Writing Graphing Calculator Software for the Nintendo DS

Cal Tipton

Computer Science and Mathematics

Wittenberg University

This thesis was submitted for a Bachelor of Arts in:

Computer Science

Mathematics

Dedication

This one is for the fraternity brothers who were always excited (or at least pretended to be) when I implemented a new feature. You guys kept me going throughout this semester's trials, and I'm not entirely sure I'd have made it without you. Thank you for being part of the ride, even if you didn't sign up for it.

Acknowledgments

From the bottom of my heart, thank you to the computer science and math faculty who helped me through this process. Your patience, guidance, and assistance made this project so much better than you will ever know.

Abstract

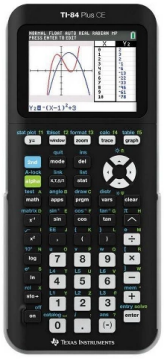
I created a program for the DS that acts as a graphing calculator in an attempt to replace a Texas Instruments TI-84 Plus CE. I did not quite succeed, but I did make a lot of progress toward that goal and set the foundation for something much greater than even the TI-84. In doing so, I greatly improved my skills as both a computer scientist and mathematician, and I am excited to learn even more. This independent study allowed me to improve at my own rate and do things that I was interested in, and it has absolutely contributed to my university experience.

Table of Contents

Introduction

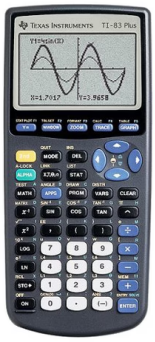
The end goal of this project is to have a practical, functional graphing calculator software that runs well on the Nintendo DS. This matters because alternative calculators (Texas Instruments, Casio) can be prohibitively expensive, especially for the students who actually need them. This graphing calculator software *could* eliminate all cost for students who already have a Nintendo DS, which for a lot of people in college or high school now (the usual demographic for a graphing calculator), is relatively likely. Even for those who do not have a DS, the hardware itself and the materials needed to run the code can generally be acquired for significantly less than the alternatives. Therefore, this software could help provide struggling students with a cost-effective way to get a tool necessary for their success.

During the development of this project, I consistently used Texas Instruments's TI-84 Plus CE as a benchmark. It is a calculator I saw daily in high school and college, and it is an improved version of the TI-83 and TI-84—two calculators I also commonly encountered. I used it as a benchmark partially because it is so common, but also because it is seen as a standard, and it is used for an enormous range of ages and subjects. For those reasons, it makes a lot of sense to compare this project to a TI-84 Plus CE. The goal was to at least be able to replace the TI-84 Plus CE, but I wanted to surpass it. So, keep the TI-84 Plus CE in mind while reading the rest of this.



TI-84 Plus CE

TI-84 Plus



TI-83 Plus

At this point, it is important to remember that this project is designed to replace a graphing calculator, and nothing less. While it certainly has all of the features a scientific calculator has, it was not designed for that application. Practically, this results in a more complicated user interface than a scientific calculator would probably have and irrelevant features that could trip-up a student. So, even though this could be used for applications and age ranges where a scientific calculator is sufficient, it is not necessarily recommended.

Personal Anecdote

I recognize that it seems far-fetched to say that someone could get a DS for less than a dedicated graphing calculator, so let me put that concern to rest. I bought a Nintendo DSi (a bit of a newer model) the week before I started writing this paper to test my theory. I purchased from a Japanese seller on eBay, and I paid a little under $45 after shipping. I did not have to search eBay for very long, and the DSi is in very good condition (any damage, which there is not much of, is purely cosmetic). Granted, this is nothing more than anecdotal evidence, but I think it still puts points in favor of this project. As a reference point, a new TI-84 Plus CE retails for around $110. A student could, of course, find a Texas Instruments or Casio graphing calculator in a similar manner to how I found this DSi, but I imagine this student would still end up spending more than I did. The point is, mileage may vary.

What is a DS?

As a basic overview, the Nintendo DS is a handheld video game console made by Nintendo. Its most notable features are its dual screens (a distinguishing feature even today), touch screen (years before the iPhone), and the fact that it closes like a clam's shell. Additionally, it reads video games through a cartridge slot in the back and has a microphone. Due to several revisions over its lifetime, there are five distinct versions[1] of the DS! Even then, each one has at least all of these features if not more.

The original DS, lovingly referred to as the "DS Phat"

A Nintendo DS game cartridge

1 Original DS, DS Lite, DSi, 3DS, new 3DS

Hardware Overview

The Nintendo DS may be the second best-selling[2] game console of all time, but it did come out in 2004, which was quite a while ago. So, as a refresher, it has

- one 32-bit ARM9 central processing unit (CPU) with a clock speed of 67 megahertz[3]
- another 32-bit ARM7 processor with a clock speed of 33 megahertz[3]
- 4 megabytes of random access memory (RAM)[3]
- 656 kilobytes of video random access memory (VRAM)[3]
- a basic LCD with a resolution of 256x192 pixels on top[4]
- a basic LCD with touch capabilities and a resolution of 256x192 pixels on bottom[4]
- a microphone[4]
- twelve unique buttons (not counting the touch screen, power, or volume buttons)[4]

This hardware is hardly impressive by today's standards, but it absolutely gives the TI-84 Plus CE a run for its money in terms of sheer computing power. Comparatively, the TI has

- one 8-bit Zilog eZ80 CPU with a clock speed of 48 megahertz[5]
- a basic LCD with a resolution of 320x240 pixels[6]
- 149 kilobytes of RAM[5]
- many unique buttons

Clearly, the DS has an advantage in terms of computing power. Its primary CPU is faster and more advanced than the TI-84 Plus CE's CPU, *and* it has another processor on top of that. Additionally, the DS has an enormous amount of RAM in comparison to the TI. These advantages are apparent when using this program (especially when graphing), and they contribute to a better user experience.

Note on Different Nintendo DS Models

There have been four major[7] revisions since the original DS came out in 2004, and each one has a different name. Every model of DS released after the first one (Nintendo DS Lite, Nintendo DSi,

2 Nintendo reports that 154 million units were sold (https://www.nintendo.co.jp/ir/en/finance/hard_soft/index.html), which comes in just under the PlayStation 2, which is reported to have sold 160 million units
3 https://en.wikipedia.org/wiki/Nintendo_DS#Hardware
4 https://www.nintendo.com/en-gb/Support/Nintendo-DS/Product-Information/Technical-data/Product-Information-619794.html
5 https://dev.cemetech.net/tools/ti84pce
6 https://education.ti.com/en/products/calculators/graphing-calculators/ti-84-plus-ce-python/specifications
7 Significant hardware change

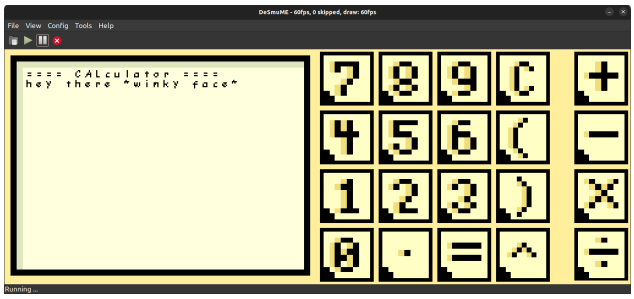
Nintendo 3DS, New Nintendo 3DS) has the required hardware to run software designed for the original DS. This means that as long as software can run on the original DS, it can run on any DS ever released! This greatly increases the number of devices that can run this program, which increases accessibility.

Note on the Development of this Project

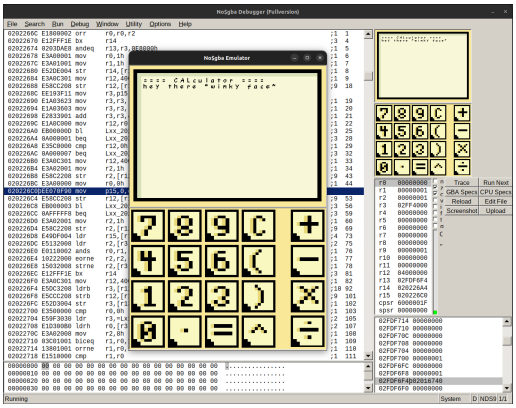
Developing for the Nintendo DS is notably different from developing for a modern PC platform, and generally more intensive. Luckily, the DS uses well-documented processors and a massive number of tools (created by the community) that simplify the process of development. The compiler used for this project (and most other amateur DS projects) is named devkitARM[8], and it is developed by devkitPro[9], an organization dedicated to making compilers for game consoles. devkitPro also develops the primary library used for this project (and most other amateur DS projects) named libnds[10]. It provides functions, tools, and utilities that make the DS's hardware *significantly* easier to work with. For the most part, I was able to ignore the underlying hardware because libnds handles it so well.

