Expression Parsing & Visualizing Complex Mappings

James Edison[*]

May 18, 2014

Abstract

The purpose of this paper is to explore the theory behind expression parsing which will then be applied for use in function evaluation. This theory driven function evaluator will then be used to calculate results of complex-valued functions, specifically, complex mappings. After calculating the values of a complex mapping, the domain of the mapping can then be colored according to each point's image location in the complex-plane, which will be represented by a 12-part color wheel. The resulting coloring helps visualize the way in which a complex mapping reshapes the entire complex domain. A basic understanding of complex numbers and programming is recommended, but not required.

Introduction

Expression parsing is an important area of study for computer scientists and modern mathematicians alike. Convincing a computer to evaluate a classically written out equation or function can be challenging, since computers don't inherently comprehend language or symbols. Instead, interpretations of those characters are remembered in the memory so that they can be understood in their most basic form. To fully parse an expression, the computer must have an interpretation for every acceptable character, where we accept any character that one might find in an equation or function. Once we parse an expression into a computer, it can be evaluated. For our purposes, we will be concerned only with mathematical expressions.

As humans, we can look at an expression non-linearly and evaluate it based on classical order of operations, essentially parsing and evaluating all in one step. However, since computers have no way of knowing what expression was inputted without looking at each individual character, a program must parse expressions linearly before moving onto evaluation. Thus, to evaluate a parsed expression on the computer, we will need to use methods that are counter-intuitive and would be cumbersome if used when evaluating expressions by hand. One of the most common methods for expression parsing is the *Shunting Yard Algorithm*, or SYA. To

[*]Thanks to Dr. Albert Schueller, whose guidance made this project possible

properly understand the SYA, we must first understand *postfix* and *infix* notation of expressions [6].

As an application of expression parsing, we can use these methods to evaluate complex-valued functions. The results of these complex calculations will then be interpreted as colors, and used to visualize the effects of a particular complex mapping on the domain itself. Frank Farris, in his review of a complex analysis book, describes this method in detail, and provides a few examples [3].

1 Overview of Expression Parsing

Before we can begin parsing an expression, we must understand a few basic theories that we will build our application upon. Namely, we will discuss *postfix notation, tokens, tokenization, stacks, arrays, parsing*, and finally the *SYA*. Tokenization and the SYA will arrange tokens into postfix notation, using stacks to parse a function and arrays to output it the resulting expression.

1.1 Infix and Postfix Expressions

Our first building block will be the idea of Postfix, or Reverse Polish, Notation. To understand what Postfix really means, we have to first look at the way we traditionally represent equations. Infix notation is the classical way of representing expressions.

Definition 1.1 (Infix Notation) *Functions are written such that each operator appears in-between its operands. For example: "$2 \cdot 2 - 3$".*

Alternatively, we are interested in *Postfix* notation.

Definition 1.2 (Postfix Notation) *Functions are written such that each operator appears **after** its operands. For example: "$2\ 2\ \cdot\ 3\ -$".*

Postfix notation is a standard way of representing expressions, but since we're accustomed to seeing them written out in infix notation, this style can be hard for humans to interpret. When the expressions become more complex they will be more difficult, as we might have something like "$531 \cdot +2+$", which can't be easily interpreted as individual expressions, just based on pattern recognition. With this particular expression, the infix equivalent would be "$5 + 3 \cdot 1 + 2$", or "$((5 + (3 \cdot 1)) + 2)$", which can be interpreted by humans much more quickly than its postfix equivalent. So why would we ever care about postfix, since it seems to be a less intuitive way of representing expressions? Well, the biggest reason is that computers have no inherent understanding of the order of operations. The only way to properly preserve an infix expression's order of operations would be to place parentheses around every sub-expression so that the computer understood when and what to evaluate, as seen above. Placing the heavy burden of proper parenthesis entirely on the user is poor design, and doesn't make a particularly robust algorithm. Postfix notation requires no parentheses, because the order of operations is written directly into the notation when understood literally. When evaluating a postfix expression, the computer can simply read through the expression and evaluate whenever it reaches an operator. If we were to evaluate our example operator by operator, we would follow the steps outlined in Postfix Array 1.

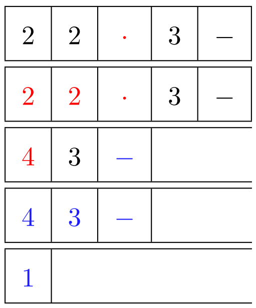


Figure 1: Postfix Evaluation

Postfix is all well and good; but without an algorithm for its creation, we will have nothing for the computer to evaluate, regardless of its notation.

1.2 Tokenization

With evaluation in mind, we must utilize *tokens* so that we can represent a given equation within our application.

Definition 1.3 (Token) *A token is a string of characters that, when analyzed together, have a unique, pre-defined meaning.*

We can now think of an expression as a string of tokens, or as a chain of boxcars all linked to one another, each containing a valuable piece of the given equation. The process of breaking an equation up into its individual tokens is called *Tokenization.* When we tokenize an expression, we are simply checking to see if a given character or group of characters exists in a pre-defined list of tokens. If the token doesn't exist in our list, we can choose to ignore said token, or we can attempt to broaden the scope of the individual token to include surrounding characters before determining whether or not we have a valid token. For example, if we have some expression $f(x) = \sin(x\hat{\ }2)$, we tokenize the right side into 6 tokens: 'sin', '(' , 'x', '^', '2', and ')'. We can see that each individual token plays an important role in the overall definition of our function, and also that if we had simply created a token for every character, we would be left with a token "s," a token "i," and a token "n," which would not help us at all in the evaluation of this expression. Once we have successfully tokenized an expression, we can use an algorithm to shunt each token to its proper place, based on the precedence table found in Table 3. However, before getting to the algorithm itself, we could benefit from a general understanding of parsing.

1.3 Parsing

When representing a function in a computer-friendly manner, we have to preserve the syntactical relationships between tokens. This method of preservation and storage is called *Parsing*. Based on Harvey Abramson's definition found in [1], we define Parsing below.

Definition 1.4 (Parsing) *Parsing is the process by which a string of tokens is interpreted by a computer such that the syntactical relationships between them are preserved.*

We can utilize a tree data-structure when understanding Parsing, creating a binary tree that wholly represents our tokenized input. For instance, given an expression "$2 \cdot 2 - 3$", we would have a parse tree like the one found in Figure 2.

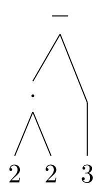


Figure 2: Parse Tree for "$2 \cdot 2 - 3$"

This tree dutifully represents our entire expression, regardless of any notational representations. Each left or right child-node represents the first or second operand respectively, with their parent being some operation. Representing our equation this way, we can create expressions in Infix and Postfix notation easily, based on when we consider the children of a parent node. We can use different tree traversal methods to create different notational representations. If we want infix notation, we can traverse the tree by taking the left child of every node until we reach a leaf. In Figure 2, this would be equivalent to reaching the left-most "2," outputting it, then we would output the other child of the $\cdot$, which is to say the other two, followed by the operator itself. The resulting output is "$2\ 2\ \cdot$", which is the postfix representation of "$2 \cdot 2$." Once we are at a leaf, we output that leaf, followed by its parent, followed by its sibling. We can then treat that parent as our new left-most leaf, and repeat the process. If we want postfix notation instead of infix notation, we can output the second child before the parent. Regardless, we can clearly see the way in which each token relates to the others. Luckily, for our application, we won't have to represent our equations in this way. Instead, we can use two different, more rigid data-structures, and a convenient algorithm.

1.4 Stacks and Arrays

First, a formal definition of stacks.

