Fractal Voyager: A Web Application for Exploring and Studying Complex Dynamics

A Computer Science Honors SYE
Dakota Bryan
Faculty Advisor: Kevin Angstadt
St. Lawrence University
Department of Math, Computer Science, and Statistics

May 2023

Abstract

Complex dynamics is a field of mathematics that studies the behavior of iterated functions in the complex plane. The complex plane is the set of all numbers that have a real part and an imaginary part, so they can include the imaginary unit $i = \sqrt{-1}$. Applying an iterated function to numbers in the complex plane often produces extraordinary two-dimensional fractals. These images are crucial to the study of complex dynamics, as is interacting with the fractals and changing how they are generated to best understand complex numbers and functions. Existing software available to generate these fractals either supports limited features or is incompatible with modern operating systems.

We present Fractal Voyager, a web application to generate fractals based on custom scripts entered by the user that represent iterated functions. The application takes scripts written in a custom complex dynamics language and converts them into code that can be run to create fractal images. Along with generating fractals, the web app provides the ability for users to interact with a fractal in ways such as: editing parameters, exploring the fractal (zooming and panning), and generating other fractals based on iterated functions that are related to the initial function. Fractal Voyager takes advantage of Web Assembly, which allows for the computationally intensive task of generating fractals to outperform pure JavaScript, the default language of the web. The application will allow mathematicians studying complex dynamics to research a wide variety of fractals conveniently on the web and can be a useful learning tool in teaching complex dynamics.

1 Introduction

A *fractal*, in the field of complex dynamics and for the scope of this report, is a visual display of how iterative functions behave in the complex plane. True fractals, in the mathematical sense, exhibit self-similar properties across all scales. Thus, the fractals being discussed here are *pseudo-fractals* based on complex dynamics functions,

but for the remainder of this report we will refer to these simply as fractals. In complex dynamics, there are two distinct types of functions which are iterated to generate fractals: those that create a *parameter plane*, and those that generate a *dynamical plane*.

Functions which generate a parameter plane iterate a variable (e.g. z) starting at a *critical point* (e.g. $0 + 0i$) until this variable meets a certain condition. We can define the iterative values of the variable as the *critical orbit*. The condition on the variable is often the critical orbit converging to zero or infinity, remaining bounded, or could be the variable no longer changing, or even comparisons on variables such as one being greater than another. Parameter plane fractals also include a parameter (e.g. c) which changes with respect to the complex plane. The second type of fractal generating function, those which create dynamical plane, iterate a function until a condition is met on a *variable point* (e.g. z) whose starting value changes with respect to the complex plane. Therefore, the difference is: parameter plane functions include two parameters, one is an iterated critical point, and the other changes with respect to the complex plane, while dynamical plane functions include only one parameter which is iterated and changes with respect to the complex plane.

Naturally, these fractals are thought of as sets. The set for a given fractal are the numbers which when entered into the function meet the condition for the fractal. These sets can be colored to create stunning images. Points, or numbers, outside the set are colored black, while points inside the set are colored along a gradient based on how quickly the condition is met. The brighter the color, the longer it took for the condition to be met.

Another important aspect of interest in complex dynamics is the *orbit* of a point in the dynamical plane. This tracks values of the iterated variable in a dynamical plane function after each iteration of the function when a particular number is the input. Thus, a complex number's orbit for a dynamical plane function is a sequence of numbers. This sequence can be plotted onto the complex plane as a sequence of line segments "on top" of the dynamical plane for the particular function. An orbit's plot can provide insight into the nature of a dynamical system as we can see how points behave in the function in more detail than the fractal coloring.

An application to generate fractals based on a simple scripting language which can define a function and condition to generate a fractal is valuable to the mathematical community. Mathematicians studying complex dynamics need the ability to look at fractals based on functions and conditions, so having a scripting language to generate fractals takes away the need for people to code their own fractal explorer, along with the specific code to generate the fractal they are studying. Looking at orbits, generating both types of fractals from scripts, and generating dynamical space fractals based points and parameter plane fractals are features that aid in the development of the field of complex dynamics.

Applications which generate complex dynamics fractals exist but support insufficient features or are incompatible with modern operating systems. There is at least one application, FracalStream [10], which supports a custom scripting language and many features including editing parameters, generating dynamical plane fractals based on

parameter plane fractals, and interacting with the fractal, but only runs on older (~2008) Mac operating systems. As far as we can find, there are no applications on the web which use a custom scripting language to generate fractals. The applications which exist on the web only support a limited suite of complex systems which can be explored, which is insufficient for mathematicians wishing to study complex dynamics as they could not be able to see the fractal from the function they wish to study. Web applications are preferred over native one's as they are cross-platform, supported across multiple devices, are generally more flexible and accessible, and have fewer problems with obsolescence. Therefore, an updated application on the web which supports features similar to FractalStream, including a scripting language, would be helpful to the field of complex dynamics as mathematicians rely on these applications in their study.

In this project, we created a web application to aid in the study of complex dynamics. It includes a custom high-level complex dynamics scripting language, *cdScript*, adapted from the language used by FractStream, which was inspired by FractalAsm [11]. Using this language, users enter scripts which represent iterative functions and conditions to generate the fractals. The application includes several core features. Users may edit options which slightly change how the fractal is generated, including what and how many colors to use, the cutoff value when a number is considered infinity, and many others. Exploring the fractal is also supported with zooming and panning, along with the feature of dragging a box to zoom in on that section of the fractal. Dynamical plane fractals and orbits can be generated based on user inputted values and a given parameter or dynamical plane fractal, respectively. Scripts allow for the creation of both parameter plane and dynamical plane fractals. If a user clicks on a point in the parameter plane, a dynamical plane fractal will be created by fixing the clicked "number". In a dynamical plane or orbit, a click will generate and display the orbit for that "number".

Fractal Voyager is implemented using a combination of *HTML canvas*, *Web Assembly*, and *React*. The HTML canvas is an HTML element that can display images and draw lines and boxes on the web. In Fractal Voyager, there are three distinct HTML canvases stacked on top of each other. One is for the fractals. This canvas takes an array of image data which consists of four elements for each pixel: three elements for the RGB color value, and one for transparency. The application generates this array based on the script and parameters, then passes it to a canvas which takes the data and draws the given fractal. The other two canvases are for drawing boxes for user box zooms and drawing orbits. These are separate as they are drawn on top of the fractal canvas.

Web Assembly, or wasm, is a low-level assembly language expressed in binary code which can be executed in a web browser. It has the ability to make computationally intensive tasks (such as generating fractals) much faster than the default language of the web, JavaScript. Using *Emscripten*, a compiler tool chain, we can compile C++ code to wasm, and then call the wasm compiled C++ code from JavaScript to run it on the web. We used wasm in this application for the cdScript language compilation process. This process uses ANTLR to define a grammar and generate parse trees based on scripts, which we traverse the to build up C++ code that represents the scripts. This

process is written in C++ and compiled to wasm for use in the web application. Another aspect of the application that makes use of wasm is an in-the-browser C++ to wasm compiler (called *Emception* [14]) which is a version of Emscripten compiled to wasm. Fractal Voyager uses Emception to compile the generated C++ to wasm, which is run to generate image data for the fractals.

Fractal Voyager is a React App, which allows for the integration of HTML canvas and wasm to be smoother and more developer-friendly than if the application was written in pure JavaScript. React is a JavaScript library for building user interfaces that makes use of a *component architecture* and *hooks*. Components are reusable chunks of code that render pieces of a web page; they include code to render HTML based on their properties, or *props* and both local and global *state*. A React App is made up of a component hierarchy, and uses hooks to handle aspects of the application that do not directly relate to rendered content, such as handling state and performing *side-effects*. Fractal Voyager was created with `create-react-app` which is a command-line tool to set up a React app with some default configurations. Furthermore, our app uses *Bootstrap React*, a library of React components, to style the application, and *Zustand*, a state-management library, to manage global state.