The compiler (devkitARM) supports both C and C++, but I have much more experience with C, so that is what this project is written in. The compiler comes with a version of the C standard library (including math functions) implemented by devkitPro for devkitARM, and this project makes good use of it. The compiler also provides a standard makefile to compile DS programs, and this project uses a slightly-modified version of that default one. I personally compile this project on a laptop running Ubuntu 22.04, but it should compile on any platform with the relevant libraries. Finally, I used the DS emulators DeSmuME[11] and No$GBA[12] to test the code before testing it on an actual DS. DeSmuME is easier and more convenient to use, but No$GBA has plethora of developer tools and is more hardware-accurate. I defaulted to using DeSmuME, but when facing a particularly nasty problem, No$GBA's tools helped me get unstuck. Using both in tandem ended up being a healthy balance of convenience and practicality.

8 https://devkitpro.org/wiki/devkitARM
9 https://github.com/devkitpro
10 https://libnds.devkitpro.org/
11 https://desmume.org/
12 https://www.nogba.com/



Screenshot of this project running in DeSmuME



Screenshot of this project running in No$GBA

No Floating Point Unit?

The final note on the development of this project points out that the Nintendo DS has no floating point unit. This means that any non-whole numbers (which there are a lot of) need to be emulated in software. Essentially, using non-whole numbers (a fraction, a number with a decimal point, etc.) can lead to a loss in precision and result in a hit to performance, both of which are problematic for a calculator. This limitation led to a few interesting decisions and implementations that are covered later on.

Design Philosophy

There are, admittedly, a few uncommon design decisions with this project. The most prevalent one is the idea that simplification and exact representation of numbers is paramount. The primary argument for this decision is the aforementioned lack-of-floating-point-unit (FPU). Not having an FPU leads to a whole slew of problems and workarounds that are headache-inducing and still result in precision loss. For this reason, numbers are *not* represented by floats or doubles (numbers with decimal points) internally. Instead, every number is either a fraction (for rationals) or an expression (for irrationals), and this preserves as much precision as possible. This decision, of course, is the reason that the expression manager (a component of this system) needs to exist and have a fair number of features.

Another uncommon, intentional decision was not dynamically allocating memory. I did this to avoid memory leaks, reduce complexity, and mitigate memory fragmentation. Since there is a lot of unused memory in this project, it is not a very big deal to allocate all memory statically. That being said, the total amount memory on the console still is not a lot, so reducing memory fragmentation is a nice benefit.

Basic Usage of the Program

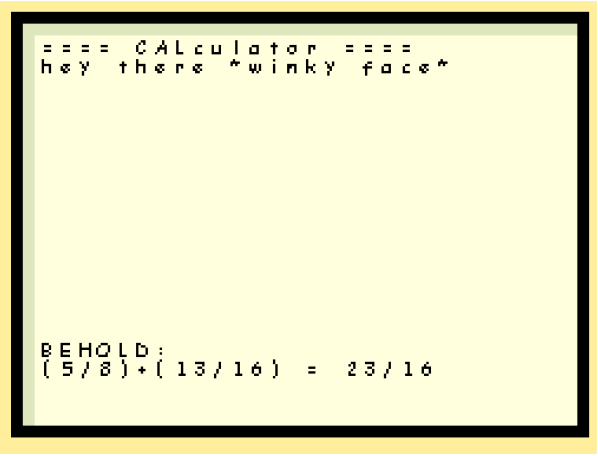
This is a straightforward overview of the usage of this program.

The Primary State

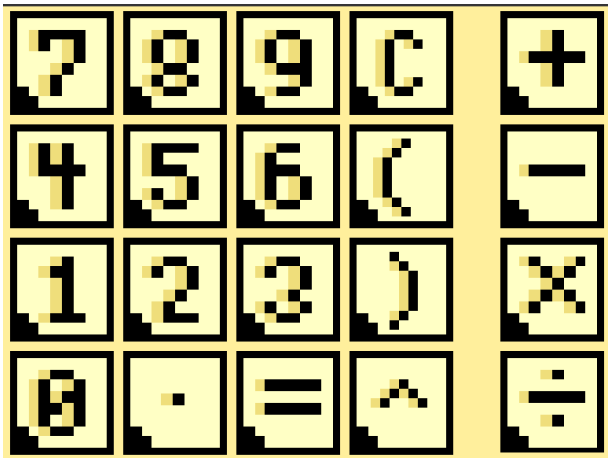
When first booting into the program, the user is greeted with a basic console on the top screen and a keypad on the bottom. The console displays messages, user-typed expressions, and answers to user-typed expressions. The bottom screen is a keypad with numbers 0-9, and addition, subtraction, multiplication, division, and exponent operators. It also has an equal sign, which indicates to the program that the current expression should be evaluated. When an expression is evaluated, it is rewritten with an equal sign at the end, and the calculated result is displayed next. As one could expect, pressing the buttons on the keypad will allow users to interact with the calculator. This primary, “default” state is designed to handle the most common use cases of a calculator, and it is designed to be simple and straightforward to use.

The program’s controls are as follows in the primary state:

- switch keypad: up or down on the DS’s directional pad
- delete the last typed character: B on the DS
- enter the graph state: R on the DS
- typing characters into the calculator: tap on the desired button on the DS’s touch screen



The console after evaluating (5/8)+(13/16)



The default keypad

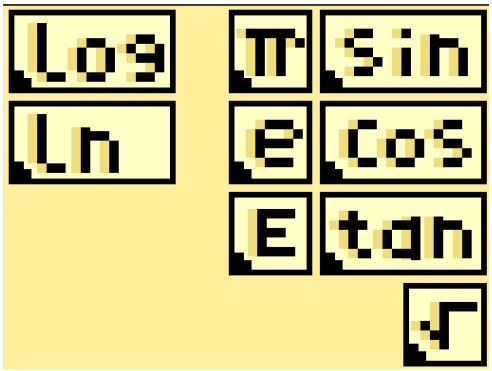
Note on Keypads

In this program, there are multiple different keypads that could be displayed on the bottom screen. There are multiple keypads partially because fitting every button onto a single DS screen would result in a very, very cramped user interface, and partially because it makes the user experience more pleasant. Most would likely remember that their first time using a graphing calculator was daunting and confusing, and that is partially because of the vast number of buttons. By compartmentalizing buttons into different keypads, the user only needs to focus on one set of buttons at a time, rather than being distracted and confused by an entire sea of them.

In this program, there are three keypads: the irrational keypad, the default keypad (the one detailed above), and the variable keypad, and they can be switched between at any point (while in the default state). Imagine these keypads are stacked from top-to-bottom in the order listed, because that is the order that the screens are traversed in. For example, a user could go from the default keypad to either other one, since the default keypad is in the middle. But, a user could not go from the irrational keypad down to the variable keypad without first passing through the default keypad. To go "up" a keypad, the user just needs to press up on the directional-pad of the DS—going down is a similar process.

The Irrational Keypad

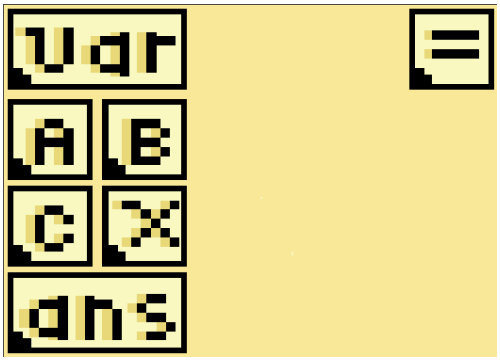
Sometimes, more than what the default keypad can provide is needed. The astute readers will notice that the default keypad provides no way to enter any trigonometric functions, roots, or known constants (π, e, etc.). To remedy this, simply switch to the "irrational" keypad. The irrational keypad contains operators for sine, cosine, tangent, square root, logarithm base 10, natural log, and scientific notation. Additionally, it provides access to buttons for Euler's Number and pi. The irrational keypad is considered to be "above" the default one, so when switching keypads, the user should press up on the directional-pad. I called it the "irrational keypad" because the mathematical functions it contains tend to result in irrational numbers.



The irrational keypad

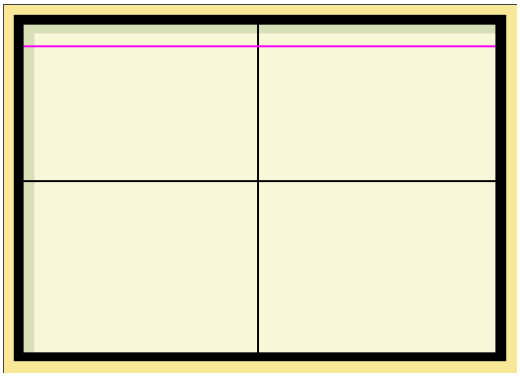
The Variable Keypad

This keypad contains five variables: A, B, C, X, and ans. It also contains an equal sign, but this one assigns a value to a variable and *does not evaluate expressions*.