Definition 1.5 (Stack) *A stack is a linked-list data structure that preserves order, and adheres to the first in-last out paradigm.*

To better illustrate the concept of a stack, imagine a stack of dining chairs. Since each chair is stacked on top of another, removing any but the top chair would be difficult. Suppose we have three chairs, one colored red, another blue, and the last green. If we place the red chair on the ground, we have essentially placed it on top of an empty stack. If we then stack the green and blue chairs on top of the red chair, we get a different top to our stack. This is important, because if we now want to use the blue chair, we can simply remove it from the top and use it. But if we wanted to use the green or red chair, we would first have to remove the blue chair and either use it or set it aside, before being able to reach the green chair below it. To use the red chair, we would repeat the process, popping the top chair off the stack until we get to the chair we care about. Stacks in the context of data structures are equivalent to these stacks of chairs, where we can only access the top, but each chair or bit of information is its own unique portion of the stack. Arrays are a much simpler concept, conforming to the classic mathematical definition of matrices, except that our cells can contain anything we want.

Now that we understand the stack data structure, we can look at how we will utilize it to evaluate an expression. Since a stack can hold any number of tokens, we could theoretically use one single stack that holds our entire expression and use this for evaluation, but we would have difficulty maintaining syntactical relationships. Instead, we can use two unique stacks, one that holds operations and one that holds numbers. Unfortunately, as discussed previously, the computer doesn't inherently understand whether a token is a number or operator, so we will have to interpret each token before we can push it onto the appropriate stack. Fortunately for us, most modern day programming languages come with some amount of built-in interpretation that makes translation from a character or token to a number simple, by automatically parsing entire numbers, leaving behind only the operators. This automatic parsing matters to us, because without it, we would have to read numbers like we read symbols. That is to say, we would parse the number 312 as 3 separate characters, and would have to ensure it was turned into the token 312, and not 3 tokens 3,1, and 2. When tokenizing operators, we must store them as text, because mathematical operations happen behind the scenes, so we can't store an actual operation per se. This becomes more apparent when we include operators that don't have explicit symbols such as trigonometric functions, because one token must include multiple letters. However, since we know the only remaining tokens in our expression are operators, then assuming we have tokenized them correctly (i.e. pulling out the entirety of "sin" and not individual letters), we can push them all onto the operator stack. Now that we have our two stacks established, we can look specifically at how to parse an expression using the *SYA*.

2 Infix-Postfix Conversion Techniques

When parsing an expression for conversion we will have two stacks: an operator stack, and a numerical stack. Since we must evaluate an expression based on order of operations, we require an operator precedence table so the computer will know when to add operators to the stack, and not to. Refer to Table 3 for the precedence table, based on David Guichard's notes on data structures [4]. Using the two stacks and precedence table, we can easily convert an infix expression to a postfix expression.

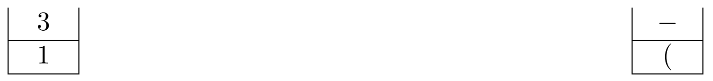
2.1 On-the-fly Evaluation

To convert from infix to postfix notation, we require a precedence table that will tell the program which operators can be pushed on top of the operator stack, depending on the stack's current contents. If an incoming operator has a lower precedence, i.e. false on the Precedence Table in Table 3, then we must evaluate our stacks until we can push our new operator.

		on stack					
		$+$	$-$	$\cdot$	$/$	ˆ	(
	$+$	F	F	F	F	F	T
	$-$	F	F	F	F	F	T
	$\cdot$	T	T	F	F	F	T
incoming	$/$	T	T	F	F	F	T
	ˆ	T	T	F	F	T	T
	(	T	T	T	T	T	T
	)	F	F	F	F	F	T

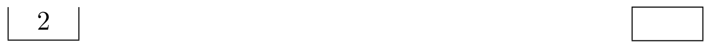
Figure 3: Precedence Table for Binary Operators

If we're given an expression in infix notation, we have to find a way of interpreting this input and converting it into postfix notation, so that we can avoid having order of operations. Given a simple expression, "$(3 - 1) \cdot 2$", we can place each token onto either a number or an operator stack. We begin with a left



(a) Number Stack (b) Operator Stack

Figure 4: Number and Operator Stacks

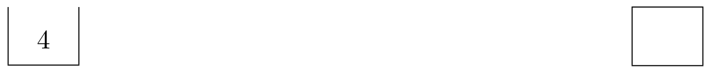


(a) Number Stack (b) Operator Stack

Figure 5: Number and Operator Stacks

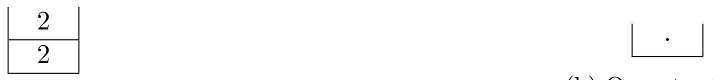
parenthesis, which we know is technically an operator, so we place it on the top of our empty operator stack. Our input is now “3 − 1) · 2”, and we have a 3 at the front of our input, so we know we must push 3 onto our number stack. Following the 3, we have a −, which we push onto the operator stack, because of its precedence over ‘(‘, and then we push 1 onto the number stack. Our stacks now look like those in Figure 4. Now our input is “) · 2”, and we're met with a right parenthesis. We know that in infix notation, a right parenthesis symbols the end of a block of evaluations that are to be completed before anything else. We need to adhere to this fact, so we don't push the right parenthesis onto our stack, and instead evaluate our current stack. We see that our number stack contains “3” and “1″, so we pop both of these off of the top, and pop the top operator, which is −. We know this means to evaluate “3 1 −”, and push the resulting 2 onto our number stack. Our stack now looks like Figure 5, and our remaining input is “·2”. We can now push the · onto our operator stack, and the 2 onto our number stack. Our input is now blank, and our stacks look like Figure 6.

Since our input is blank, we know it's time to finish evaluating the expression based on the remaining tokens in our stacks. Since our operator stack's top element is now a ‘·’, we know to pop the top 2 numbers from our number stack, 2 and 2, and evaluate 2 2 ·. This evaluation results in a 4, which is placed back onto our number stack. Our stacks now look like Figure 7.



(a) Number Stack (b) Operator Stack

Figure 7: Number and Operator Stacks



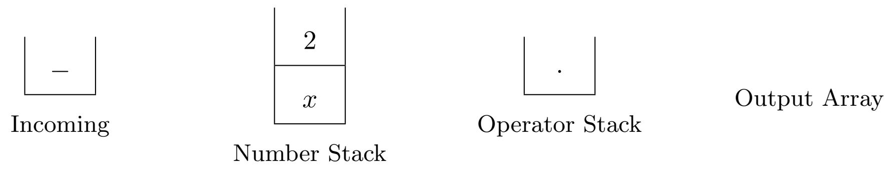
(a) Number Stack (b) Operator Stack

Figure 6: Number and Operator Stacks

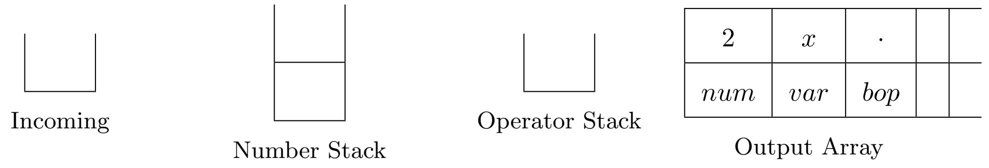
Since the operator stack is empty, we look at the number stack and notice that it contains a single number, 4, which we now know to be the result of our expression. If we were left with multiple numbers on the number stack, we would know there was an input error. Similarly, if we had a single number left on the number stack with a binary operator left on the operator stack, we would know an input error occurred. The algorithm described above is known as the *Shunting Yard Algorithm*, because the algorithm handles tokens the same way a railroad shunting yard handles incoming trains. Both assign meaning to each individual car or token, then place it onto the proper track or stack based on its assigned meaning and the yard's contents. However, note that during the process we evaluated the contents of our operator and number stack, replacing values on the number stack with the result of an evaluated sub-expressions. If we are faced with an expression containing variables, we can't possibly ask the computer to evaluate the result of "$x+2$", and therefore must use a different strategy.