When a user enters a script in the web application, it first gets passed to the cdScript code generator. This uses the ANTLR visitor pattern to build up a C++ program that can take basic parameters and generate an image data array for a fractal defined by the script and the parameters. Once the React app receives the C++ code, it uses Emception to compile this C++ code to WASM, all in the browser. This WASM is dynamically imported to the React app, and then called to generate the initial fractal's image data. The C++ is heavily abstracted with many parameters, so once the initial program is compiled to WASM, the user can: edit parameters, switch to a dynamical plane from a parameter plane (and vise-versa), and generate orbits without need for a recompile. Fractal Voyager has been tested on a variety of complex systems to determine that it will be useful to the mathematical community. Furthermore, using WASM to generate the image data is a unique approach that allows for fast fractal generation on the web, along with run-time compilation, two features that make this type of application possible on the web.

To summarize, the contributions of this project are:

- The adaptation of an existing fractal-generating language to bring the language to the web, and integrating it with a fractal exploration application. The language supports both parameter and dynamical plane fractal generation, and knows the difference.
- The creation of a fractal explorer web application which allows for users to not only generate fractals based on scripts, but also edit parameters to slightly change the appearance of a fractal, zoom and pan the fractal, and generate dynamical plane fractals and orbits based on parameter/dynamical plane functions and user-inputted values.
- A proof-of-concept in using a wasm compiled compiler to compile C++ code to wasm and run it on the web

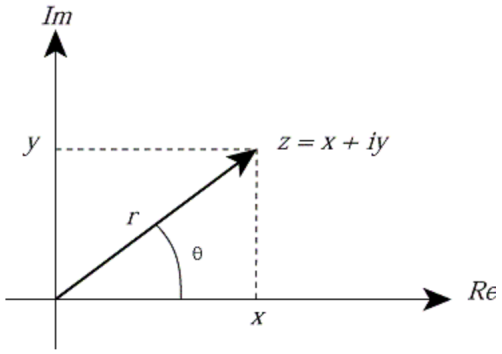


Figure 1: Plotting of z, a complex number, on the complex with z expressed as $x + iy$ and $r \cdot e^{\theta i}$. [6]

all in the browser, and all integrated with React hooks.

The remainder of this report is organized as follows. A background section (2) that goes into detail on all important technologies, mathematics, and computer science used to build, understand, and update Fractal Voyager. An implementation section (3) to describe all aspects of the implementation of the application. A worked example section (4) which describes how to use the application. Evaluation (5) to provide sample scripts that have been tested, conclusions (7) to summarize all sections, and an appendix which includes insightful code snippets.

2 Background

2.1 Complex Dynamics & Fractals

Complex numbers are at the core of the mathematics behind this application. These are numbers which include both a "real" and "imaginary" part. *Real numbers* are those which can be expressed along the real number line, such as $-1, 0, \frac{1}{3}, \pi$, and so on. *Imaginary numbers*, on the other hand, are those which can be written in the form, bi, where b is a real number, and i is the imaginary unit, or $\sqrt{-1}$. i is used as mathematics have no other way of expressing $\sqrt{-1}$ in terms of real numbers. That being said, complex numbers are a combination of these two types of numbers, so could be purely real or purely imaginary; so any real number is complex (imaginary part of 0), and any imaginary number is complex (real part of zero), along with any combination. These numbers are often expressed in the form $a + bi$, where a is the real part and b is the imaginary part. They can also be expressed in *exponential form* as $r \cdot e^{\theta i}$ where r is the magnitude of the number, θ is the angle, and e is the mathematical constant, e.

The complex plane is the set of all complex numbers, expressed on a plane. Any complex number can be plotted on this two-dimensional plane, and they are done so with the real part along the horizontal axis, and the imaginary part along the vertical axis. Any point on the plane corresponds to a complex number, which can be seen in figure 1.

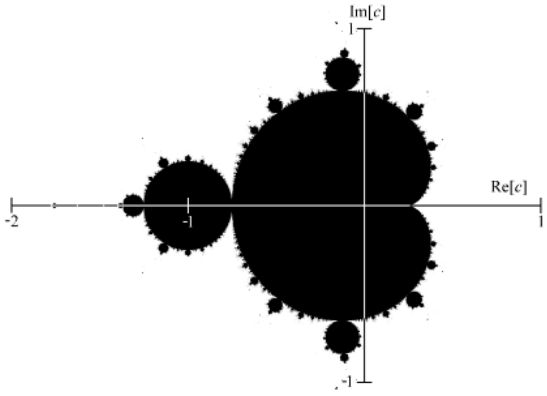


Figure 2: The Mandelbrot set with axes. [9]

Complex dynamics is the field of mathematics which study iterated functions in the complex plane. These are functions in which the inputs and outputs are complex numbers. This report does not intend to be a source of information on complex dynamics, but simply wishes to provide an extremely brief introduction in order to begin to understand the most basic level of the mathematics behind the fractals generated our application, and at the very least, understand how the application generates images. For our purposes, complex dynamics functions are separated into *parameter plane* functions and *dynamical plane* functions. Parameter plane functions iterate a variable that starts with a *critical point* (a value of z such that $f'(z) = 0$), and include a parameter which changes with respect the the complex plane. On the other hand, dynamical plane functions iterate a variable point which changes with respect to the complex plane. Thus, parameter plane functions include two parameters, while dynamical plane functions only include one. It can be seen how these two types of functions interact with each other; the parameter plane include all possible inputs for a set of dynamical plane functions. By setting a value for c, we can make any parameter plane function a dynamical plane function. In the following paragraphs, we will use examples to illustrate sets and fractals in the parameter and dynamical plane, how these fractals can be colored, along with a brief introduction into the importance of these fractals and the nature of their behavior.

The most famous parameter plane function is that of the *Mandelbrot set*. The set uses the function $z \mapsto z^2 + c$, with z always starting at 0. Any points of c across the complex plane that cause the iterative values of z to remain bounded are consider in the set. Figure 2 an illustration of the Mandelbrot set where points in the set are black, and points not in the set are white. Although it may not look like it, 2 is a fractal which exhibits self-similar properties. It can be zoomed in on at infinitely small scales along the edges of the fractal and will continue to have geometric patterns of points in the set.

In the Mandelbrot set, we set z to start at 0 because 0 is a *critical point* of the function $f(z) = z^2 + c$. A critical point is a complex number that is the value of z for which the derivative of $f(z)$ evaluates to 0. So, we have $f'(0) = 0$

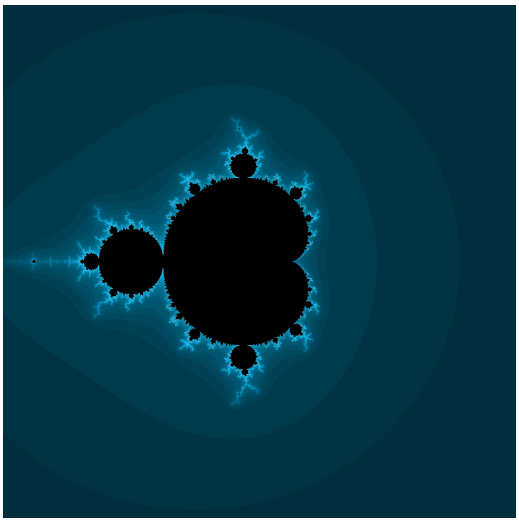


Figure 3: The Mandelbrot function, $z^2 + c$, as a fractal colored based on how quickly the critical orbit escapes past a magnitude of 4. It is plotted on -2 to 2 on both the real and imaginary axes, and reveals the chaotic behavior of the function along the boundaries of the Mandelbrot set. [5]

as the only critical point for the Mandelbrot set. In parameter plane functions, we always want to iterate a critical point. When we add a parameter, c, which changes across the complex plane, values of c which cause $z \mapsto z^2 + c$ to remain bounded are in the set. We define *critical orbit* as the values of z (the iterated variable) after each iteration of the function, when it starts at a critical point [8]. So, we could also say that if the critical orbit for a given c is bounded, then that value of c is in the set.