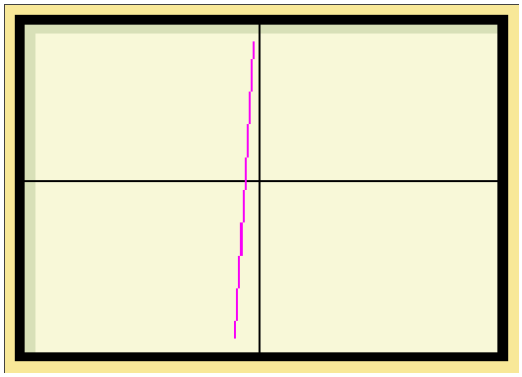


The variable keypad

Variables A, B, and C can be used to store any number (or expression) the user wants, and they can be used directly in expressions. The X variable is a little different, however. In the default state, it always evaluates to 17 (which is just some number I picked). When graphing, however, X is treated as the input for the function being graphed. If the equation $y = 13A + 7$ were graphed, it would look like a horizontal line, since A's value is constant (after the user sets it). It would be the equivalent of graphing a single number. Conversely, if the equation $y = 13X + 7$ were graphed, it would look like a diagonal line with a slope of 13 and an x-intercept at $-7/13$. Another special variable is ans, which always stores the value of the last evaluated expression. If the last expression evaluated was $19 + 17$, ans would evaluate to 36.



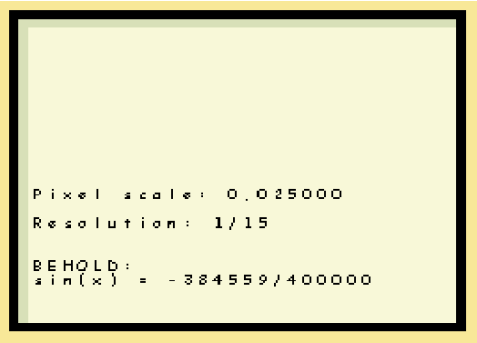
Graph of 13A+7 when A = 0



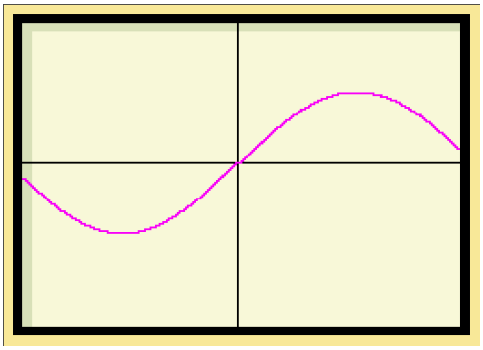
Graph of 13X+7 when A = 0

The Graph State

The graph state is a slightly modified version of the default state. In the graph state, the graph of a function is displayed on the bottom screen instead of a keypad. Additionally, the top screen displays the current function, pixel scale, and resolution of the graph.



The top screen in the graph state



The bottom screen in the graph state, currently displaying a sine wave

The controls for the graph state are:

- increase graph resolution: R on the DS
- decrease graph resolution: L on the DS
- increase pixel scale (zoom out): Up on the DS's directional-pad
- decrease pixel scale (zoom in): down on the DS's directional-pad
- Shift graph: touch and drag on the touch screen
- revert to primary state: B on the DS

The graph displayed is the graph of the last evaluated expression. If the expression does not contain the variable x, it will be a horizontal line.

Components

In a program this complex, it makes a lot of sense to break one large problem into smaller, almost-independent ones. Having many small, simply defined components makes it easier to conceptualize, develop, and debug programs. Think of this system as a:

- User Interface
  - is the part the user interacts with

    - manages the graphical user interface and builds strings for the input processor
    - is absolutely critical to the user experience, and needs to be handled carefully for that reason
    - generates strings that are passed to the input processor

- Input Processor
    - is responsible for breaking a string into smaller parts that the rest of the program can work with more easily
    - contains a lexer and a parser
    - passes the parts of the given string straight to the expression manager

- Expression Manager
    - is responsible for representing mathematical expressions and directly modifying them
    - handles things like simplifying expressions, substituting variables, and representing irrational numbers
    - builds mathematical expressions from the input processor's outputs
    - prepares expressions used by the calculator component

- Calculator
    - handles the raw calculations
    - is probably the most straightforward component
    - made up *mostly* of the calmath library

User Interface

The user interface (UI) is the part of the program that end users will become most familiar with. It includes

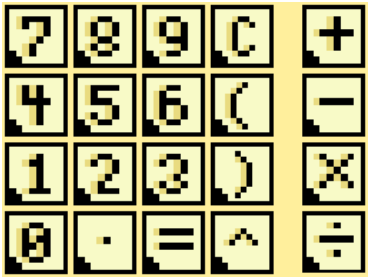
- anything seen on either screen (keypad, numbers, graphs, etc.)
- button presses (includes touchscreen)
- the program's controls

At first glace, it looks like the UI makes up most of the program! Thankfully, it is not nearly as much as it seems. The UI *is* responsible for a wide range of tasks, but none of them are particularly complex.

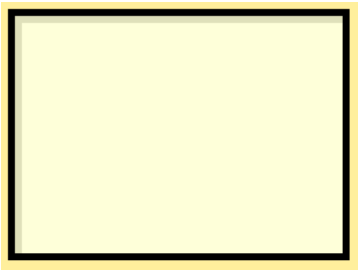
The Graphical Side of this Project

When programming, every DS screen has layers. The layers are ordered, and there is a top layer and a bottom layer. The final picture displayed on the screen is a combination of all of these layers, which means that an individual layer could be blocked by a layer on top of it. Thankfully, this is about as conceptually complex as this section needs to get, because I used a library named NightFox's Lib (NFlib)[13]. This library is great at simplifying the process of displaying and moving two-dimensional graphics around the DS screen, and it saved me a great amount of time. Because of NFlib, I was able to once again ignore a majority of the underlying hardware when graphing.

In the context of this project, the only two things a layer could have on it are *backgrounds* or *sprites*. A background is a still image that takes up an entire layer, and these are things like the image of the console on the top screen, or the keypad that appears on the bottom screen. Sprites are also images, but they are usually much smaller than a background, and since they do not take up an entire layer, they can be moved around the screen. Backgrounds are generally easier to render than sprites, but they cannot move and there can only be one background per layer.



The default keypad, which is just an image (background)



The console with no text, which is also just a background

Primary State

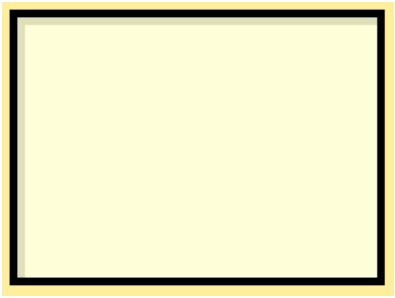
The graphics in the primary state are trivial. Everything that can be seen (except for text) is just a static image, which NFlib is great at. The top screen utilizes two layers: the background layer and the text layer. Again, the background layer is just a still image, and it is the lowest layer on the top screen. The next-highest layer on the top screen is the text layer, and it is the only part (in the primary state) that is not a still image. Instead, NFlib uses an array of sprites to display text, where each sprite is a character/number. It handles most of the logistics there, requiring the user to just pick a font, words, and a place to put them. Again, this is the only layer in the primary state that uses sprites.

13 https://github.com/knightfox75/nds_nflib

The bottom screen is even simpler, with only a single layer that displays the keypad. When the keypad needs to be switched, the current keypad's background is cleared from the layer and replaced with the background of the new keypad.

Graph State

Graphically, the graph state is a little more complex than the primary state. The top screen is identical (two layers, one background one text), but the bottom is notably different. It still has a layer for a background, but it additionally has a second layer for the sprites that make up the actual graph. The coordinate axes and points of the graph are all part of the second layer.

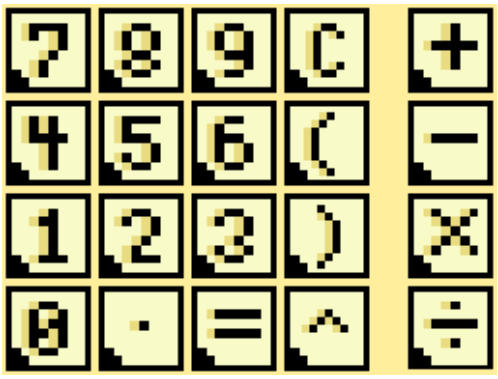


The background of the bottom screen in graph mode. Incidentally, it looks identical to the background of the top screen

Controls and User Experience

The user interface should be simple and easy to use and understand. Graphing calculators are complex machines, and managing all of the decisions and buttons a user can interact with is of the utmost importance. Nobody wants to use a program with an unintuitive UI, especially while trying to take a math test. For this reason, a lot of care and thought went into designing this UI. For example, most people who have already used a graphing calculator will be familiar with the keypad layout of the TI-84 Plus CE, and users who have not previously used one have no preference anyway. For this reason, I decided to layout the number buttons (digits) on the default keypad in the same way they are on the TI-84 Plus CE.