2.2 Conversion of Variables

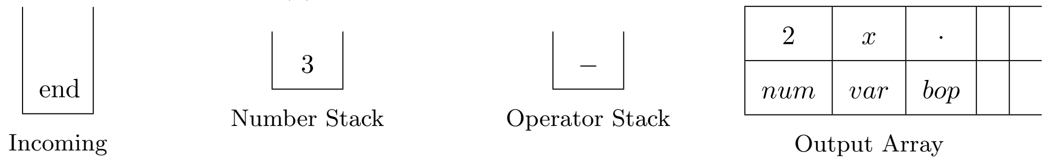
We can see in the previous example we evaluated our expression as necessary to preserve the order of operations, but if our expression contains variables, this method breaks down. To get around this problem, we can output a given sub-expression, instead of attempting to evaluate it. By outputting sub-expressions in this way, we can store the function in a computer-friendly format without compromising accuracy or syntax. Another point to consider, is that by parsing variables into our application, we can drastically cut down on computing time, because we aren't required to re-parse our expression for every new value of x. We can now use the SYA on a simple function, $f(x) = x \cdot 2 - 3$. Figures 8a, 8b, 8c, and 8d illustrate the way in which the string "$x \cdot 2 - 3$" can be turned into a postfix equation array.



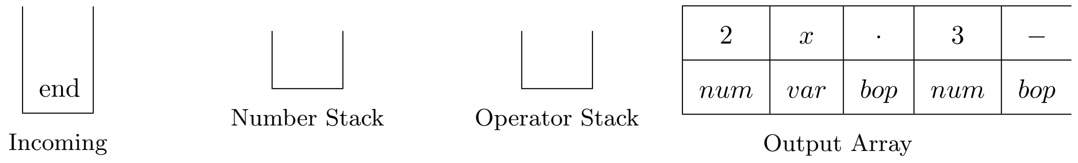
(a) First Output Step



(b) First Output Step Complete



(c) Second Output Step



(d) Second Output Step Completed

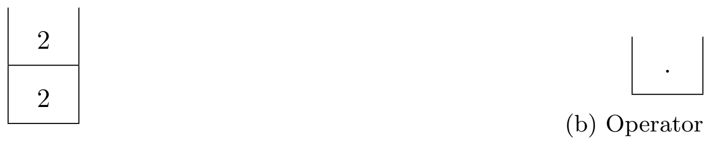
This string of characters is thus stored in our application as an array representing the inputted expression. Our array represents this function in Postfix notation, because the computer can evaluate the expression literally and linearly based on a postfix representation. Now that we can convert an equation from human format to computer format, we can look at ways of using these representations to evaluate functions of variables within the computer.

3 Evaluating Functions

Now that we have a way to represent expressions within a software application, we can use our previously established algorithms to evaluate them. We can eventually use these same techniques to calculate functions of variables.

3.1 Numerical Expression Evaluation

Suppose we have an expression, say "$2 \cdot 2 - 3$" that we want to represent programmatically. We would begin by using the SYA to create a postfix expression. This expression is relatively simple, and doesn't require many steps, the first of which can be seen in Figure 9.

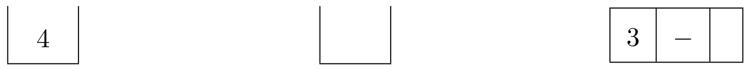


(a) Number Stack

(b) Operator Stack

Figure 9: Number and Operator Stacks

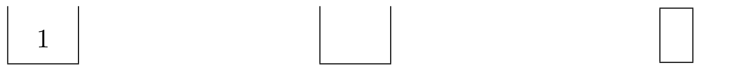
However, an astute observer might note that our stacks do not contain the entire expression currently. We still have “−3” left of our expression that isn’t currently accounted for. This is because according to the Precedence Table in Table 3, we cannot push a “−” onto a “·”, which means we have to flush our stacks until “−” has a higher precedence than whatever exists on the stack. In this case, we will evaluate the current contents of our stacks, and treat the result as a new number. This is to say we will now have stacks and output that look like those found in Figure 10.



(a) Number Stack (b) Operator Stack (c) Input

Figure 10: Number and Operator Stacks with Input

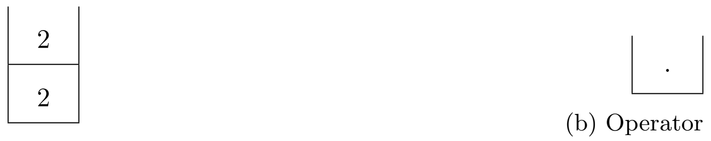
The algorithm is now faced with a singularly populated number stack and an empty operator stack, which means we can simply shove everything else onto our stacks. We now reach the end of our input, and have 2 numbers with a binary operator. So we pop both numbers, 3 followed by 4, and our operator, −, and know to evaluate “4 3 −” or $4 - 3$, which leaves us with a final answer of 1, which is now the only element on any stack, as illustrated in Figure 11.



(a) Number Stack (b) Operator Stack (c) Input

Figure 11: Number and Operator Stacks with Input

This application of the SYA works well for evaluating purely numerical expressions, but we can’t always evaluate “on-the-fly”. Thus we have to use an output based model of the SYA, instead of an evaluation based model. Below is a re-work of the above example, using output instead of evaluation for stack de-population.

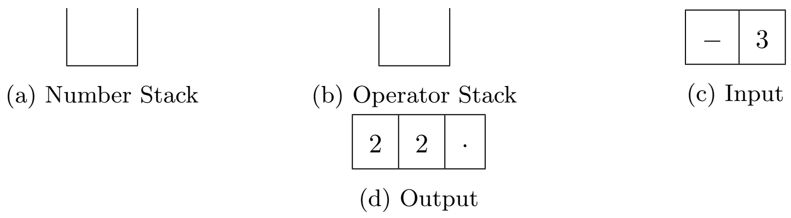


(a) Number Stack

(b) Operator Stack

Figure 12: Number and Operator Stacks

If we want to store the expression in its entirity, we can't simply evaluate $2 \cdot 2$ and shove the value back onto the stack, because we would lose part of the expression. If we place each token into an array, in postfix notation, we can preserve our expression for later evaluation. In this case, we would pop both of our 2's, and the $\cdot$, and place them into the first 3 cells of an array, which will look like Figure 13. We then push the remaining "+ 2" onto their appropriate stacks, before reaching the end of our input, at which point we pop them and place them into the array.



(a) Number Stack

(b) Operator Stack

$-$	3

(c) Input

2	2	$\cdot$

(d) Output

Figure 13: Number, Operator Stacks, Input & Output

We now push our input onto stacks, reach the end of our input, pop everything off of our stacks, and place the now parsed input into the array. This results in the following postfix expression as an array that looks like the one in Figure 14

2	2	$\cdot$	3	$-$

Figure 14: Postfix Array

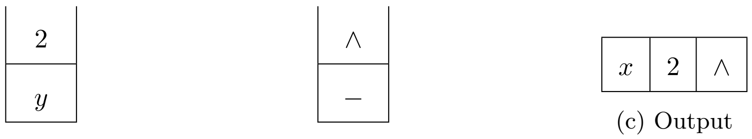
While this array can be interpreted easily by a human, who would note that it contains 3 numbers with 2 binary operators, and likely knows what each symbol implies. Unfortunately, computers don't have any inherent interpretation of these characters, so providing flags or character classes to go along with each token would be useful. With this in mind, we can create an array with 2 rows, and keep track of what each symbol in each position represents. The resulting array will look like the one found in Figure 15, where *num* stands for numbers, and *bop* holds the place of binary operators.

2	2	$\cdot$	3	$-$
num	*num*	*bop*	*num*	*bop*

Figure 15: Postfix Array

3.2 Accounting for Variables

We have now completed examples of the SYA using numbers, but what if we want to store a function that uses variables instead of literals. With our Array model of equation storage, there is practically no difference between a variable and a number. However, we want to keep track of which position in the array is a pre-defined number and which is supposed to be a variable. Using the idea of a 2-dimensional array, we can add new flags $varX$ or $varY$ to ensure we know precisely where our equation has variables. Now, suppose we are given an expression, say "xˆ$2 - y$ˆ2". We can place all of "x ˆ2" onto stacks before being forced to shunt this subexpression to our array as seen in Figure 17. We can then push the remaining input onto each token's respective stack, not needing to flush the stacks. After reaching the end of our input, we will have stacks and a postfix array that look like those in Figure 16.