Taking this set, we can color it based on how quickly z (or the critical orbit) escapes or converges. It is easiest to understand Fractal Voyager's coloring method, along with most fractal coloring methods, as coloring points "in" the fractal along a gradient based on how quickly it meets the condition, and all other points black. The lighter the color, the longer it takes to meet the condition. If we color $z \mapsto z^2 + c$ based on how quickly the critical orbit of z converges to zero, we do not get very interesting results as most of the points inside the Mandelbrot set do not cause convergence to zero, rather they simply do not cause divergence. So, coloring the new set of points, *not in* the Mandelbrot set based on how quickly they "escape" will yield something similar to figure 3.

Turning our attention back to the mathematics of the Mandelbrot set, we can see how it acts as a map for all possible functions of the form $f(z) = z^2+c$ where c is a *fixed* complex number, and z is initially set to points across the complex plane. These functions can be considered the corresponding dynamical plane functions of the Mandelbrot set. These functions are often colored in the same way as the Mandelbrot set, based on how quickly z escapes. In figure 4, we color the function $z \mapsto z^2 + 0$, plotted on $-2 \leqslant Re \leqslant 2$, $-2 \leqslant Im \leqslant 2$, based on how quickly z escapes (left) and how quickly z converges to 0 (right).

Notice how points outside the until circle (circle centered at $0 + 0i$ with radius of 1) escape to infinity, and points

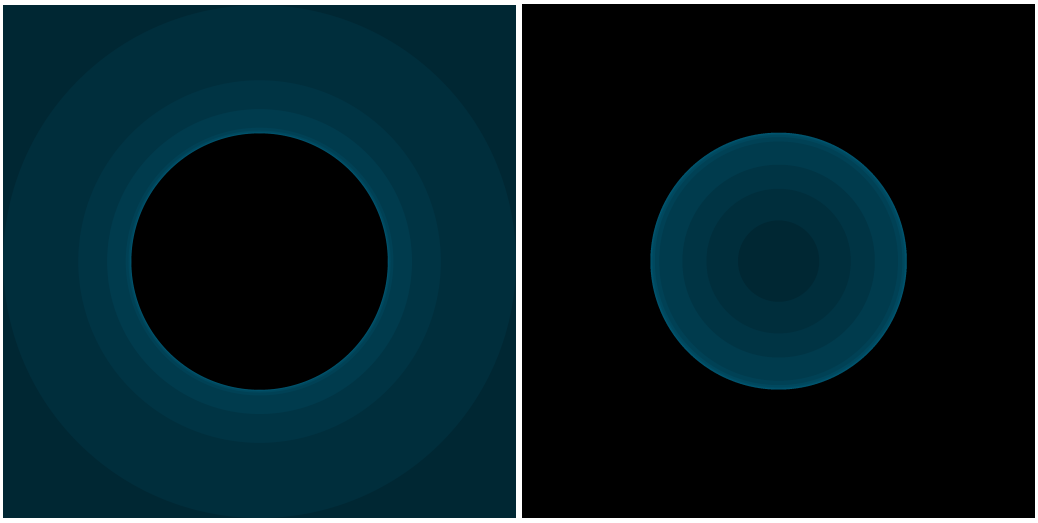


Figure 4: The dynamical plane fractal function, $z \mapsto z^2 + 0$, colored based on how quickly z escapes (left) and how quickly z converges (right). On $-2 \leqslant Re \leqslant 2$, $-2 \leqslant Im \leqslant 2$. [5]

inside the circle converge to 0. This gives insight to the nature of the function; points along the unit circle are not very "well-behaved" as they do not converge or diverge. They are infinitely close to points that converge to a *fixed point* (here, infinity is consider a point). This, is the definition of a *Julia set*: the set of points for a particular dynamical space fractal which do not converge to a fixed point, or exhibit "chaotic" behavior. If you go a little to either side of points on the Julia set, they will behave dramatically different under iteration. A related set, the *Fatou set* is the *compliment* of the Julia set, thus can be defined as a set of points for a particular dynamical plane function where all neighboring points behave similarly under iteration [8]. For $f(z) = z^2$ the entire complex plane is the Fatou set except for the unit circle. As stated earlier, the Mandelbrot set includes all possible inputs for a set of dynamical plane function by fixing a value of c. Now that we understand Julia sets, some mathematical properties of Julia sets fall out of the Mandelbrot set. For example, any values of c which are not in the Mandelbrot set yield disconnected Julia sets, while values of c which are in the Mandelbrot set yield connected Julia sets.

Using simple complex analysis, we can show how $z \mapsto z^2$ behaves and is constructed based on it's *orbits*. A complex number's, say w's, orbit for a particular dynamical plane function, say g, is the sequence of values after each iteration of the function when the number (w) is the starting point, so in this case, the values of $w \mapsto g(w)$. Say we are coloring our dynamical plane fractal, $z \mapsto z^2$, based on how quickly z escapes. This can also be thought of as the orbits of values of z across the complex plane's behavior as follows: if an orbit of z escapes to infinity, it is "in" the colored set. If an orbit of z *does not* escapes to infinity, it is colored black. Earlier, we noticed that values outside the unit circle escape to infinity. Let's show this by considering orbits of values with magnitude greater than 1.

Recall that complex number's can be represented as $re^{\theta i}$ where r is the magnitude of the number and θ is the angle. If we take a number outside the unit circle, it will have a $|z| > 1$, or a magnitude greater than one. Let's

consider $z = re^{\theta i}$ for any θ and an $r > 1$. Iterating this z in our function $z \mapsto z^2$, will first yield $(re^{\theta i})^2 = r^2 e^{2\theta i}$. It can be seen that as this continues, so long as r starts above 1, the magnitude will escape to infinity. θ will get multiplied by 2 each time, so the orbit will "circle" the origin as it escapes it infinity. If r starts < 1, we will have a number less than 1 being squared, which will cause it to converge onto 0. The above demonstrates how orbits relate to the coloring of a dynamical plane fractal, along with how the two interact with Julia and Fatou sets.

The behavior of $z \mapsto z^2$ is somewhat uncommon in the complex plane, as it is one of the few functions which create a simple geometric shape where the values inside the shape have orbits that go towards 0, and the values outside have orbits that tend towards infinity. Often, when examining dynamical plane fractals based on the Mandelbrot set, the values of z which do not tend towards infinity, end up on a *cycle* instead of tending towards 0. A cycle can be observed in the point's orbit, as the orbit will jump between n numbers. For example, a *2-cycle* could be the orbit of $z_0, z_1, z_3, -1, 0, -1, 0, -1, \ldots$. Notice how the orbit does not start on the cycle, but eventually reaches two numbers that it jumps between. For coloring dynamical space fractals based on z escaping, the points which end on a cycle are simply colored black as they do not escape. Given this, points inside Julia sets exhibit truly chaotic behavior, while points inside the Fatou set will land on cycles (could be a 1-cycle by stopping to iterate), or converge to a fixed point.

We can examine dynamical plane fractals based on the Mandelbrot set to observe stunning Julia set fractals, which are much more interesting than the until circle. The set of chaotic points that make up the Julia set are colored extremely lightly since they take a very long time to reach the *approximated* condition. Truly, a Julia set is the points on the boundary of the points colored black and those which are colored "in" the fractal set as z escapes, but these types of dynamical plane fractals are often simply referred to a Julia sets. Figure 5 include four dynamical plane fractals based on the Mandelbrot set by fixing values of c.

One can observe cyclic behavior of points based on plotted orbits. For example, consider figure 6 which includes the orbit for a dynamical plane fractal based on the Mandelbrot set with a c fixed at -.5+.56i. The orbit is for .5-.54i, and all points inside the black regions (those that do not escape) get stuck on the same 5-cycle.