The keypad on the TI-84 Plus CE



This project's keypad (note the layout similarities)

Additionally, the DS's directional pad was selected as the button that switches the keypad because most people will use the DS's stylus with their right hand. Keeping the keypad controls on the left hand allows a user to switch the keypad and still type without a hitch. Up and down on the directional-pad were specifically selected because the alternative is right/left, and right/left are better suited to moving the cursor around. This is not relevant, since this project does not currently have a cursor when typing expressions. However, that feature is desired, and getting users used to switching with up/down leaves left/right free for that feature in the future.

There was actually a major revision in the UI during the development of this project. When first implemented, the graph in the graph state appeared on the top screen. This made sense to me at the time and looked right, but as I thought about it, I realized that moving the graph around with the touch screen would be much more user-friendly and enjoyable than using the d-pad. I also realized that there was no reason to have the keypad displayed while in the graph state, so replacing it with the graph saves space and allows information to still be displayed on the top screen. This is another example of how much care and thought went into this UI.

Input Processor

The input processor is used to convert a string in infix notation to an array of strings in postfix notation. Infix notation is the “standard” and what someone would imagine when thinking of an expression. In an expression in infix notation, the operators ($+, -, \times$, etc.) are between the two numbers they operate on. For example, assume the expression is $3 + 2 \times 19$. The multiplication operator ($\times$) operates on 2 and 19 because it is between both of them, then the addition operator ($+$) operates on 3 and the result of the multiplication (which is 38). The final result in this case, is, of course, 41. That being said, unless the reader strictly follows regular order of mathematical operations[14], there is some ambiguity. A reader could easily assume that the addition operator acts on the 3 and 2, and the multiplication operator acts on the result of the addition (which is 5) and 19. This

14 Mathematical expressions should be evaluated in PEMDAS order: Parentheses, exponents, multiplication/division (whichever comes first), and addition/subtraction (whichever comes first)

would erroneously result in 95, which is a large difference! While the second interpretation is incorrect, it highlights the point. With infix notation, writing code to explicitly respect order of operations is a requirement. This would, however, add complexity to other parts of the program, which is not ideal. Instead, why not consider other methods of representing expressions?

In postfix notation, the operator comes after the numbers it operates on. For example, the expression from before looks like $3, 2, 19, \times, +$ when in postfix notation. It certainly looks stranger than before, but it follows a straightforward pattern. Postfix notation is easiest to read from operator to operator, starting at the left. So, starting at the first operator, the 2 and 19 are multiplied since they are the first two numbers to the left of the $\times$. Then, the addition happens between the result of the multiplication (38) and 3. This correctly results in 41. Assuming the expression is read left to right, this leaves absolutely no ambiguity when it comes to order of operations, which simplifies the rest of the program. Both expressions mathematically mean the same thing, but expressions represented in postfix notation have the order of operations built-in. In addition to that, evaluating an expression in postfix notation requires only a stack and simple logic. For these reasons, the input processor's final output is in postfix notation. To accomplish this goal, the input processor implements both a lexer and a parser.

It is important to quickly note that an expression in postfix notation *could* be ordered incorrectly if done so on purpose. However, when postfix notation is referred to later in this paper, it means an expression is in postfix notation *and* has correct order of operations. It will be explained later why correct order of operations can be assumed. Realistically, postfix notation is convenient just because it avoids the ambiguity associated with infix notation.

The Lexer

A lexer is a machine that converts a string into tokens. A lexer geared toward the English language would break a whole sentence (string) into a collection of individual words (tokens). Lexing is conceptually simpler than parsing, but no less important! The lexer in this program is, of course, responsible for separating a mathematical expression (normal string) into an array of its base parts (collection of tokens). The expression $13 + 26 \div 42$ would look like `{“13”, “+”, “26”, “/”, “42”}` after lexing. This form significantly reduces the amount of work the parser has to do later.

The lexer implements a few features that *seem* like they would live in the parser. For example, the parser does not know that $(3)(6)$ is equivalent to $(3) \times (6)$. The parser would likely return some kind of error given this string. Handling this in the parser would add unnecessary complexity, so the lexer takes care of it. If the lexer sees a close parenthesis followed by an open one, it adds a multiplication token between them so that the parser does not need to deal with it. Another feature heavily implemented in the lexer is the decimal point. The parser does not really know what a decimal point does. It treats it as an operator and does nothing special with it. For this reason, the lexer needs to

account for the fact that any numbers after a decimal point are assumed to be divided by a power of ten. When the lexer sees a lone decimal point followed by a number, it first adds a zero before the decimal point. Then, it adds an open parenthesis and the number to the output. Next, it adds a divide and a power of ten relating to the number of digits the number has. It finally closes the parentheses to make a valid, clear expression for the parser. For example, the expression $.237 \times 18$ would be converted to `{“0”, “.”, “(“, “237”, “÷”, “1000”, “)”, “×”, “18”}`. As a final example, negative numbers are handled in the lexer as well. The parser does not know (or care) what a negative number is, so the lexer is responsible for modifying the output to make a clear statement that the parser can use. To do so, it adds an open parenthesis, a zero, then the minus, then the number, and a close parenthesis to the output. There are a few other syntax features that are simplest when implemented in the lexer, but these are the standout ones. The other two features are a unary minus and multiplying variables and numbers without using a multiply sign.

The Parser

A parser is a machine that converts a collection of tokens into another form that can be used later by the program. In this case, the parser (like the lexer) returns an array of strings. However, the primary difference between the parser output and the lexer output is the notation they use. Like mentioned above, the lexer outputs an array of strings in infix notation. The parser, however, outputs the same array of strings in postfix notation. To do so, it uses the shunting yard algorithm conceptualized by Edsger Dijkstra in 1961. This is important because the expression manager (the next stage in the pipeline) expects to build expressions from postfix string arrays. Remember the expression $13 + 26 \div 42$ from before? After lexing, it looks like `{“13”, “+”, “26”, “/”, “42”}`, and after parsing it looks like `{“13”, “26”, “42”, “/”, “+”}`, which is the correct postfix representation.[15]

Shunting Yard Algorithm

This is where I explain why correct order of operations can be assumed with postfix expressions in this project. The shunting yard algorithm is a powerful algorithm that converts an expression in infix notation to its representation in postfix notation. But, everyone already knew that. The most notable feature of the shunting yard algorithm is that it can take order of operations into account. This means that given any string in infix notation, it can convert it to postfix notation *and* order it so that it respects order of operations. Assume that the expression from before ($13 + 26 \div 42$) is passed to the shunting yard algorithm. As seen above, it would output $13, 26, 42, \div, +$. However, what if the shunting yard algorithm were passed the expression $(13 + 26) \div 42$? This expression uses parentheses to imply that regular order of operations should not be followed, but can the shunting yard algorithm respect that? Yes, it absolutely can, and this is the reason that the postfix expressions in this project are *always* in the correct order of operations. The exact order in which any expression in infix notation is supposed to be evaluated in is *always preserved by the shunting yard algorithm*. Therefore, if an expression is in postfix notation in this project, it indisputably has the correct order of

15 https://en.wikipedia.org/wiki/Shunting_yard_algorithm

operations. As a final note, the shunting yard algorithm discards parentheses, which is why there are never any parentheses after the parsing phase.

As far as the implementation of the shunting yard algorithm in this project goes, I wrote the algorithm almost word-for-word in C into the parser, and I developed it from there. The version used right now has only slight deviations to better support this project, because by default, the algorithm does not account for variables, constants, or functions with only one operator ($\sin x, \cos x, \sqrt{x}$, etc.). Therefore, the current version is not a "pure" shunting yard algorithm, but it absolutely still has the heart of it and does the same general process. The version seen in the expression manager (used to build abstract syntax trees) is a modification of the one in this project's parser, but designed for the applications inside the expression manager.

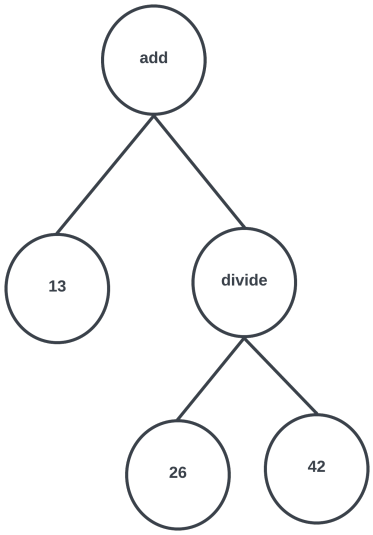
Expression Manager

The expression manager is responsible for representing and manipulating expressions. What does that actually mean? One of the core focuses of this project is maximizing precision by simplifying expressions as much as possible before any calculations are approximated. For example, imagine a user wants to know the value of $\sqrt{2} \times \sqrt{2}$. Most readers, hopefully, will recognize that the exact answer to this expression is 2. A simple way to solve this would be multiplying two approximations[16] of the square root of two, which would result in a number very close to two, but not exactly two. This is fine for a lot of applications, but it goes against the design philosophy of this project. Instead of directly evaluating expressions, imagine the expression is first passed to the expression manager for simplification. The expression manager would recognize the pattern in the expression (square root of a number squared) and rewrite it as just two. Then, that simplified (but equivalent) expression is evaluated. This makes the expression trivial to evaluate *and* (more importantly) it results in an exact answer, which is much better than an approximation. This is how expressions and calculations are handled in this project. The expression manager will first simplify expressions as much as it can, and only then will they be evaluated.