(a) Number Stack (b) Operator Stack (c) Output

Figure 16: Number, Operator Stacks, & Output

We then reach the end of the expression, and output the stacks to our array, which will now look like Figure 18.

x	2	ˆ
$varX$	num	bop

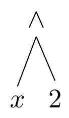
Figure 17: Partial Postfix Array

x	2	$\wedge$	y	2	$\wedge$	$-$
$varX$	num	bop	$varY$	num	bop	bop

Figure 18: Complete Postfix Array

This is now the entire expression converted from infix to postfix while accounting for the use of variables.

This method will be used for our purposes; however, there are other ways we can represent an expression within a computer. We can instead use a relatively basic binary tree to represent our entire function, or a parse tree, like the one found in Figure 2. Using the previous expression, "xˆ$2 - y$ˆ2", we can create a parse tree where each node represents a token of the expression. With this model, we can store flags like in the array model, but we also have much more flexibility for storing syntactical information. Since this method of expression parsing is simply a new model of storage, we can use the SYA to help us create our tree. We therefore proceed exactly as with the previous example, but when prompted to flush the stacks, instead of outputting to a postfix array, we output to a binary tree. The first output would then look like the tree in Figure 19, with empty stacks and the remaining input.



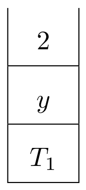
(a) Parse Tree for "xˆ2"

$-$	y	$\wedge$	2

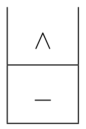
(b) Remaining Input as Array

Figure 19: Parse Tree & Input

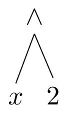
We now continue with the SYA, placing an indicator or index T_n back onto the number stack so that we can preserve order of operations, and keep track of sub-trees. With the indicator in place, we run the rest of the input through the algorithm without having to flush the stacks. Our stacks and tree will look like Figure 20.



(a) Number Stack



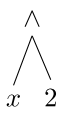
(b) Operator Stack



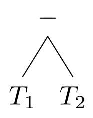
(c) Partial Parse Tree, T_1

Figure 20: Number, Operator Stacks, & Parse Tree

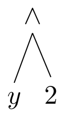
Now that we've reached the end of our input, we flush our stacks to our storage model, but we now must create a tree for each sub-expression. That is to say, we pop the $\wedge$, the 2, and the y from our stacks, and place them into tree T_2, which we place onto the number stack. We now have 2 "numbers" on our number stack, and one remaining operator, so we create another tree using T_1, T_2, and $-$. The process will result in empty stacks, and 3 sub-trees, looking like those in Figure 21.



(a) Parse Tree, T_1



(b) Parse Tree, T_3



(c) Parse Tree, T_2

Figure 21: Three Parse Trees

We can combine the trees to create on large parse tree, as seen in Figure 22, or leave the sub-trees as-is, since we aren't losing or gaining any information either way.

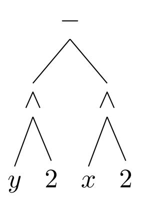


Figure 22: Concatenated Parse Tree

These two methods both have their merits, but as stated earlier, we will be mainly focusing on a postfix array model of storage. With that in mind, we will now look our the main motivation for storing expressions in this form: function evaluation.

3.3 Function Evaluation

By storing a postfix array within a computer's memory, we're giving it the ability to read through the array and calculate the result relatively quickly. If we have variables in the postfix array, we have to replace them before we can evaluate the stored expression. However, what if we want to plot or visualize a function in some way? We'll need to replace the expression's variables hundreds, thousands, even millions of times depending on the complexity of our equation and the domain. When we take our postfix array, replace the $varX$ and $varY$ slots with specific values, evaluate the expression, and receive a result, we're left with an array containing a single element: the result. However, if we have to evaluate our postfix expression millions of times, erasing it on the first run won't get us very far. Instead, we are forced to copy the array every time we want to evaluate the expression at for a specific value of x or y. This allows us to parallelize the process of calculation though, which can drastically speed up total time required to plot or visualize a function. The entire process in code would look something like the code found in Figure 23.

```
Parse Expression
For x = 1 .. n
    For y = 1 .. k
          tempArray = PostfixArray
          Evaluate tempArray
          Plot Result
          .
          .
          .
    End For
End For
```

Figure 23: Pseudo-Code for Repeated Evaluation

When actually evaluating a given expression, we can linearly proceed through the associated postfix array with substituted variable values until we reach our first *bop* flag. When we run into this flag, we grab the preceding two numbers j_1 and j_2, that are guaranteed to exist by the SYA, or else the algorithm would have failed. We then note the definition of whatever binary operator token is in that slot, and evaluate $j_1 bop j_2$, placing the result back into the array, replacing the exact 3 slots we just vacated with our 1 result. This ensures we don't lose any of the syntactical relationships previously analyzed. Figure 24 shows how this process works on a small scale.

j_1	j_2	bop	etc
num	num	bop	etc

$j_1 bop j_2$	etc
num	etc

Figure 24: Postfix Array Evaluation

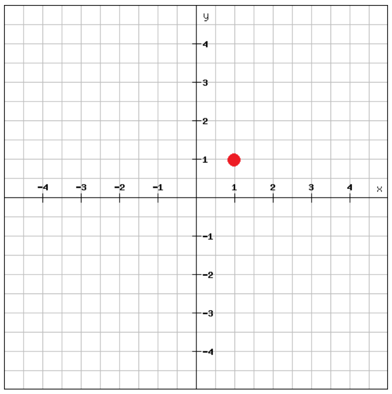
We can now evaluate a given function of variables as many times as needed to accurately visualize or plot it. For our purposes, we will be looking at ways to visualize the complex plane, and functions of the form $\phi : \mathbb{C} \to \mathbb{C}$.

4 Complex Mapping

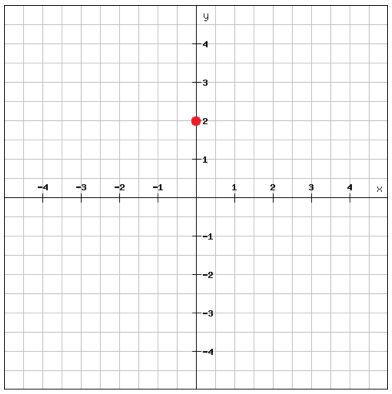
Any given complex number z is made up of a real part and an imaginary part, which is to say that any complex number $z = a + bi$. We can then consider the complex plane, made of a real axis and an imaginary axis, where any given z can be plotted to the point (a, b). Based on this understanding, we can look at the way in which a function $f(z) : \mathbb{C} \to \mathbb{C}$ maps a section of the complex plane.

4.1 Mapping a Point

Even though each individual complex number z is made up of a real half and an imaginary half, when we use a function to map the points, we must map the entire number z to an entirely new complex number c_1. We now have a point (a, b) in the domain of $f(z)$, and associated point (u, v) in the co-domain of f. This is to say, our function f maps the point (a, b) to the point (u, v), both in the complex plane. For example, if we have some function $f(z) = z^2$, we would have $f(a + bi) = (a + bi)^2$. If we expand this equation and reform it into its real and imaginary parts, we get $f(a + bi) = (a^2 - b^2) + (2ab)i = f(z)$. To see how this function maps the point $(1, 1)$, which is to say $f(1 + i)$, see figures 25a and 25b.