Examining dynamical plane fractals that are based on parameter plane fractals such as the Mandelbrot set is interesting, but not the extent of dynamical plane functions in complex analysis. Another application of the dynamical plane is constructing Julia and Fatou sets for Newton's Method fractal. Newton's method is a protocol for approximating roots of functions by taking a starting number, say $z = 0$ and iterating $z \mapsto z - \frac{f(z)}{f'(z)}$ where f is the function to approximate roots for. Any given z will tend towards roots of the function. This method can be turned into a fractal in the dynamical plane by iterating z values across the complex plane in a given Newton's method function, and points will have orbits that tend towards a root of the original function. Consider the function $z^3 - 1$. Its derivative is $3z^2$, so iterating $z \mapsto z - \frac{z^3-1}{3z^2}$ will eventually "stop" at a root of $z^3 - 1$ for any value of z if Newton's method holds. These roots are $z = 1$, $z = -\sqrt[3]{-1}$, and $z = (-1)^{\frac{2}{3}}$. A particular orbit of z will get closer and closer to one of those roots until the difference between the value of z at iteration n is infinitely close its value at iteration

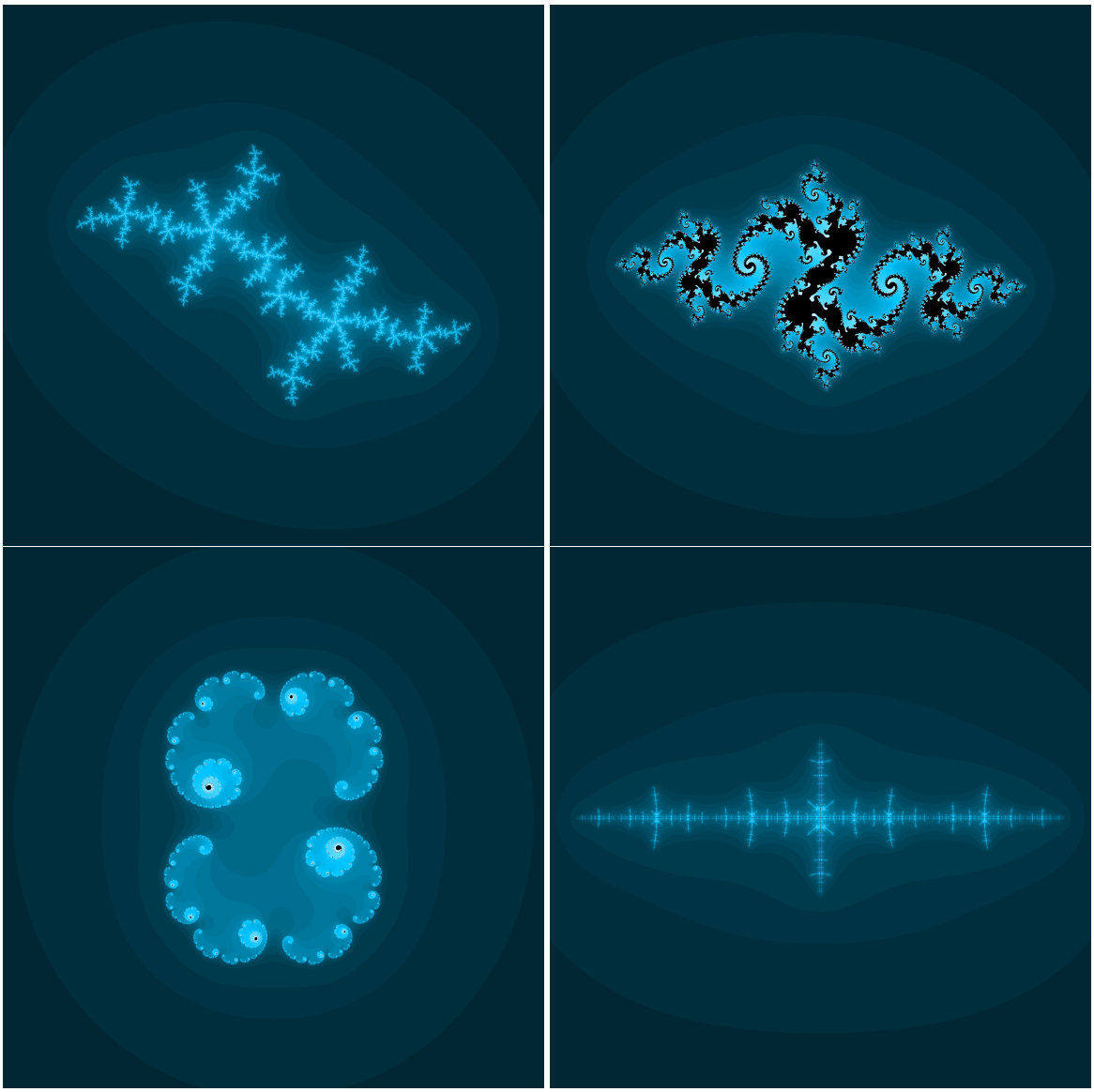


Figure 5: Julia set (dynamical plane) fractals based on the Mandelbrot set with c values of (-0.55+0.63i), (-0.79+0.15i), (0.3-0.01i), and (-1.467), respectively, from left to right, top to bottom. They are all plotted on $-2 \leqslant Re \leqslant 2$, $-2 \leqslant Im \leqslant 2$. [5]

Figure 6: Dynamical plane fractal for $z \mapsto z^2 - .5 + .56i$ with the orbit of $.5 - .54i$ is red on top of the fractal. [5]

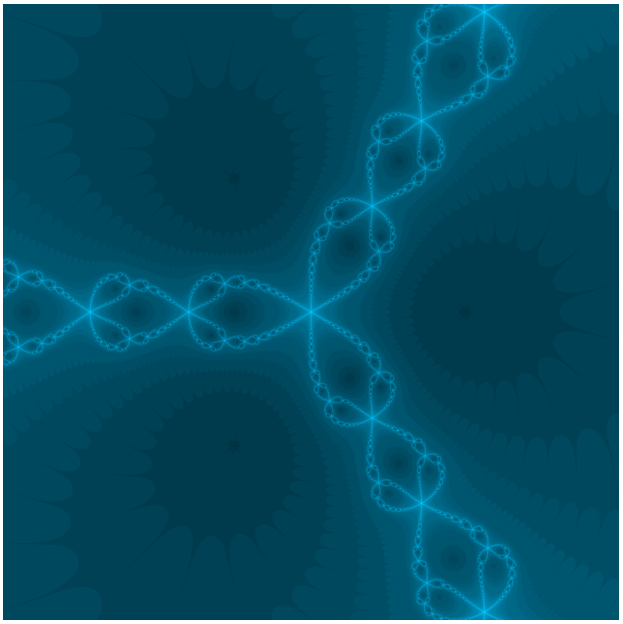


Figure 7: Newton's fractal for $z \mapsto z - \frac{z^3-1}{3z^2}$ colored based on how quickly z stops. [5]

$n-1$. We can construct a fractal in this way, iterating $z \mapsto z - \frac{z^3-1}{3z^2}$ until z "stops", as seen in figure 7.

We can also find the orbits of values to see which root they tend towards, or coloring's of this fractal exists that color regions of the plane based on what roots they go towards. These two graphics can be seen in figure 8. It can be observed that the Fatou set for the function is the union of the shaded red, blue, and green areas, as points in these areas exists in neighborhoods of points which tend towards the same root. The Julia set for $z \mapsto z - \frac{z^3-1}{3z^2}$ is the boundary of the colored areas, as these are points which act chaotically, going a little to either side changes which root the point is attracted to.