Representing Expressions

That is all well and good, but how does the expression manager work? First, it helps to understand how an expression is actually represented. This project represents expressions using abstract syntax trees (AST), which are a type of binary tree. Each node of the AST represents either a number or operator, and each edge connects operators to their operands. Consider once more the expression $13 + 26 \div 42$. In this form, it is a regular string representing a mathematical expression using infix notation. Conversely, when represented by an AST, this expression would look like:

16 Remember that the square root of a number cannot be exactly calculated in this system, so it must be approximated. This is the case for most calculators/cpu architectures.



Notice that each operator has two nodes attached to it, and that higher precedence operators (the ones that are evaluated first) are lower on the tree. Now, how would someone get a number out of this? Surprisingly, the only thing needed is a postorder traversal. When it comes to binary trees (which an AST is), a traversal refers to the order in which each node is visited. A postorder traversal goes left, right, then evaluates the current node, and it is usually performed recursively. If this tree were traversed in a postorder manner, it would visit nodes in the order `{“13”, “26”, “42”, “divide”, “add”}`. Does that look familiar? A postorder traversal of a correctly-generated abstract syntax tree results in a postfix expression. Obviously, this is great, because it means there is a very simple path toward evaluating expressions, *and* it correctly manages order of operations since it is a postfix expression.

Why use an abstract syntax tree at all? Certainly, it adds conceptual complexity when compared to leaving each expression as a string in infix notation, and as of now, the only benefit is correctly handling order of operations. Is that really worth the complexity? Surprisingly, yes, but there is more to it. Remember that in this project, there is much more to evaluating an expression than just evaluating it—the expression must first be simplified. Once correctly implemented, abstract syntax trees allow much more flexibility when it comes to manipulating expressions—something the expression manager does frequently. This added flexibility allows the expression manager to be less complex overall than if expressions were left as strings. This property of simpler expression manipulations combined with built-in order of operations makes the decision to use ASTs very easy.

General Process of Simplifying Expressions

Simplification of expressions happens in multiple different phases. The first phase does any calculations it can do that would not result in an irrational number. If an expression was initially $\tan(13 + 27) + 5$, after phase one it would be $\tan(40) + 5$. Note that any operations that could result in irrational numbers are not performed. After phase one is the *rewriting phase*. During the rewriting phase, the expression manager searches through the expression for any known patterns that can be

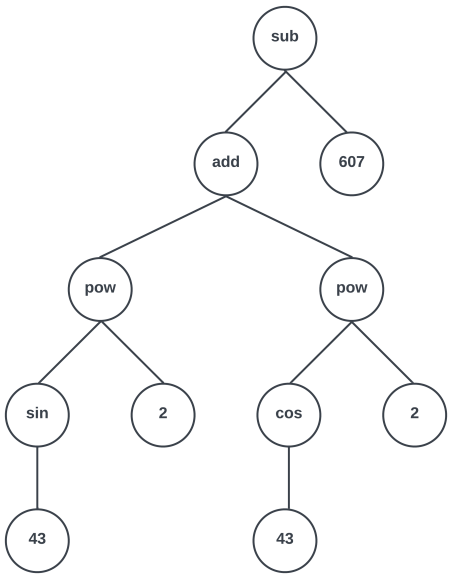
simplified and simplifies them. As a few examples, $sin^2(x) + cos^2(x)$ is replaced with 1, 3×0 is replaced with 0, $\sin(0)$ is replaced with 0, and $\cos(\pi)$ is replaced with -1. Typically, these patterns exist in expressions that evaluate to irrational numbers. But, when rewritten, a lot of these patterns end up being an expression that is rational, which allows more exact results. Again, going from $sin^2(x) + cos^2(x)$ to 1 is a big simplification, and a single rational number is more exact than an approximation would be. Clearly, the rewriting phase allows for a lot of retained precision and exactness. After the rewriting phase is the last phase, which is identical to phase one. Sometimes, rewriting expressions results in a new expression that has more numbers to trivially combine, and the last phase takes care of all of those.

Finally, here is a list of all of the currently implemented patterns

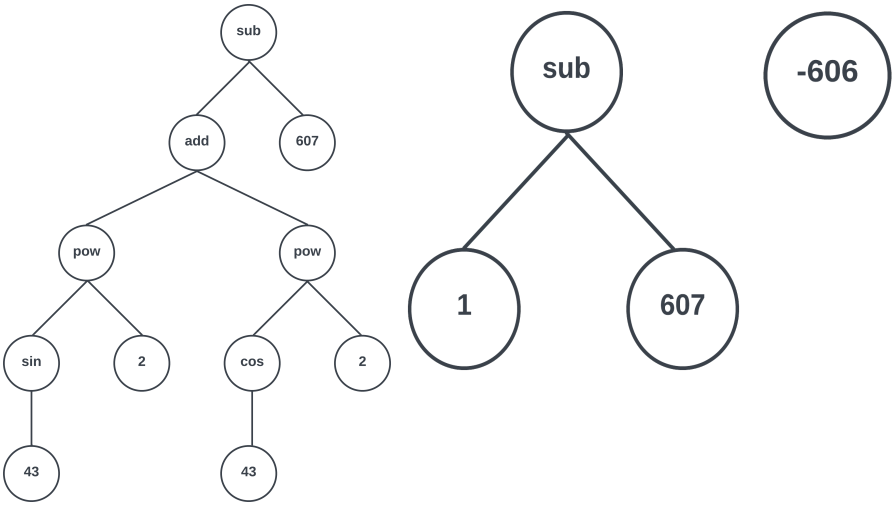
- trig identities for sine, cosine, and tangent in radians (e.g. $sin(0)$ --> 90)
- $\sqrt{(x)} \times \sqrt{(x)}$ --> x
- multiplications and additions by zero (e.g. 16×0 --> 0)
- $sin^2(x) + cos^2(x)$ --> 1

Rewrites with an Abstract Syntax Tree

Imagine the expression $sin^2(43) + cos^2(43) - 607$. The abstract syntax tree for this expression looks like this



and the simplification process looks like this



The first part of simplification is simple combinations. Since there are none, nothing happens and the rewriting phase starts. The expression manager sees $sin^2(x) + cos^2(x)$ and replaces it with 1 (the *x* really doesn't matter). Then, the final round of simple combinations happens and results in -606.

Variables in Expressions

Sometimes, expressions contain variables. Variables are numbers that need to be substituted into an expression, and this is done just before simplification (in almost every case). Typically, rational values are used for variables, but they do not have to be. Since irrational numbers should be able to be used as variables without having to approximate them first, handling variables comes with a little more overhead than would be expected. The only way to exactly represent an irrational number in this program is with an expression. There is no other data type that can represent $sqrt(32)$ exactly (if it were stored in a rational number, the value would need to be approximated). This means that sometimes when variables are substituted into expressions, entire separate ASTs need to be merged into the initial expression, and this is where the additional overhead comes from.

Calculator (calmath library)

The calculator is the least concrete part of the entire project, but it is no less important because of it. I consider this to be parts of the program related to directly manipulating or representing numbers. This includes things like the calmath library, the standard C math library, and functions I wrote to represent and manipulate fractions.

calmath library

The first thing that should be said about the calmath library is that it needs to be renamed. The next is probably that it is a collection of functions and header files I created for the mathematical part of this

project. This includes things like representing fractions, representing operators, representing unsimplified expressions, representing matrices, and finding the greatest common denominator of an integer. This library is absolutely critical to the operation of this program. Here are its notable features.

Representing Fractions

I keep repeating this phrase “representing fractions,” but what does it mean? Representing fractions refers to the process of making a way to replicate the functionality of a fraction in code. It means that somewhere in the code, there is a distinct group containing a numerator, denominator, and sign. I chose to represent fractions using a C struct. They look like this

```
struct fraction
{

      u64 numerator;
      u64 denominator;
      int8 sign;
      int64 sci_notation;
};
```

There is not a lot to a fraction. To precisely represent a rational number, a numerator, denominator, and sign are needed. Since the sign is tracked separately, both numerator and denominator can take advantage of an unsigned 64 bits of memory, which allows a much wider range of representable numbers. Attent readers will notice that there is an additional 64-bit signed integer named `sci_notation`. This is integral to circumventing a problem named overflow.

What is Overflow?

Overflow occurs when a number is outside of the range of numbers a single variable can represent. When a single variable experiences overflow, its intended value can change drastically (and often in a way that is hard to track). For example, an 8-bit signed integer can represent numbers between -128 and 127. If the number 3 were added to this variable, it would probably[17] overflow and become -126, which is quite different from the expected result of 130. This is an atrocity, and it can absolutely destroy precision and accuracy, so it needs to be handled carefully.