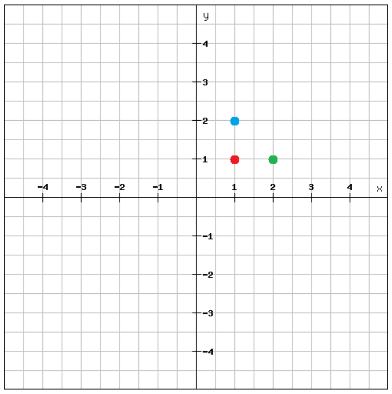
(a) $1 + i$ represented in the complex plane



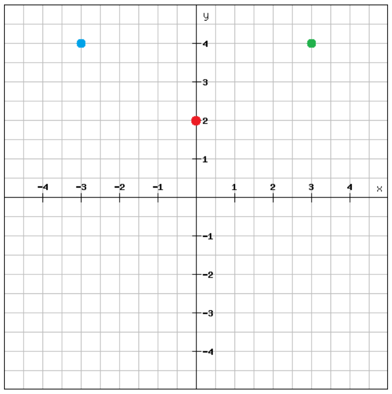
(b) $f(1 + i)$ represented in the complex plane

Figure 25: Mapping a Point

Now that we've mapped a single point using a complex function, we should look at what happens to multiple points. Since our mapping is essentially a function of two variables, we can look at $f(a, b)$ for a number of points to try and understand how exactly the mapping manifests itself in the plane. This technique is similar to graphing a few values for some function $f(x)$ to understand its effect on x. If we look at figures 26a and 26b, we can see how this mapping moves these particular points around the plane, but it doesn't give us a particularly good understanding of the entire function.



(a) Three points represented in the complex plane



(b) $f(a, b)$ on 3 points represented in the complex plane

Figure 26: Three arbitrary points mapped by $f(a, b) = (a + bi)^2$

We need a better way of representing our mapping, because simply mapping point by point will likely not reveal much about the way our function reshapes the domain. With this in mind let's look at the most common method of representing these complex functions within the complex plane.

4.2 Domain Coloring

Instead of thinking about the complex plane as strictly points in a two-dimensional space, we can instead think of the plane as a color wheel, where each point has an associated color value. Based on Frank A. Farris' model of complex visualization, which appears in his review of Tristan Needham's **Visual Complex Analysis**, one can visualize a complex-valued function in the plane by computing the value of $f(a, b)$ for each point in the domain. The resulting complex point $(u, v) = f(a, b)$ will have an associated color value, based on the color wheel found in Figure 27.

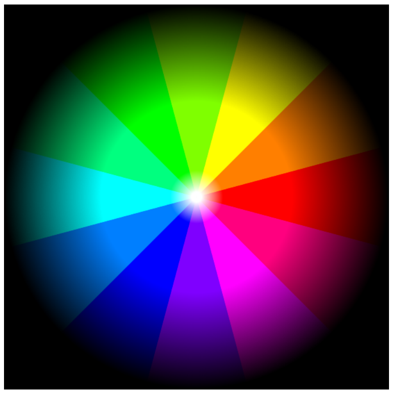
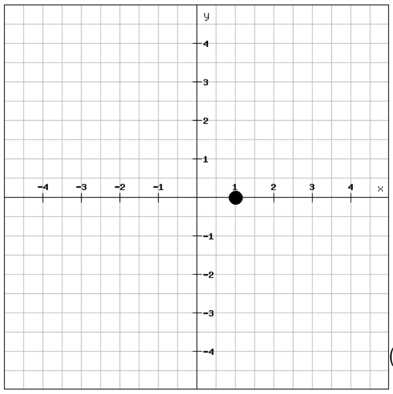
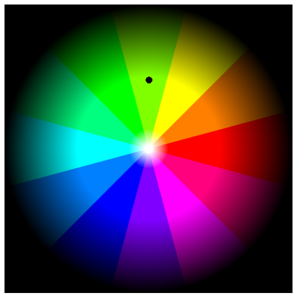


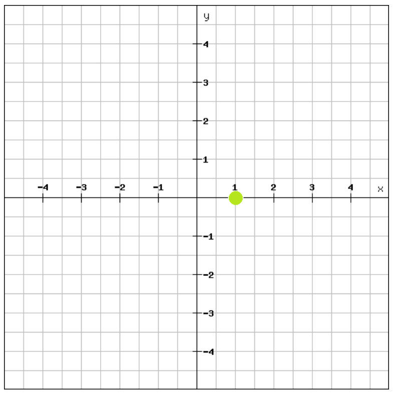
Figure 27: Complex Color Wheel

In Figure 27, we have placed red along the first cube-root of unity, with green and blue comprising the other two cube-roots. While this is just one particular model and selection of colors, having the three primary computer colors lie on the third roots of unity adds a nice mathematical touch to the color wheel. This approach also allows for a more careful analysis of where these third roots are mapped under our function. However, the main purpose is to illustrate the effects of a complex-valued function on the complex plane. Let's work through an example of this model. If we were to map the point $(1, 1)$ using our function $f(z) = z^2$, we must first split this function into its real mapping and its imaginary mapping. Since we know previously that these two halves are $(a^2 - b^2)$ and $(2ab)i$, we have $U(a, b) = (a^2 - b^2)$ and $V(a, b) = (2ab)$. If we now map the point $(1, 1)$, we have $U(1, 1) = (1 - 1) = 0$ and $V(1, 1) = (2 \cdot 1 \cdot 1) = 2$. We now see that the point $(1, 1)$ maps to the point $(0, 2)$, so we would color the point $(1, 1)$ light green, because its image lies directly on the positive imaginary axis. See Figure 28b and Figure 28c for an illustration of the concept.



(a) $(1, 0)$ in the Complex Plane


(b) $f(1, 0)$ Plotted on Color Wheel



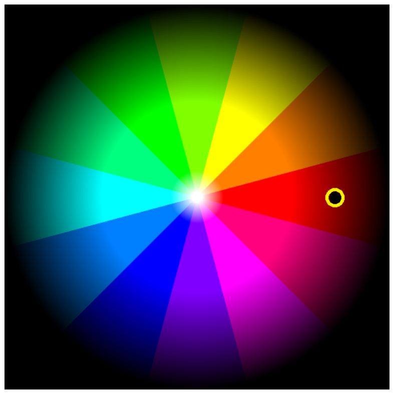
(c) $(1, 0)$ Colored According to $f(1, 0)$

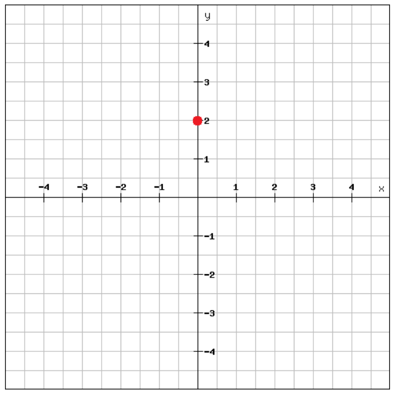
Figure 28: Basic Domain Coloring

This method of complex visualization will be used for our purposes due to its relative simplicity, but there are other methods. The other most common method is to map colors from the domain to the co-domain, and plot the co-domain. This particular visualization model, known as color mapping, has some theoretical and computational problems, but produces a uniquely different effect, and so should be considered.

4.3 Mapping Colors

Instead of mapping a point, and coloring the original point based on the computed point's value, we can color the destination based on the original. That is to say, $(u, v) = f(a, b)$ is colored with (a, b)'s color. Figure 29 shows this model in action.


(a) $(1, 0)$ on the Color Wheel



(b) $f(1, 0)$ Colored By $(1, 0)$

Figure 29: Mapping a Color

For our purposes we will choose to use the domain coloring method, because direct color mapping creates many problems associated with color theory that are not easily addressed in this scope. Due to the fact that complex functions are not inherently invertible, which is to say that some points in the co-domain will be have multiple pre-images in the domain, there's no way to guarantee we won't map two colors to the same point. One can see how this would create problems for a color mapping model of visualization, because if we have multiple points mapping to the same place, what color do we make the destination? That is to say, if we have some $f(a_1, b_1) = (u, v) = f(a_2, b_2)$, what color do we assign to the point (u, v)? There are a few different methods to solve the problem, including nearest-neighbor coloring and weighted average coloring. However, our problems don't stop here, because complex functions also don't aren't one-to-one necessarily. This means we might have points in the image that have no pre-image, and would thus be blank if we were mapping colors. This is why nearest-neighbor coloring schemes become important, because to get an accurate representation of a function's effect on the complex plane we need to have a cohesive visualization, not something that looks like spray paint. Using nearest-neighbor coloring along with weighted averages based on various factors (number of pre-images, color neighborhood, pre-image color frequency, etc), one can create a complex function visualizer based on color mapping. Similar approaches are used in topographical color mappings, as can be seen in Dave Hale's *Image-guided blended neighbor interpolation* **??**. Since color theory adds one too many extra layers of complexity to this particular application, we will take a more detailed look at Domain Coloring.