2.2 Programmatic Fractal Generation

Now that we have developed some understanding the mathematics behind coloring fractals, let's consider programmatic fractal generation. Let's first construct a program which can determine if a given complex number is in a fractal set. Since both parameter and dynamical plane fractals are defined with iterative functions which contain one point that varies based on the current position in the complex plane, there is not a big difference between generating them. The difference in generation is the same as the difference between the types of functions themselves: one iterates a critical point and includes are variable point (parameter), and the other iterates a variable point (dynamical). For the purpose of this example, we will assume that 0 is the "guess" for the critical point in a parameter plane fractal. That is, since parameter plane fractals must iterate a critical point, we will always assume that 0 is a critical point. The following snippet of C++ code could be used to determine if a point is "in" in the Mandelbrot set fractal, as z escapes.

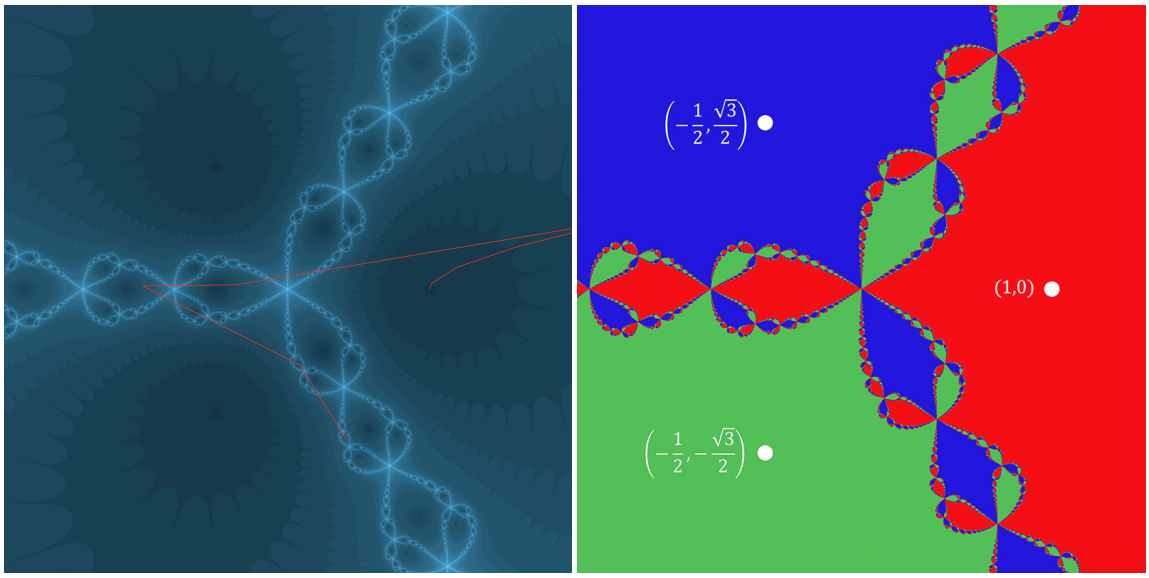


Figure 8: Newton's method fractal for $z \mapsto z - \frac{z^3-1}{3z^2}$. Left is colored based how quickly z stops, and shows an orbit tending towards a root of $z^3 - 1$ [5]. Right is colored based on what root the orbit for any z tends towards [4].

```
1  int mandelbrot (double c_re, double c_im) {
2      std::complex<double> z(0, 0);
3      std::complex<double> c(c_re, c_im);
4      for(int i = 1; i < 64; i++) {
5          z = z*z+c;
6          if(abs(z) > 2) {
7              return i;
8          }
9      }
10     return 0;
11 }
```

A complex number c, which is $c_{re} + c_{im}i$, is passed into the function. The mathematical function, $z \mapsto z^2 + c$, is iterated a fixed number of times, in this case 63, which is an estimate of infinity. We are saying that if z does not escape within 63 iterations, it never will and thus is not in the fractal set. This iteration is achieved with a for loop that runs 63 times, and inside the for loop, z is "passed in" to the mathematical function ($z \mapsto z^2 + c$). It is squared, "z*z", then added to c, "+c". If the resulting value's magnitude is greater than 2, we say that the value has escapes to infinity. This is once again an estimate; we are assuming once the magnitude of z goes past 2, it will go to infinity. We can pass in a variety of values for c to determine if they are in the Mandelbrot set fractal or not.

If they are "in the fractal", the function they produced has a critical orbit that escapes, so a number other than 0 will be returned because the condition eventually will be true. If they are not, a value of z never had a magnitude larger than 2 (meaning the orbit never escapes), so 0 will be returned since the for loop will be run entirely through without returning. It should be noted that we are using the C++ complex number library (complex.h) which allows us to define complex numbers, do math on them, and provides helpful function such as `abs()` which computes the magnitude of a complex number. It should also be noted that values "in the fractal" are the values of c that cause z to escape, since that is the condition we are using. The values that are not in the fractal are actually the ones "in" the Mandelbrot *set*, since they do not cause the critical orbit to escape.

Now, let's adapt the function to calculate the dynamical plane fractal of the function $z \mapsto z^2 + .5 + .5i$ until z escapes.

```
1  int julia (double z_re, double z_im) {
2      std::complex<double> z(z_re, z_im);
3      for(int i = 1; i < 64; i++) {
4          z = z*z+std::complex<double>(0.5, 0.5);
5          if(abs(z) > 2) {
6              return i;
7          }
8      }
9      return 0;
10 }
```

This function takes a complex number and sets it to z, the variable that is being iterated in the function, and iterates the given function until z "escapes". It is very similar to the function for the Mandelbrot set fractal. Thus, it is desirable and possible to create an abstracted function which can calculate *either* the Mandelbrot fractal or a Julia set fractal based on the Mandelbrot set. Let's consider $.5 + .5i$ as the "c" value, and pull that out of the mathematical function, into a variable like the following: `std::complex<double> c(.5,.5)`, and iterate the function `z = z*z+c`. Now, this is nearly identical to the Mandelbrot function, but in the Mandelbrot function above we have z set locally instead of a passed in parameter, lets change that to yield:

```
1  int calcPoint (double z_re, double z_re double c_re, double c_im) {
2      std::complex<double> z(z_re, z_im);
3      std::complex<double> c(c_re, c_im);
4      for(int i = 1; i < 64; i++) {
5          z = z*z+c;
```

```
6           if(abs(z) > 2) {
7               return i;
8           }
9       }
10      return 0;
11  }
```

In which, we can call `calcPoint(0, 0, c_re, c_im)`to determine whether the given value of c is in the Mandelbrot fractal or not, or `calcPoint(z_re, z_im, 0.5, 0.5)` to determine if the given value of z is in the dynamical space fractal defined by $z \mapsto z^2 + .5 + .5i$ until z escapes. Given this function, we can create a dynamical plane fractal based on any value of c in the Mandelbrot set function.

We have established that we can determine whether a complex number is in a fractal set with a simple program, and now we may consider how this relates to the coloring of that point. If the point is not in the fractal set, 0 is returned, so we can simply color all points which return 0 black. If a point is the set, the number of iterations it took for the point to reach the condition is returned, with the maximum number of iterations being 63. Thus, we can color a point along a colored gradient based on, relatively, how fast the point escaped. Say we are coloring a fractal with points that take longer to escape on the lighter end of a gradient. We can take the fraction of how long a number took, say $\frac{55}{63}$ if the number took 55 iterations to escape, and find a color that is $\frac{55}{63}$ of the way down the gradient if 1 is the lightest, and 0 is the darkest.