Overflow tends to occur when numbers are operated on. Addition and multiplication can cause numbers to be too big, and subtraction and division can make them too small (this case is named “underflow”). Therefore, it makes sense to prevent overflow *before* operating on numbers. In this system, the numerator and denominator are independent of the sign of the number, and they can only be positive, which means numbers in this system can only get too big—not too small. Additionally, this system does not do anything special for subtraction or division. A subtraction is just

17 Overflow is "undefined behavior," which means there is no assurance that the same thing will happen every time. This is what makes overflow so unpredictable.

an addition with a negative number, and a division is just a multiply-by-the-reciprocal. For both of these reasons, this system checks for overflow before each addition and multiplication.

Detecting Overflow

How could someone actually know if an operation will result in overflow? As it turns out, algebra does just the trick! To determine if $a + b$ will overflow, use this expression:

```
a > (largest_representable_number – b)
```

If the above expression is true, then the addition will result in overflow. The order of *a* and *b* do not matter. Similarly, to determine whether or not $a \times b$ will overflow, use this formula:

```
a > largest_representable_number / b
```

If this statement is true, then the multiplication will result in overflow. So, now overflow can be detected, but what can be done about it? Knowing about it does not help much if there is no way to avoid it.

Coping with Overflow

This is the part where it all finally connects back to `sci_notation`. At this point, everyone involved must accept that precision may be lost. If the operands involved are large enough to overflow, it seems fair to lose precision anyway. For fractions, I noted three types of overflow: numerator overflow, denominator overflow, and both.

Since the numerator and denominator are two separate variables, they could each overflow. If the numerator is at risk of overflow, remove some number of digits from the end of it until it is small enough, then add the same number of digits to the `sci_notation` variable, which represents ten to the power of `sci_notation`. For example, if the biggest number that can be represented is 9999 and 3 is added to it, the last digit would be removed, the `sci_notation` variable would be incremented by 1, and 3 would be added to it. The final result would be 9993×10^1. Since 9993 is less than 9999, it can be represented, and not too much precision is lost since the final value is still close when the power of ten is taken into account. That being said, this example illustrates how destructive these approximation methods can be. The true answer to the above example is 100002, but the approximated result is 99930, which is not very close. This difference seems so drastic here because of the small scale of the numbers. Luckily, the biggest number that can be represented in this system is 18,446,744,073,709,551,615. At this scale, the inaccuracy noted in the last example is negligible. That being said, I think there is a better way to do this that induces less inaccuracy, I just have not thought of it yet.

In the case that both the numerator *and* denominator overflow, an equal number of digits is removed from the ends of both the numerator and denominator, and `sci_notation` is unchanged.

Working with Matrices

Currently, only matrices of rational numbers can be represented, so any irrational numbers need to be approximated when used in a matrix. This is because the DS quickly runs out of memory when representing thousands of expressions at a time. Because only rational numbers are supported, the C struct is named `rat_matrix` and defined like this:

```
struct rat_matrix
{
	u8 rows;
	u8 cols;
	struct fraction elements[RAT_MATR_MAX*RAT_MATR_MAX];
};
```

There are two unsigned 8-bit integers that store the number of rows and columns in the matrix. Then, there is an array of size 32×32 to store all of the values. The most numbers a matrix can have in one dimension is 32. The array is one dimensional, so the dimensions of the matrix do not actually matter as long as it has less than 1024 elements.

Matrix Operations

Currently, the only mathematical operation that can be performed with matrices is multiplication. Matrices cannot be added, multiplied by a scalar, or anything similar. The multiplication operation does not use a special algorithm—it is just directly calculated by the algorithm I came up with on the spot. Admittedly, it is not particularly fast or optimized. It does, however, give correct results, so it can be relied upon.

Finally, handling a one-dimensional-array matrix (especially one with a fixed size) needs to be done carefully, since every operation needs to take the set dimensions of the matrix into account. As far as the computer knows, the matrix is just an ordered list of values. It is up to the programmer to assign meaning to the positions of the numbers in the array. To do this, there is a function that converts a row and a column into an array index, and it looks like this

```
u64 rowcol_to_index(u8 row, u8 col)
{
	// RAT_MATR_MAX is the largest a matrix can be in one dimension (which is 32)
	return (row * RAT_MATR_MAX + col);
}
```

It is a simple conversion, but it is important to do it every time the matrix is accessed. Otherwise, the numbers and their positions in the matrix can easily get mixed up, which leads to incorrect results.

Mathematical Functions

Sine, cosine, tangent, logarithm, and natural log all use the standard C library's implementation. That being said, the library's functions expect to get floating-point numbers, but the numbers in this system are all fractions. To remedy this, the fraction is divided to get a floating-point approximation, and that approximation is passed to the functions. That being said, the math functions also return a double, which is not what this system uses. So, the result returned from the library's math functions is converted into a fraction using the calmath library for it.

Converting a Floating-Point number to a Fraction

As a reminder, floating-point numbers are a method of representing rational numbers using decimals. For example, 3.14 is a floating-point number, -2.414 is also a floating-point number. When writing code, floating-point numbers are usually stored in variables with a type of either `float` or `double`. The code used to convert a double into a fraction looks like this:

```
struct fraction double_to_frac(double a)
{
    // 10-17-24
    // Turns a double into a fraction by first converting it to a
    // string, counting the digits after the decimal place, turning the
    // original number into a u64 and using it as the numerator, and using
    // the appropriate power of 10 as the denominator

    // make the number string
    char numstr[20];
    sprintf(numstr, "%0.7f", a);

    // count the number of digits after the decimal place
    u8 numstrlen = strlen(numstr);
    u8 digits_after_decimal = 0;
    bool counting_digits = false;

    for (int x = 0; x < numstrlen; x++)
    {
        if (counting_digits)
        {
            digits_after_decimal += 1;
        }

        if (numstr[x] == '.') { counting_digits = true; }

    }

    // determine the appropriate sign
    int8 sign = -1;
    if (a >= 0) { sign = 1; }
```

```
    // calculate the denominator
    u64 denominator = power_double(10, digits_after_decimal);

    // generate the fraction and simplify it
    return fraction_simplify(fraction_init(ceil(fabs(a) * denominator),
      denominator, sign));
}
```

Once the double is converted back into a fraction, it can be used again by the system, and everything moves on. Here is the code for the logarithm function

```
struct fraction fraction_log(struct fraction a, struct fraction b)
{
    double mag = fraction_mag(a);
    if (mag <= 0) { return fraction_init_error(); }

    return double_to_frac(log10l(mag));
}
```

The implementation of sine, cosine, tangent, and logarithms all look very similar to this one. The function `fraction_mag` divides a fraction's numerator and denominator and multiplies by the sign.

Square Root Algorithm

I was specially requested by an advisor to implement a particular root algorithm for this project, so that is what I did. The requested algorithm comes from the minds of Matt Davis, Adam Parker, and Daniel Vargas, and can be read about in their paper "Being Rational About Algebraic Numbers." Their algorithm was inspired by Theon of Smyrna's approximation of the square root of two, which looks like this:

$$\begin{bmatrix} S_{n+1} \\ T_{n+1} \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} S_n \\ T_n \end{bmatrix}$$

The sequence $(\frac{S_n}{T_n})$ converges to $\sqrt{2}$, and this can be seen after enough iterations. Usually, the starting conditions are $\begin{bmatrix} 1 \\ 1 \end{bmatrix}$, but they do not need to be, and they can actually impact the convergence speed of the sequence. This can be generalized to any rational number. This method is good because it relies on rational numbers—something this calculator works well with. Most alternative square root algorithms rely on floating point numbers, so finding one that natively uses rational numbers is an exciting prospect.

After meeting with Dr. Parker and working with him to implement some code, I noticed that this method works very well with small numbers (2), but as I tried larger and larger numbers (by "larger" I mean 32), the results were less and less accurate. I could, of course, continue iterating in an attempt

to get more accuracy in the approximation, but this comes with two problems. The first problem is overflow/precision loss. Once safeguards for overflow were implemented, this became much less of a problem. However, the second drawback of doing more iterations is speed. Matrix math in this system is, admittedly, quite slow, so iterating over those two matrices hundreds of times is impractical for the user. For those reasons and more, additional iterations would not be the solution to the imprecision.