5 Software Based Complex Visualization

We will now put together all the concepts previously covered into a cohesive, example-driven, walk-through of how we can create an application to color the domain of a complex function.

5.1 Parsing User-Input to Postfix Array

When presented with input from the user, our application will begin by using the SYA, while simultaneously interpreting characters into tokens. This can be achieved by examining each character of the input, and comparing it with a table of acceptable tokens. If we are presented with a character that doesn't exist within our dictionary of acceptable mathematical inputs, we can include the next character in our interpretation. This is to say we would interpret "$\sin x$" beginning with "s," noticing "s" is not an acceptable character, and attempting to interpret "si", which is also not included in our dictionary. We then reach "n", and interpret "sin" appropriately as an entire token itself, which we push onto the operator stack as a unary operator. We now have an input of "(x)" and can use the SYA in the way we are familiar to complete the process. However, since looking at $f(z) = \sin z$ adds too much complexity to our calculations, we will look at the more simplistic example, "$f(z) = z^2 = (x + yi)^2$", which can be represented by the real-valued mapping "$u(x, y) = x^2 - y^2$" and the complex valued mapping "$v(x, y) = 2xy$". We can use u and v to map every point in our domain, and color the point based on its mapping. Before we can do that though, we have to parse our two equations. Having previously worked through similar examples, we won't spend too much time on the algorithm. Needless to say, we can store the resulting postfix expression arrays as separate entities within the application. We now have two arrays that look like those found in Figure 30.

a	2	$\wedge$	b	2	$\wedge$	$-$
$varX$	num	bop	$varY$	num	bop	bop

(a) $u(x, y)$ Postfix Array

2	x	$\cdot$	y	$\cdot$
num	$varX$	bop	$varY$	bop

(b) $v(x, y)$ Postfix Array

Figure 30: Postfix Arrays for "User" Input

Now that we have the expressions stored within our application, we can evaluate them based using repeated substitution of variables to produce plots.

5.2 Evaluating Equation Arrays

Because our method for evaluating a postfix array involves the collapsing of cells, imagining how one might substitute a new x or y value into the array can be difficult. In fact, so difficult that the easier solution is to not pursue that method of evaluation. Instead, within the memory of our application, we create a temporary copy of each array with the value for x and y substituted for the variables themselves. This way, we can do the actual evaluation of our arrays in the way we are accustomed, without compromising the original function. For our application, we're going to loop through every individual point of the domain, based on individual pixels' (x, y) coordinates on the screen. We will map the pixel values (P_x, P_y) from

$P\_x = 0 \ldots Image\_Width$ and $P\_y = 0 \ldots Image\_Height$ to a square domain, such that the maximum height and width are equal, and the minimum height and width are equal to the negative of the maximum. For example, we can map the points $(0,0) \ldots (600,600)$ to a portion of the complex plane such that $|y| \leq 5$ and $|x| \leq 5$, for every complex number $z = x + y \cdot i$, represented in the plane as (x, y). This is to say that we will map our set of points $(0,0) \ldots (600,600)$, where each (x, y) is integer valued, to the set of points $(0,0) \ldots (\pm 5, \pm 5)$ where every (x, y) is real-valued. So, to evaluate our equations $u(x, y) = x^2 - y^2$ and $v(x, y) = 2xy$, we begin by creating a temporary copy of both equation arrays. We can then evaluate each copy based on the mapping of our current P_x and P_y positions. The results will be in the complex plane, but not necessarily within our domain, which we can quickly see since $u(5, 0) = 25 > 5$. Unfortunately, computers don't have a particularly great understanding of the complex plane, which is to say we will have to map our results to a more image-friendly format.

5.3 Result Mapping and Domain Coloring

Every point (P_x, P_y) will be mapped to some point (x, y) in the complex plane. This point will be exactly $\sqrt{x^2 + y^2}$ units away from the origin, and will be $\theta = \arctan y, x$ radians from the real number line, with. We can thus represent the complex plane using polar coordinates, where every point can be represented as (r, θ) instead of (x, y). This becomes of use to use when we want to use the color-wheel model for complex visualization, because now we can easily map from a point to the color wheel and back again. With this as our goal, we can use the Hue, Saturation, Brightness color model for our image. This particular model differs greatly from the classical ordered triple R, G, B, but is also itself an ordered triple. The first argument, the Hue, determines an angle around a cylinder, while the Saturation determines the radius of the point, and the Brightness determines the height. Saturation ranges from full color to full white, and brightness ranges from full darkness to none. Figure **??** shows how this color model can be represented, though since as all colors get darker they begin to look the same, it is represented as a cone rather than a cylinder. [5]

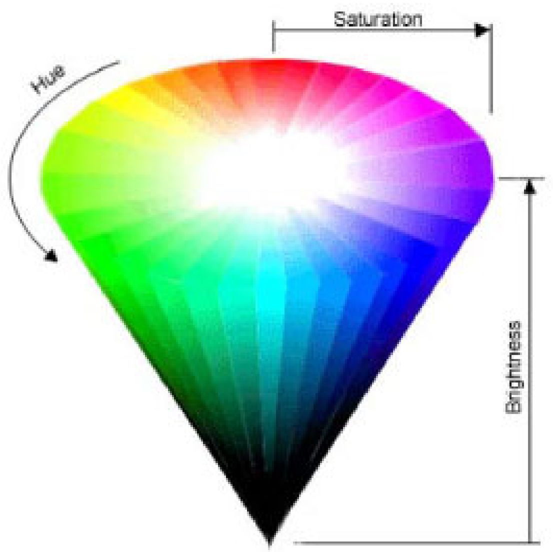


Figure 31: HSB Cone

Using this model for coloring a point, we can determine the Hue based on our θ value, with Saturation and Brightness based on r. We can divide our color-wheel into 3 sections, an inner wheel, a middle wheel, and an outer wheel. This way, we fade from 0% saturation to 100% saturation from the origin to $r/3$, stay at 100% saturation and brightness for the middle 3rd, and then fade from 100% brightness to 0% at the edge of our domain. Thus we can have a mapping from the complex plane to the complex plane to the complex color wheel, all while keeping track of which point in the domain we mapped. We can then color our original (P_x, P_y) point with whatever color is associated with θ, which will be rounded to the nearest 12^{th} root of unity; which is to say we are looking at 12 chunks of color rather than a true continuum. Thus, we are coloring the entire domain of our function based on each point's image, which is where the method gets its name. Since each point in the domain will have its own color, we don't have to worry about collisions, and we don't have to worry about color blending, because there will be no empty pixels.

5.4 Results of Domain Coloring

The resulting color patters based on visualizations of complex-valued functions can be aesthetically pleasing, but also can provide valuable insight into the behavior of a given function. For example, we know there is a complex asymptote anywhere we find a black hole, because as points get farther from the origin they get darker. Similarly, we know where zeros exist based on white holes, because the closer a point is to the origin, the white it gets. We also track where each 12^{th} root of unity ends up, which can be useful for function analysis, where there are more than just real square-roots of 1. Frank Farris offers some intricate examples of domain coloring, like providing the domain coloring for $f(z) = (z^2 - i)/(2z^2 + 2i)$, which can be seen in Figure 32.