The same logical approach that was taken to create a function to color the Mandelbrot fractal and associated dynamical plane fractals can be used to color any fractal. The approach is as follows:

- Pass in a z and c value and initialize
  - For a parameter plane fractal: z will be always be 0 and c will vary across the complex plane to determine if c points are in the fractal.
  - For a dynamical plane fractal: z will vary across the complex plane, and c will only be passed in if it is a dynamical plane fractal based on a parameter plane one.
- Use a loop that iterates a given number of times
- Set the iterated variable, z, to the result of the iterated function. For the Mandelbrot set fractal, this is $z^2 + c$, but could include other constants and more complicated mathematical functions, such as sin.
- Check if z has reached a certain condition
  - If it has: return the number of iterations

- If it hasn't: continue the for loop

Above is the logic for taking a complex number and determining how to color it. Now, we must consider how to color an entire surface that represents the complex plane. Bounds on the complex plane must be defined, let's say $-2 \leqslant Re \leqslant 2, -2 \leqslant Im \leqslant 2$. Next, we must define how many *pixels* to have in this space. The space will be filled with pixels, and each pixel will be equated to a complex number. The more pixels we have, the higher the resolution and the more accurately the surface is projected to the complex plane. For example, if we have 500 pixels, we are saying that the complex plane from $-2 \leqslant Re \leqslant 2, -2 \leqslant Im \leqslant 2$ has 500 points, when mathematically, it has an infinite number of points. As we increase the resolution, the closer to infinity we get, and the more complex numbers we color. For the sake of easy math, let's say there are 800 pixels, and thus, 400 pixels along each axis. Since each axis has a "width" of 4 $(2 - (-2))$, points along an axis will be 0.01 away from each other $(4/400)$. So, with this plane projection, we will have $0 + 0i$, $0.01 + 0i$, $0 + 0.01i$, $0.01 + 0.01i$, and so on, for all points between $-2 \leqslant Re \leqslant 2, -2 \leqslant Im \leqslant 2$.

To create a fractal, first, the complex number representation of each pixel in our plane will get passed into our function to determine the number of iterations that number takes to meet the condition. Then, the given pixel will be colored that way to create a pixel in the fractal. This method works the same for the parameter plane and dynamical plane. Also, we can calculate the orbits of points by passing that point through the fractal's function and storing the value after each iteration. In the process of programmatic fractal generation, there are many places where estimations are made; tweaking these assumptions can change the fractal, mainly by increasing or decreasing the accuracy or quality. Increasing the number of pixels, or the concentration of pixels (we had 50 pixels for 1 square unit - $800/(4*2)$), will increase the quality. Increasing the number of iterations will create a more accurate fractal, as we may "miss" some values that would have escaped after 64 iterations. The same is true for increasing the number for when we call a value infinity. To generate some other fractals, we may need other types of conditions that also need a "bailout" value. This could be when to call a value 0, or the minimum difference between two numbers to call them the same value.

2.3 Language Implementation

Fractal Voyager uses a scripting programming language, cdScript, which we designed specifically for use in the application. Therefore, some background on the implementation of a programming language is warranted for the scope of this report. Before we begin to explain the steps taken to implement a programming language, let's first define exactly what a programming language is. It is a set of rules and the syntax associated with those rules that are implemented by a *compiler* or an *interpreter*. A programming language allows people to *program*, which can be thought of as "the art of telling another human being what one wants a computer to do" [1]. This is because

programming languages are written by humans, and these languages attempt to take something that a human wants to do, and convert it into something a computer can understand and run. Programming languages can be divided into *paradigms* which define the general style of a language. A language's paradigm can be thought of as the way a programmer goes about writing code. cdScript is a *declarative* programming language, which is a paradigm in which programmers declare a desired output, without specifying how to achieve it.

When building a programming language, one key decision is if the language will be compiled or interpreted. Compiled languages take the source code program (text which represents a program in the language) and transform, or *compile* it into a different, often lower-level, programming language. By lower-level language, we mean one which has less abstraction. In practice, this often involves taking a program and converting into into an *assembly* language. Assembly languages are very low-level and provide instructions for the computer's operating system in a human-readable and -writable way (for the trained programmer). Assembly programs are representations of *machine code* which is binary code that computers understand and can directly execute. On the other hand, interpreted languages use an *interpreter* which run the program through a *virtual machine* that reads and executes statements of the program to produce output. Compiled languages are often faster since the initial program gets converted into a programming language which exists at a low-level of abstraction, and often, interacts with the computers operating system directly. These lower-level languages are much more efficient that higher-level languages, and thus, much faster. However, compilers are often harder to build than interpreters, and require a more complex build process to run programs in the initial programming language. For either type of language, the initial steps that a source program goes through before it is executed are the same. Furthermore, these steps mimic that of the development process of implementing a programming language.

The first step in the process begins with source code; this is simply a text file often stored as ASCII or Unicode that is a "program" in our language. This text must be broken up into valid "chunks" or *tokens* in a process called *lexical analysis*. A token is a syntactic category that correspond to a set of "valid" strings, and are associated with a *lexeme*, which is a base until in the language and one of those valid strings [1]. For example, "7" is the lexeme for a particular token that could be an int literal for a language. Some other examples of tokens are identifiers, keywords, punctuation, etc. Often, in lexical analysis, a list of token-lexeme pairs are returned. The first step in building a program for lexical analysis, or a *lexer*, is to determine what token types are valid in the language. Then, programmers often use *regular expressions*, or regex, to define different token types. A regex uses a combination of ordinary characters and special symbols to "match" a particular string. Thus, we write regular expressions for each token type, that will match all strings which are of that token type. Languages whose tokens can be expressed purely in regex are consider *regular languages*. Finally, an input (source-code text) must be processed by a program which breaks up the input into tokens. In this process, the *maximal match rule* is used, which says that the analysis program will pick the longest possible sub string that matches to one of the tokens. This process can either be written from

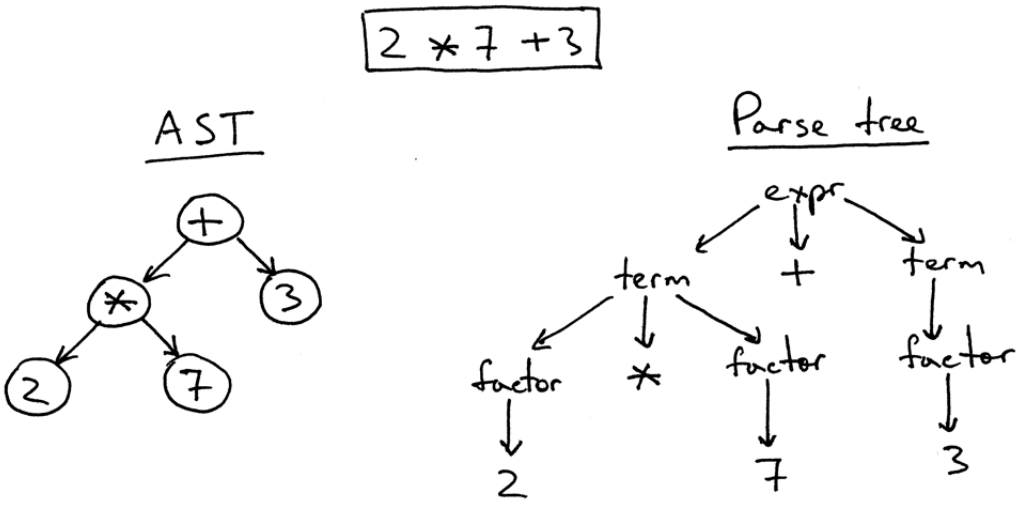


Figure 9: Two representations of the program "2 * 7 + 3": an AST and a parse tree. The differences between the two can be seen [3].

scratch, or can use a *lexical analysis generator* or other tools to aid in the process. These tools often take the regular expressions that define tokens, and will break up the source into those tokens for you.

The next step is parsing. A *parser* is a program which takes a takes a list of token-lexeme pairs produced by a lexer, and returns an *abstract syntax tree* (or AST). An AST is a tree data structure that represents the syntactical structure of a program. An AST is similar to a *parse tree* which is more detailed and contains all of the original information from the source code. For example, a parse tree may contain parenthesis, but this is not needed in an AST since the structure of the parenthesis can be shown within the tree structure. 9 is a representation of the difference between an AST and a parse tree; here, the parse tree also shows the components which make up the program that are built up from tokens.