Noting that the square root of two was approximated very well and relatively quickly, I figured it would make sense to break single numbers apart into a collection of $\sqrt{2}$s and multiply all of those together. With this method, matrix math is mostly avoided and roots only need to be approximated once (for the initial value of $\sqrt{2}$), which circumvents the speed problem. This method worked really well for powers of two because they can be exactly broken up into groups of $\sqrt{2}$. For example,

$$\sqrt{32} = \sqrt{16} \times \sqrt{2} = \sqrt{8} \times \sqrt{2} \times \sqrt{2} = \sqrt{4} \times \sqrt{2} \times \sqrt{2} \times \sqrt{2} = \sqrt{2} \times \sqrt{2} \times \sqrt{2} \times \sqrt{2} \times \sqrt{2},$$

which is easy for the calculator to do. But, what about when the number is *not* a power of two? Surprisingly, this does not add too much complexity to the algorithm. Take a look at this equation

$$\sqrt{36} = \sqrt{18} \times \sqrt{2} = \sqrt{\frac{9}{2}} \times \sqrt{2} \times \sqrt{2}$$

$\sqrt{36}$ can be similarly broken down, but since it is not a power of two, there will always be some kind of radical in the expression that is not $\sqrt{2}$. This, however, is not much of an issue. The root algorithm is already fast enough and precise enough for small numbers, so finding the root of a small number that is not 2 is completely fine. Additionally, note that $\sqrt{2}$ does not need to be calculated for every time it appears in these expressions. For the final form of this algorithm, I hard-coded the value of $\sqrt{2}$ into the program, so only the root of the non-$\sqrt{2}$ number needs to be calculated. With this method, I was able to get relatively precise numbers with a small amount of iterations, which felt like a win. However, its performance compared to the standard C library's root algorithm leaves a lot to be desired, since it is both slower and less accurate than that one. However, there is still a lot left on the table with this method. I could have implemented more strategies from "Being Rational About Algebraic Numbers" to decrease convergence time, and I could improve the overflow-handling strategies to make this method even better. Additionally, speeding up matrix operations would make this algorithm faster as well. Therefore, just because this implementation of the algorithm does not compare well to the standard C library's does not mean that it never will, and I think the work done here could be vastly improved upon to get a method that *is* faster.

Greatest Common Denominator Algorithm

Finding the greatest common denominator (GCD) is really important when doing fraction math. As such, the GCD function gets used a lot, and it is directly based upon Euclid’s GCD algorithm. It looks like this

```
u64 gcd(u64 a, u64 b)
{
```

```
        u64 r;
        L0: r = a % b;
        if (r == 0) goto L1;
        a = b;
        b = r;
        goto L0;

        L1: return b;
}
```

This is about all there is to it. The algorithm was published around 300 BC, and I have not added anything to it.

Final List of Functions

All of the interesting functions of the calmath library have been documented thus far. These are the remaining features of the calmath library

- simplifying fractions
- performing basic arithmetic with fractions (addition, subtraction, multiplication, division)
- evaluating fractions to powers
- evaluating whole numbers to powers
- finding the least common multiple of a whole number
- checking the number of digits in a whole number

An Example with the Entire Pipeline

This is an example of how the expression $13 + \sqrt{12 \times 3.2}$ is evaluated from start to finish.

The User Interface

The string representing this expression is generated by the user using the keypad. The user would simply type the expression $13 + \sqrt{12 \times 3.2}$ into the program, then press the equal sign.

The Input Processor

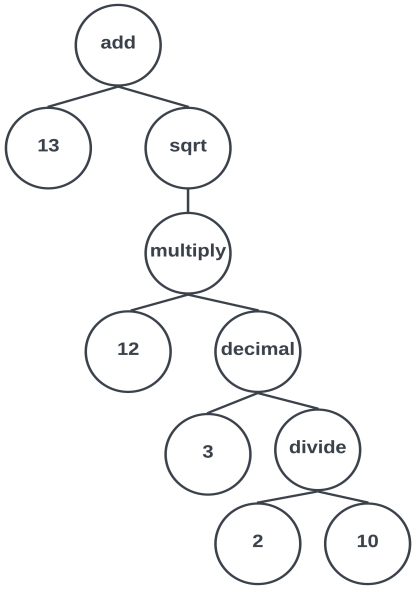
Next, the user interface will pass the string `“13+sqrt(12*3.2)”` to the input processor. The input processor must first lex the string, and this will result in the collection of tokens `{“13”, “+”, “sqrt”, “(“, “12”, “*”, “3”, “.”, “(“, “2”, “/”, “10”, “)”, “)”}`. Remember that

decimal points are handled in the lexer, which is where the additional divide by 10 and set of parentheses come from.

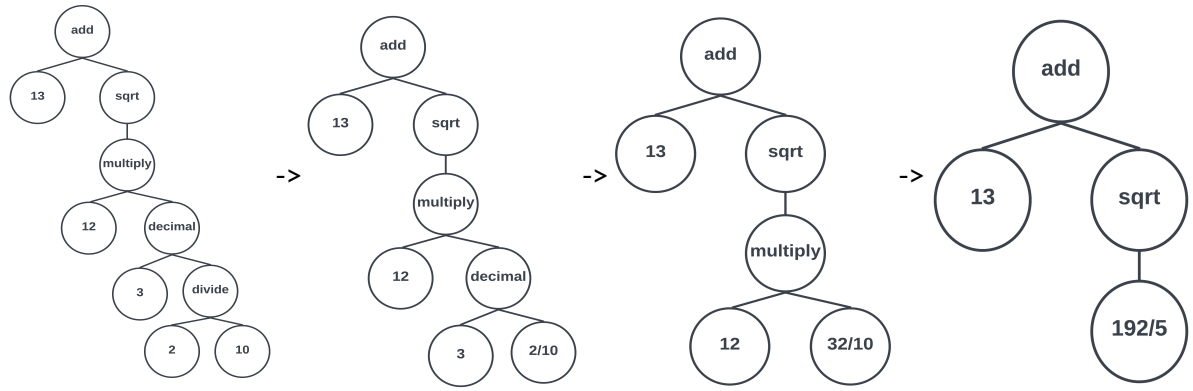
Then, the parser uses the Shunting Yard Algorithm to put the above tokens into postfix order. This would result in `{“13”, “12”, “3”, “2”, “10”, “/”, “.”, “*”, “sqrt”, “+”}`. Remember that the shunting yard algorithm discards parentheses.

The Expression Manager

Now that there is a collection of tokens in postfix notation, it can be passed to the expression manager. The expression manager will first build an abstract syntax tree from the tokens. That AST looks like this



After the AST is built, the expression manager looks for simplifications. The first simplification is the things that are easy to combine. Remember that “easy to combine” means that there are no possible irrational results. Those simplifications look like this



Initial state

2, 10, and divide are combined

3, 2/10, and decimal are combined

12, 32/10, and multiply are combined

Note that this phase of simplification stops at the square root node. This is because the square root of a number could be irrational.

The next phase of simplification looks for known patterns in math. For example, it would simplify the expression $sin(\frac{\pi}{2})$ to 1. There are no simplifications of this nature in this AST though, so this phase makes no changes. The final phase of simplification is another round of simple combinations. Since there has been no change since the last round of simplifications, nothing happens during this phase either. Now that the expression manager is finished with the expression, it is ready to be evaluated.

Calculator

Now that the expression is simplified and in abstract-syntax-tree-form, it can be evaluated. The calculator will once again navigate the tree in a postorder traversal (left, right, current). As operator nodes are found in the tree, they are used to combine the numbers underneath, then that value is replaced in the operator node. Eventually, this process will result in the final evaluation of the expression, and it will be stored in the single node at the top of the tree. The postorder traversal for this example's tree results in `{13, 192/5, sqrt, add}`. Remember that simplification of expressions combines divisions into fractions, which is why 192/5 is its own token.

Graphing

Graphing can be seen as the culmination of all of the features of the calculator. Conceptually, graphing is simple. Anyone with an equation and a domain could probably draw a graph. However, doing it programmatically, accurately, and quickly is a bit more challenging. Graphing is the culmination of all of this project's features because it relies on all of them to work harmoniously together. What does that look like, practically?

Basics of Graphing

Once again, the method for graphing that this project uses is fairly straightforward. Points along the x axis are picked so that they are evenly spaced from each other. Then, those x coordinates are plugged into the given equation. The answers to those equations are used as the y coordinates for the points. This results in a table of x values and their corresponding y values. From there, the points are plotted on the graph and lines are drawn between the points. Sounds simple enough on the surface, but even this simple approach has a lot of steps and pitfalls associated with it.

Calculating the Table's X Coordinates

The first step in drawing a graph is calculating all of the values in the table of x and y coordinates. Before that can happen, the x coordinates themselves need to be picked, and there are two values that affect this: the camera's position, and the graph's "resolution." When graphing, there is an associated camera which just refers to which point the graph view is centered on and how much area is shown around that point. The camera is what affects the zoom and positioning of the displayed graph. In addition to the camera's values, the program *also* needs to know the graph's resolution before it can pick the x values. The resolution of a graph refers to the number of x coordinates picked for graphing. A graph with only five x values has a lower resolution than a graph with seven.