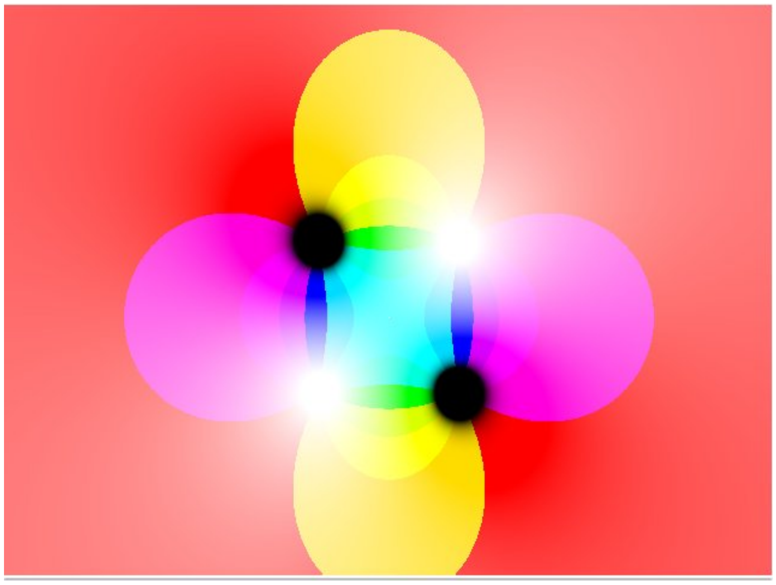
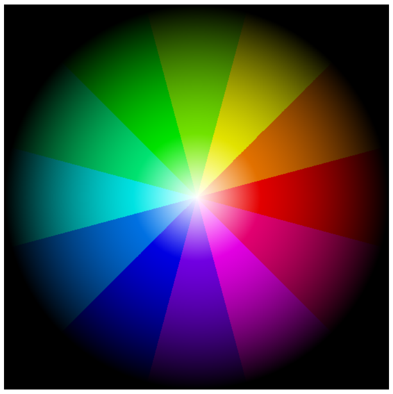


Figure 32: Domain Coloring of $f(z) = (z^2 - i)/(2z^2 + 2i)$

We can look at a more simple example, like $f(z) = z^2$, which we have broken down before. We can now look at the way in which the entire domain is mapped by this function. Intuitively, we can imagine this mapping might warp the domain in a parabolic way; however, since we're using a polar model of representation for the complex plane, we don't actually see anything that looks parabolic. Instead, the color wheel is inverted upon itself, such that we have our 12 colors repeated twice around what our circle. We also have a larger number of points trending towards 0, which is to say the white center of our color wheel expands to enclose more of the entire wheel. We also head towards infinity much faster, as is indicated by the decreased radius of our circle. Figures 33a and 33b show the original color wheel and the color wheel mapped by $f(z) = z^2$ respectively. We can see why this function maps our domain in this way by looking at the representation of our mapping $f(z)$ in polar coordinates. By substituting and converting from Cartesian coordinates to Polar, we can see that $f(z)$ has polar representation $f(\theta) = 2\theta$ and $f(r) = r^2$. We can now intuit our result in Figure 33b, where we see that we have each 12th root of unity reflected across the origin, which could be intuited from the fact that $f(\theta) = 2\theta$. We can also see that the interior wheel, where points are trending towards zero, is larger, due to the fact that $k^2 < k$ when $|k| < 1$. Similarly, our color wheel has a smaller radius, because for all $k > 1$, $k^2 > k$, which is to say everything trends towards infinity even faster than previously, reflected in the fact that our wheel becomes darker faster relative to the original [2].


(a) $f(z) = z$

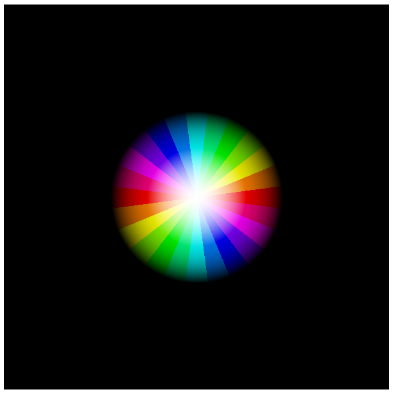

(b) $f(z) = z^2$

Figure 33: Domain Coloring of $f(z) = z^2$

This application of complex analysis and computer algorithms can hopefully help mathematicians more easily comprehend complex mappings. Having a piece of software create visualizations based on user input allows for a more interactive learning environment compared to text on paper or screen, thus making this type of application important.

6 Conclusions

Many of the books and articles I referenced included theory sections, but lacked any concrete application. Similarly, many of the applications lacked any significant theory, which means that most gaps between theory and application had to be bridged in this paper. Due to the limitations of time and medium, the application I created to accompany this paper lacks some desired functionality, though still works as intended. Creating a complete mathematical evaluator by hand quickly becomes tedious, and accounting for every operator becomes difficult when there isn't an equivalent expression within the programming. Most common operations and syntax should be acceptable, but some escaped interpretation within the program. Another problem associated with my application is that I've had to interface a standalone applet with a web-page handling user input, which makes for some difficulties in communication and data storage. Finding the proper way to interpret user input in the web page before transferring the equation array to the standalone application is difficult, but not impossible. Many pre-existing, robust libraries exist that provide expression parsing functionality, in practically every programming language imaginable. Unfortunately, none of these libraries were easily cannibalized, so writing an expression parser within the web page would be highly unfeasible. However, using a library we could convert user inputted equations into a postfix array with flags, and send the resulting array to the applet. Another piece of functionality that would be nice to include is the ability to parse equations of the form $f(z)$, but expanding $f(z)$ into $g(a, b) = f(z)$ is a difficult programming challenge when not mathematically impossible. With a deeper understanding of complex analysis, accounting for these impossible scenarios within an application would be possible, as would the expansion of complex-valued functions. However, these are all things left undone in this project. Fortunately, we've outlined all the theory to create a complex-valued function visualizer based on the domain coloring paradigm, and I've built a minimally functional application as a proof of that concept.