Parse tree's could be produced by a parser as an intermediate step to producing the AST (or even the final), but this report does not intend to go into detail of the of how lexers or parses actually *work*, rather just provide a brief overview of the steps involved in building a programming language.

People building languages need a way of defining what valid uses of the tokens are, that will be used by the parser to create the AST. This can be though of as a *grammar*. There are many types of grammars and ways to define languages, but let's just consider an example of defining a simple expression in a language. Let's says the expression could either be simply an integer, two expressions added, or two subtracted. Given this grammar, it can be seen that the core "element" of all expressions is an integer; that is, all expressions are made up of them. These are considered *terminal nodes* in parsing. That is, the parse tree will end on these nodes. Other terminal nodes are the symbols

used as indicators, such as "+". *Non-terminal nodes* are nodes that will not be leaves of the parse tree. In our small grammar example, an expression is a non-terminal node. Given these definitions and our simple example above, let's create a more formal representation of what an expression could be.

```
1    E -> integer
2    E -> E + E
3    E -> E - E
```

The above shows how we can break down an expression into a simple grammar made up of *rules*. The terminal nodes are tokens which will be matched with regex (ex. "integer", "-"). The non-terminal nodes are expressions which will continue to be matched. We must be careful when defining these grammars to avoid *ambiguity*. A particular expression would be ambiguous if it has more than one way of being parsed that is valid. The job of a parser is to use a grammar definition and a list of token-lexeme pairs to create the AST representation of the program. Depending on how the parser works, there are certain considerations we must have when creating a grammar. Here, parsers also use the maximal match rule. For example, a parser given `2 - 3 + 2`, will take this as `E + E` (if it considers order of operations), then, break up one of those Es into `E - E`, then break them into integers. It can be seen how this creates a tree-like data structure.

The next step in taking source-code to execution is *semantic analysis*, or *type checking*. This is the process of making sure our AST "makes sense" and often returns and *annotated* AST. In programming languages, a *type* defines a set of valid values and operations over values. For example, we may have an integer type which can be added or subtracted. The type checking process varies greatly based on what type of language we are working with. For example, assembly languages do not involve any type checking, as they don't really have types, only bytes and bits of data. *Statically typed* languages (such as Java and C) type-check before execution, or in this phase of type-checking. This is because types must be defined in the language, for example: `int x = 4`. *Dynamically typed* languages (ex. Python and JavaScript) perform most of the type checking as part of the execution because users do not define types in the program. However, dynamically typed languages certainly still have types, as it doesn't make sense to multiply "4" and "cat". Typically, compiled languages are statically-typed, while interpreted languages are dynamically typed.

After type checking, if there was any, the process changes drastically for compiled languages and interpreted languages. At this point, we have a tree data structure that represents the program. This may be passed to an optimizer to make this tree the most efficient representation of the program. After optimization, an interpreter will be run in a virtual machine to process the program and produce output. In this process, we will actually evaluate expressions and the program in general; *do* `foo = 5 + 5` and store the value of "foo". This virtual machine will evaluate the program and "do what it asks". These virtual machines are normally packages that are downloaded and run on a local machine. On the other hand, for a compiler, we must perform *code generation* to turn our program

into a valid program of a different language. This is often assembly, but could be a *source-to-source transformation* which takes one programming language and turns it into another programming language that is not assembly. Then, the code in the new programming language may be compiled again to run on a computer. Whether a language is interpreted or compiled, code-generation (or cgen) and interpretation is done by *traversing* the AST and often uses *operational semantics* which are the rules of exactly how expressions are evaluated in the language.

For an interpreter, after the program is interpreted the process is finished as this executes the code. For a compiler, after the cgen to an assembly language we still are not done, as the code must be executed by a computer. The process of executing compiled code varies greatly for each language (like most of these steps) but usually, a compiler eventually produces a ".exe" file, which can be understood directly by a computer's operating system. This file is run by the user's operating system, which produces the desired output.

The steps described above for taking source code to execution merely scratches the surface of this process. We did not go into detail on *how* any of these processes work, and even skipped over many of tasks these steps must do. For example, most programming languages have an idea of *scope*, some have *classes*, and almost all cannot use variables before initialization. These were not mentioned as this section wishes to provide an overview of the steps so the implementation of our language makes sense.

The language for this application was implemented with a tool called *ANTLR*, or (ANother Tool for Language Recognition). ANTLR is a widely used tool for creating programming languages, and simplifies the process described above by doing some of it for us. At its core, ANTLR is a tool that can take a grammar which includes both lexer and parser rules and convert it into a parse tree data structure [12]. Once this data structure is created, some may choose to traverse it themselves, or implement ANTLR's built in protocols for traversing the tree: either the visitor pattern or the listener pattern. Before diving into the specifics of these two methods and how we can go from a tree to a program in ANTLR, let's figure out what ANTLR really is.

ANTLR not only generates a parse tree, but generates a lexer and parse which creates this tree. Thus, we can edit the lexer and parser to fit our needs, even after we have defined the grammar. Before ANTLR can do anything, we must define a grammar with ANTLR's format. This format allows us to define both a lexer and parse grammar in one file. It is made up of rules which dictate what are valid programs in our language. The lexer rules, which always start with lower case, define what our tokens are using a combination of regex and custom ANTLR syntax. This allows ANTLR to generate a lexer for our tokens. The parser rules, staring with upper case, are the grammars rules defined in our discussion of parsing. These allow us to define what structures are valid in our language. The basic structure of parser rules is `RuleName : alternative1 | ... | alternativeN ;`. Following our above example when discussing grammar, our expression grammar could be translated into an ANTLR grammar as follows:

```
1       E :
```

```
2       integer |
3       E + E |
4       E - E
```

When "integer" is a lexer rule defined with the regex for an integer.

Once we define a grammar, we can begin to use ANTLR. ANTLR is "a tool written in Java that translates your grammar to a parser/lexer in Java (or other target language) and the runtime library needed by the generated parsers/lexers" [12]. So, in short, ANTLR uses the defined grammar to translate it into a parser and lexer written in Java (or a lnaguage of our choice, we used C++). Then, we can edit these tools, and in combination with the runtime library, we can run source code through our lexer and parser, to generate parse trees, and can even traverse these trees with built-in ANTLR methods. As mentioned earlier, the ANTLR built-in methods are broken into two protocols for traversing the parse tree: the visitor method and the listener method. These both perform a *depth-first traversal* of the parse tree. This is the same way almost all interpreters and code-generators traverse the AST or parse tree.

In a depth-first tree traversal, the "deepest" node on the furthers left side is visited first, and once a leaf node is hit, the algorithm backtracks to the parent of the leaf node, and visits the next child of that parent. Figure 10 is a visualization of a depth-first traversal, where the red numbers correspond to the order the nodes are visited in, and the orange line is how the nodes are visited.

Traversing the parse tree in this way is best for code generation and interpretation as it visits all nodes exactly once, is efficient, and "makes sense" for language generation. It makes sense because if we think of a program, we have little bits, like integers, that get combined in all sorts of ways to make up the program. If we visit the integers and terminal nodes first, we can evaluate those small pieces first, and slowly work our way up to pieces of the code that are more complicated and involve more pieces. There are many known and easy-to-implement algorithms to perform a depth-first traversal, but ANTLR's built-in methods are preferred as we not only do not have to implement this algorithm ourselves, but they provide additional boilerplate code to handle passing data between the nodes, have many easy to use methods that save us time, among other reasons.

