solutions that would be feasible for us.

A few of these functions ($sin(x)$, $cos(x)$, $tan(x)$, $e^x$, and the $arc*(x)$ functions could be approximated relatively easily by Taylor polynomials. Series lend themselves naturally to iterative methods (ie. simple 'for' loop) so that is what these algorithms are.

For example, here is the algorithm for $e^x$:

```
$result \leftarrow 0$
$precision \leftarrow p$ (arbitraty precision integer)
$iterator \leftarrow 0$

while $iterator < precision$ do
  $result \leftarrow result + \frac{x^i}{i!}$
  $i \leftarrow i + 1$

return $result$
```

This algorithm does not pass test for a wide range of values of $e^x$. Currently, we are debugging this. This is an extremely simple algorithm compared to the other but they mostly take this form. The complexity of this particular algorithm is constant, however, it calls the $x^i$ function which we have implemented hamfistedly as $x$ multiplied by itself $i$ times. We intend to optimize the power function in later iteration to take advantage of the many optimizations possible (ie. bitshifting, etc.).

The algorithm for $ln(x)$ is:

```
if $result \leq 0$
  $return$ error

$precision \leftarrow p$ (arbitraty precision integer)
$iterator \leftarrow 0$

while $iterator < precision$ do
  $result \leftarrow result + \frac{1}{(2i+1)} * \frac{(x-1)}{(x+1)}^{(2i+1)}$
  $i \leftarrow i + 1$

return $(2 * result)$
```

The $ln(x)$ algorithm is of a similar vein as $e^x$ and of a similar complexity. Tests have been successful up to about x = 870. We are still debugging this.

## 0.4 Requirements gathering

In order to create software requirements, our team got together to brainstorm ideas for what features the calculator would have and how to implement them.

We made sure our features were realistic, keeping in mind the time constraint of the project and the development experience of the team. Once we had a base for how we thought our calculator could be developed, we decided to conduct interviews on potential users to find out what features everyday users of calculators actually valued and if our initial ideas complemented these. From the answers from our interviewees, we created user personas that had concrete problems and tasks that needed to be completed and that our software would solve. The user personas along with our initial discussions would then inform what different use cases might be for our calculator. The use cases were developed in diagrams to

Tasks Insert Dan's part on tasks?

## 0.5 Testing strategy

For testing purposes, we decided to use Junit to write all of our unit tests. These tests would verify the functionality of our different mathematical functions. We would need to think of and write up as many test cases as possible for different scenarios (negative numbers, large and small numbers, invalid input, etc) in order to cover all potential issues.

Test coverage is incredibly important in the early stages of a software project. Early and thorough testing reduces the risk of having an insurmountable amount of bugs later on. Taking this into consideration, we decided to employ a Test Driven Development strategy in which we would write unit tests before the functions were complete and imposed a rule that all valid tests must pass before any team member makes a push to our master branch. This would ensure that whenever anyone pulled from the master branch, they would not have to waste time fixing someone's bugs to work on their own task.