Now that the camera and resolution of the graph are known, points on the x axis can be picked! The formula for the number of points is

```
num values =  greatest number of points / resolution
```

The DS only has 256 pixels across the x axis, and the design of the UI only allows 235 of them to be used to graph. This is what the greatest number of points refers to. Since the DS does not have enough pixels to plot more than 235 points, every graph must have fewer points than that. Note that this number is actually *divided* by the resolution. The resolution of a graph can be thought of as the number of pixels skipped for each one graphed. A resolution of 1 will result in all possible points on the x axis being used, while a resolution of 15 (the highest value it can have) results in merely 15 points. The *value* of each point on the x axis is calculated like this

```
current x value = last x val + step
```

Since this is a recursive[18] function, there needs to be a specified first x value. The value chosen for this is always the first x value furthest to the left. This, of course, depends upon the position and size of the camera. The `step` variable depends directly on the "pixel scale" of the camera, which refers to the number of units each pixel is worth. If a graph has a pixel scale of 1, then each pixel is worth one unit, and each x will be 1 greater than the last x. Changing the pixel scale results in zooming either in or out of the graph.

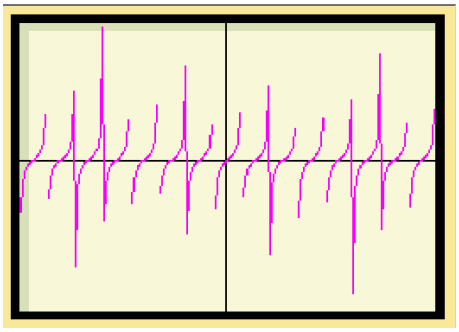
18 Mathematically, not in a computer science way

Calculating the Table's Y Coordinates

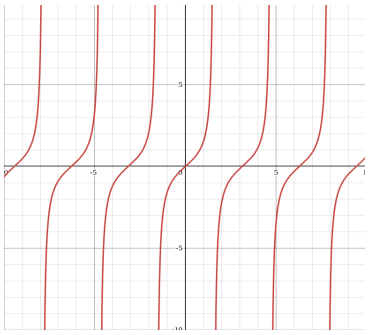
At this point, the table has every x coordinate it intends to use! The next step is plugging those values into the given function, which hinges heavily upon the expression manager. The expression manager is responsible for replacing each x variable in the expression with an x from the table. Again, this is conceptually simple, but making sure it happens correctly and consistently for all functions was the challenging part. But, since the expression manager can be relied upon, simply substitute the x values into the expression and evaluate it using the calmath library. That gives the corresponding y values for each x value, and now that the table is complete, it can be graphed.

Drawing the Points

Given an x and y value, graphing is pretty easy. Determining how to connect those points is less easy. Graphically connecting each point with a line is not necessary, but it adds enough to the user experience to be worth implementing. libnds includes a function that can connect two points with a line, and that is exactly the one that is used to connect points. The start of the line is the coordinates of the last point plotted, and the end is the coordinates of the current point. In a vast majority of cases, this method looks and works great. However, there are certain functions that make using just this approach completely irresponsible.



A graph of tan(x) with irresponsibly-connected points

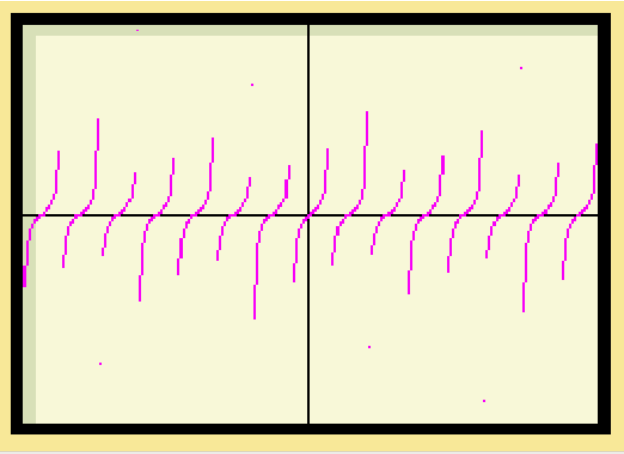


The actual graph of tan(x), according to desmos.com

Connecting Dots Responsibly

As can be seen above, sometimes functions have points that are not actually connected, which means the graph cannot just connect all of the points together without checking first. So, which points should be drawn alone? Admittedly, I do not have a concrete answer to that question. But, I do have a workaround. The distance between the current point and the last one is always calculated using distance formula. Two points are connected if they are less than 30 units away from each other (graph units, not pixels). If they are further than that, they are plotted as single points. I have had relatively

good success with this method, and almost every graph can be manipulated so that it displays correctly (changing zoom, position, resolution, etc.). It is certainly still possible to make some graphs look awful with this method, but one has to actually try to make that happen now.



A graph of tan(x) with responsibly connected points. Note the lone points at the peaks and troughs

Shifting the Graph

A major feature of this system is being able to move the graph's camera in real time. A user can simply tap and drag on the bottom screen to pan the graph, press up or down on the directional pad to zoom, or press the left or right triggers on the DS to change the graph's resolution. This feature is the reason that the camera needs to exist. Remember that the camera's position and zoom is taken into account when calculating the table's values. The camera is complex and tedious to set up if one only ever wants to look at the origin, but it makes shifting the graph around simple and liberating. Changing the view is as easy as updating the camera's values and recalculating the tables values. From a programming perspective, the only thing needed to shift a graph is a single function call.

Graphing as a Culmination

The tables used in graphing are arrays of the fractions defined in calmath. Calculating the numbers used as x coordinates in the table is all fraction math. The pixel scale for the camera of a graph is also a calmath fraction. The "functions" used for graphing are repurposed expressions from the expression manager. The process of substituting variables for values was something that needed to be implemented in the expression manager when variables were added. Points on the graph can only be drawn because this project's UI was designed for it. Graphing directly demonstrates the speed of calmath functions because of the repetitive calculations. When phrased this way, it becomes very apparent that graphing really is only possible when all of the components of this system work

together. The ability to graph is itself a test of this system and its components, and a testament to its functionality. No, it is not perfect, nor am I entirely satisfied with it. However, the ability to graph truly illustrates all of this system's features, and it should not be undervalued.

Conclusion

In the beginning, I set out to create a capable graphing calculator software for the Nintendo DS. While I *did* make an enormous amount of progress this semester (both on this project and as a student), I do not think I achieved everything I aimed to. As it currently stands, this project doubtlessly demonstrates that an ideal DS graphing calculator could exist, and that is exciting! However, this project is not it. The UI, while generally simple and easy to understand, feels a little clunky. The algorithms used for calculations are not as good as I think they should be. Like I mentioned in the beginning, I want this project to be able to surpass a TI-84 Plus CE, but it still does not. Finally, the calculator is not quite “stable” enough for my liking. Its evaluations are generally correct, but it needs to be much better than that. Implementing more in-depth tests and test cases is something I would like to do as well.

Quick Comparison to the Competition

I mentioned that this project still does not surpass the TI-84 Plus CE. Here are a few areas where that can be seen. There are functions (stats, probability, basic integrals, etc.) that a TI calculator can do that this project cannot. While most of these *would* be straightforward to implement, they still have not been. Similarly, the TI-84 Plus CE is quite pleasant to use for matrices. Meanwhile, this project only barely supports them. Also, most TI calculators for the past few decades have supported TI Basic[19], a programming language designed to extend the capabilities of the calculator. Users can write programs directly on their calculators. Back in high school, I used TI Basic to implement a program that finds roots of quadratic equations (using quadratic formula), and that program still saves me time to this day. I would really like to implement something similar for this calculator at some point. I am not sure what that looks like yet, but I have a few ideas, and most of them are feasible at worst. Other than that, TI calculators can (slowly) graph multiple functions at a time instead of just one. Many more features than that are missing from this project, but those are the big ones that I am currently thinking of.

Next Steps

In case it was not obvious, I do not intend to drop this project after graduation. Even in the face of the most relevant and realistic reasons to stop this project, I would likely still continue on out of passion. This project has been very enjoyable for me and has allowed me to make boundless strides not as just a programmer, but as a problem solver and a designer. This is the first time I have designed a system of

19 This is a very good resource for all things TI-Basic: http://tibasicdev.wikidot.com/starter-kit

this scale and complexity, and trying to keep everything organized was has been a shock to me. So, I will carry on.

All that said, here is a list of features I would currently like to see implemented:

- refactor and clean up codebase
  - most of the code is headache inducing and barely documented
- improve stability of overall system by going back over and solidifying the input processor and expression manager (most inaccuracies are related to those)
- better matrix support
  - UI support
  - more matrix operations
  - potentially supporting using functions as matrix entries
- more graphing capabilities
  - multiple functions
  - tick marks
  - ability to "trace" the graph
- a UI that feels more "crisp"
  - will likely require outside people for testing
- more customization and accessibility options
  - customizable colors
  - ability to rearrange the UI
  - customizable controls
- a capable computer algebra system
  - not sure what this looks like yet, but it sounds interesting

Final Remarks

To wrap everything up, I designed and wrote a graphing calculator software for the Nintendo DS. It is not yet ready to replace Texas Instruments's TI-84 Plus CE, but I have demonstrated that it certainly could someday. Again, the features I implemented barely scratch the surface of the amount of RAM and computing power that the DS has. Through proper utilization of hardware and implementation of new features, the DS could be an excellent calculator, even for graduate level students. Someone just needs to write the code to make it all happen.