7 Code Appendix

```
static Token[] equationU;
static Token[] equationV;
int[] colorFreq;
PImage wheel;
PImage myImage;
int underCount = 0;
int tempWidth = 0;
int temptempHeight = 0;
static float maxmapX = -100000000;
static float maxmapY = -100000000;
static float minmapX = 100000000;
static float minmapY = 100000000;
final static int VAR_X = 10;
final static int VAR_Y = 11;
final static int NUM = 1;
final static int BIN_OP = 2;
final static int UN_OP = 3;
final static int CONDENSED = -1;
final static int EMPTY = 0;
static int curX = 0;
static int curY = 0;
int eqUSize = 7;
int eqVSize = 5;
float compX = 0.0;
float compY = 0.0;

Token[] tempU = new Token[100];

Token[] tempV = new Token[100];

boolean testFlag = true;

static float lowHue = -10000;

static float lowSat = -10000;

static float lowBrt = -10000;

static float hiHue = 10000;

static float hiSat = 10000;

static float hiBrt = 10000;

static float maxD = -10;

static int domain = 5;void setup()

  size(600, 600);


  int tempWidth = width;

  int tempHeight = height;

  colorFreq = new int[tempWidth*tempHeight];

  equationU = new Token[100];

  equationV = new Token[100];

  for (int i = 0; i < 100; i++)

  {

    equationU[i] = new Token();

    equationV[i] = new Token();

  }

  colorMode(HSB, 360, 1, 1);


  for (int kol = 0; kol < (tempWidth*tempHeight); kol++)

  {

    colorFreq[kol] = 0;

  }

  background(0);

  equationU[0] = new Token("x", VAR_X);

   equationU[1] = new Token("2", NUM);
```

```
 equationU[2] = new Token("^", BIN_OP);
 equationU[3] = new Token("y", VAR_Y);
 equationU[4] = new Token("2", NUM);
 equationU[5] = new Token("^", BIN_OP);
 equationU[6] = new Token("-", BIN_OP);


 equationV[0] = new Token("2", NUM);
 equationV[1] = new Token("x", VAR_X);
 equationV[2] = new Token("*", BIN_OP);
 equationV[3] = new Token("y", VAR_Y);
 equationV[4] = new Token("*", BIN_OP);
 int temp1size = 0;
while (equationU[temp1size].flag != 0)
{
  temp1size++;
}
eqUSize = temp1size;
int temp2size = 0;
while (equationV[temp2size].flag != 0)
{
  temp2size++;
}
eqVSize = temp2size;
println(eqUSize);
println(eqVSize);  myImage = createImage(600, 600, RGB);
myImage.loadPixels();
for (int loopX = 0; loopX < tempWidth; loopX++)
{
  for (int loopY = 0; loopY < tempHeight; loopY++)
  {
    int l = loopX + loopY * tempWidth;
    myImage.pixels[l] = color(0, 0, 0, 100);
```

```
float evalX = map(loopX, 0, tempWidth, -domain, domain);
float evalY = map(loopY, 0, tempWidth, -domain, domain);
for (int reset = 0; reset < 100; reset++)
{
  tempU[reset] = new Token();
  tempV[reset] = new Token();
}
for (int reset = 0; reset < 100; reset++)
{
  tempU[reset] = new Token(equationU[reset]);
  tempV[reset] = new Token(equationV[reset]);
}
float tempcompX = evalEqn(tempU, evalX, evalY);
float tempcompY = evalEqn(tempV, evalX, evalY);
if (tempcompX > maxmapX)
{
  maxmapX = tempcompX;
}
if (tempcompX < minmapX)
{
  minmapX = tempcompX;
}
if (tempcompY > maxmapY)
{
  maxmapY = tempcompY;
}
if (tempcompY < minmapY)
{
  minmapY = tempcompY;
}
float tempD = sqrt(tempcompY*tempcompY + tempcompX * tempcompX);
if (tempD > maxD)
{
      maxD = tempD;
    }
  }
}
myImage.updatePixels();
println("SETUP COMPLETE");

myImage.loadPixels();

image(myImage, 0, 0);
println("myImage created");
while (curX < 600)
{
  for (curY = 0; curY < 600; curY++)
  {
    for (int reset = 0; reset < 100; reset++)
    {
      tempU[reset] = new Token();
      tempV[reset] = new Token();
    }

    for (int temp = 0; temp < eqUSize; temp++)
    {
      tempU[temp] = new Token(equationU[temp]);
    }
    for (int temp2 = 0; temp2 < eqVSize; temp2++)
    {
      tempV[temp2] = new Token(equationV[temp2]);
    }

    float mapY = map(curY, 0, tempHeight, -domain, domain);
    float mapX = map(curX, 0, tempWidth, -domain, domain);
    compX = evalEqn(tempU, mapX, mapY);
    compY = evalEqn(tempV, mapX, mapY);

    float newmapX = map(compX, -domain, domain, 0, tempWidth);
    float newmapY = map(compY, -domain, domain, 0, tempHeight);

    float arg2 = PI+(-1*atan2(compY, compX));
    float arg = round(arg2/(PI/6.0))*(PI/6.0);
    float radius = sqrt(compX * compX + compY * compY);
    float diameter = map(sqrt(compX * compX + compY * compY), 0, domain, 0, tempWidth);

    int pixelTo = 0;
    int pixelFrom = 0;
    pixelFrom = max(0, min(round(curX + curY*tempWidth), tempWidth*tempHeight-1));

    int temppixTo = 0;
    temppixTo = (int)((newmapX + newmapY * tempWidth));

    boolean firstFlag = (boolean)(temppixTo < (tempWidth * tempHeight));
    boolean secondFlag = (boolean)(temppixTo > 0);
    boolean changePix = (boolean)(firstFlag && secondFlag);
    changePix = true;
    if (changePix == true)
    {
      myImage.loadPixels();

      colorFreq[pixelFrom]++;

      float myHue = (degrees(arg)+180)%360;


      myImage.pixels[pixelFrom] =  color(myHue, ((3*diameter)/(tempWidth)),
1-pow((diameter/(tempWidth)),2));
```

```
      }
      myImage.updatePixels();
    }
    curX++;
  }
  if (curX >= 600)
  {
    myImage.updatePixels();
    myImage.loadPixels();
    image(myImage, 0, 0);
  }
void draw(){}
float evalEqn(Token[] equation, float x, float y)
  boolean evalFlag = true;
  int i = 0;
  int flag = equation[i].flag;
  while (flag != EMPTY)
  {
    if (flag == (VAR_X))
    {
      equation[i].token = ""+x;
      equation[i].flag = NUM;
    }
    if (flag == (VAR_Y))
    {
      equation[i].token = ""+y;
      equation[i].flag = NUM;
    }
    i++;
    if (equation[i] == null)
    {
      flag = EMPTY;
    }
    else
    {
      flag = equation[i].flag;
    }
  }
  while (evalFlag)
  {
    evalFlag = false;
    i = 0;
    flag = equation[i].flag;
    while (flag != EMPTY)
    {
      int j = 1;
      int k = 1;
      if (flag == (NUM))
      {
      }
      else if (flag == (UN_OP))
      {
        while (equation[i-j].flag == CONDENSED) {
          j++;
        };
        equation[i].token = ""+evalUN(equation[i].token, float(equation[i-j].token));
        equation[i].flag = NUM;
        equation[i-j].token = ""+0;
        equation[i-j].flag = CONDENSED;
        evalFlag = true;
      }
      else if (flag == (BIN_OP))
      {
        while (equation[i-j].flag == CONDENSED) {
          j++;
        };
        while (equation[i-j-k].flag == CONDENSED) {
          k++;
        };
        String operator = (String)equation[i].token;
        float in2 = float(equation[i-j].token);
        float in1 = float(equation[i-j-k].token);
        float returned = evalBin(operator, in1, in2);
        equation[i].token = ""+returned;
        equation[i].flag = NUM;
        equation[i-j].token = ""+0;
        equation[i-j-k].token = ""+0;
        equation[i-j].flag = CONDENSED;
        equation[i-j-k].flag = CONDENSED;
        evalFlag = true;
      }
      i++;
      if (equation[i] == null)
      {
        flag = EMPTY;
      }
      else
      {
        flag = equation[i].flag;
      }
    }
  }
  int result_ind = 0;
  while ( (equation[result_ind].flag!=EMPTY) && ((equation[result_ind].flag!=NUM)))
  {
    result_ind++;
  }
  float toReturn = float(equation[result_ind].token);
  return toReturn;
```

```
float evalUN(String OP, float Value)
  float toReturn = 0.0;
  if (OP.equals("-"))
  {
    toReturn =  ((-1)*Value);
  }
  if (OP.equals("sin"))
  {
    toReturn =  sin(Value);
  }
  if (OP.equals("tan"))
  {
    toReturn =  tan(Value);
  }
  if (OP.equals("cos"))
  {
    toReturn =  cos(Value);
  }
  else
  {
    toReturn =  Value;
  }
  return toReturn;
float evalBin(String OP, float in1, float in2)
  float toReturn = 0.0;
  if (OP.equals("^"))
  {
    toReturn =  pow(in1, in2);
  }
  else if (OP.equals("+"))
  {
    toReturn =  (in1+in2);
  }
  else if (OP.equals("-"))
  {
    toReturn =  (in1 - in2);
  }
  else if (OP.equals("*"))
  {
    toReturn =  (in1 * in2);
  }
  else if (OP.equals("/"))
  {
    toReturn =  ((in1*1.0) / in2);
  }
  else
  {
    toReturn =  (-1.0);
  }
  return toReturn;
}public class Token
  public String token;
  public int flag;
  public Token()
  {
    token = "";
    flag = 0;
  }
  public Token(String in1, int in2)
  {
    token = in1;
    flag = in2;
  }
  public Token(Token inToken)
  {
    token = inToken.token;
    flag = inToken.flag;
  }
}
```

References

[1] Harvey Abramson. *Theory and Application of a Bottom-up Syntax-directed Translator*. Academic Press, Inc., 1973.

[2] John B Conway and John B Conway. *Functions of one complex variable*, volume 2. Springer, 1973.

[3] Frank A Farris. Visualizing complex-valued functions in the plane. *AMC*, 10:12, 1997.

[4] David Guichard. *Data Structures and Algorithms*. Whitman College, 2014.

[5] Tom Jewett. Color tutorial. *California State University, retrieved on*, 2009.

[6] Theodore Norvell. Parsing expressions by recursive descent. *Parsing Expressions by Recursive Descent*, 2001.