We will provide a brief introduction into the visitor pattern, since that is what our application uses. To create visitor classes in our ANTLR project, we simply create the lexer and parser with an additional flag: `antlr4 grammar.g4 -visitor`. This will generate a visitor parser and lexer in Java, and we can change the language by using additional flags and a different target. It will also generate a method and class object for each parser and lexer rule we have, in which the default behavior is simply to visit all children. Now, we can override this default behavior to change how our parse and lexer works. For example, if for some reason we want to add 1 to a global counter variable every time we reach a specific parser rule node, we can override the visitor method for that node, and before visiting children, add one to a counter variable. The basic idea of building an interpreter with this visitor pattern is

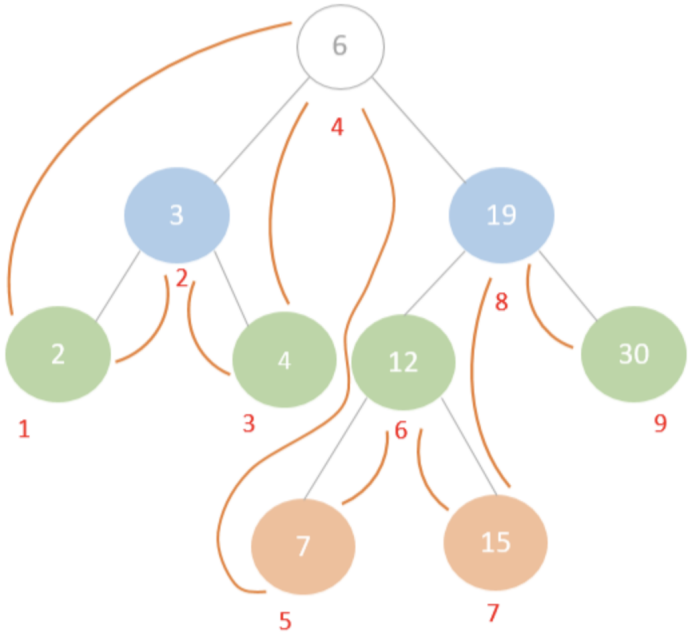


Figure 10: Visualization of a depth-first tree traversal where the numbered circles are nodes, the red numbers correspond to the order the nodes are visited and the orange line represents how the nodes are visited [2].

to have each node method return a result, then as we traverse the tree, this result will get added up until the final result is the evaluation of the program. To build a code-generator, we could have a global scope variable that is a string representation of the code in the new language, and as we traverse the tree, build up this string.

Our discussion of ANTLR only provides an extremely brief introduction into how we used it. ANTLR has a website, documentation, and even a book which can be found by visiting their website, `antlr.org`. We also barely touched on how language processing works, this section merely exists to provide some background into it so our implementation can be best understood.

2.4 Web Development

When building an application, one major choice is whether to develop a *native application*, or a *web application*. Native apps are built for a specific operating system, are installed directly on a user's device, and run on a user's operating system. This allows them to take full advantage of a local machine, but needs to be developed for each operating system (ex. macOS, windows, etc...). Web applications, on the other hand, run on the web. The web is a vast system of content which is accessed by the internet through *web browsers*. A browser takes user requested web sites and retrieves and displays (or *renders*) the page on a user's screen. Web applications could be preferred over native as they are extremely versatile because they can be written once and work on all devices that have access to

the internet. Web pages are primarily written in three programming languages: JavaScript, HTML, and CSS.

HTML is a *markup language* that defines the structure of a web page. HTML pages are made up of HTML *elements*. Some examples of an HTML element are `<div>`, `<h1>`, `<table>`. Different HTML elements are meant for different purposes; for example, `<h1>` typically contains header text which will be bigger than text defined in a `<span>` which is used for inline text. `<div>`s are often the building blocks of websites as they can contain other elements and are simply container dividers that can be *styled* with CSS. CSS, or cascading style sheets, is the language which defines how HTML elements should appear on the web page. CSS code often, but doesn't have to, exists in a separate document than the HTML for a web page. We could have an HTML element, ex. `<div id="foo">`, then *select* it with a CSS selector for an ID attribute with `#foo`, and apply styles to it. These styles could include something like `background-color: black` which will make that div black. Styling with CSS involves much more than coloring, it is the main way which we define the layout of a web page.

One import HTML element is the `<canvas>` element. This element allows us to have an artistic canvas of sorts on a web page. Lines, rectangles, circles, and text can all be drawn onto the canvas, and it can even display images. This drawing is done through JavaScript by passing a canvas element either specifications of how to draw a rectangle or other shape, or an array of image data so the canvas can render the image. The image data for canvases is an array that holds four elements for each pixel, three for the RGB color value of the pixel, and one for the pixel's transparency. We can define how many pixels a canvas has by creating it with, or editing it to include, width and height properties. A canvas with width of 200 and height of 200 will be 200 by 200 pixels. The axes of HTML canvas start with (0,0) in the upper left side of the canvas, and go down from there. So, in this case, the middle would be (100, 100), meaning 100 pixels across and 100 pixels down.

The HTML elements of a web page have opening and closing *tags* with content in between. The entire HTML page can be represented as a tree data structure, and is. This is called the *Document Object Model*, or DOM. The DOM can be programmatically accessed with JavaScript, and represents the structure of any given web page. Figure 11 is an example of a simple DOM for a web page that would merely display "My header" is a larger font, then "My paragraph". That is, without any JavaScript manipulating this DOM.

The other extremely important programming language, often consider the language *of* the web, is JavaScript. JavaScript, or simply JS, is a multi-paradigm, dynamically typed, interpreted programming language that is used alongside HTML and CSS to build websites. JS handles all of the *client-side* logic on a web page, which is the content that a user can see on the site. JS can manipulate the DOM, and has API methods for fetching data from servers or other web pages. JS is used to making *dynamic* web pages; if a website simply contains content to read, it could be written purely in HTML and CSS, but, if there is any amount of user interaction, it must include JavaScript code. For example, we could have a website where clicking a button adds to a counter that is displayed on the page. For this functionality, we would use JavaScript to handle a click of the button, then access the HTML element that holds the

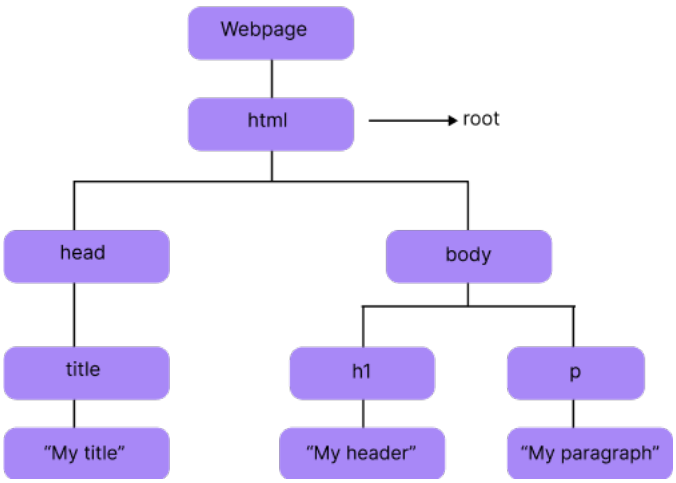


Figure 11: A simplified example of a possible DOM for a web page that only has two elements: a header and a paragraph [7].

number of clicks, add to it, then the web page would update. JS works on the web because all browsers contain a JS engine, which works as a virtual machine described in subsection 2.3. It interprets the JS code and uses its results to render a web application with functionality.

When web applications grow in size and scope, if all of the code is written only in JS, HTML, and CSS, the JavaScript code becomes hard to manage and write. The application will have a lot going on, and it can be hard to manage the intricate components and effects that should happen in pure JavaScript. Luckily, there are many JS *frameworks* which provide developers with pre-written pieces of JavaScript code, along with a general structure/rule set one should follow when building an app with that framework. For FractalVoyager, we choose to use the React framework for JavaScript.

The React framework allows user interface applications to be easier to develop than plain JS by using JSX, components, props, state (with hooks), and a virtual DOM. JSX is an extension on HTML that allows developers to add JavaScript code into HTML, and it is the way we write HTML is React. Another key feature of React is the virtual DOM it implements. When there is a change in data, or *state*, React generates a new virtual DOM and compares it to the old saved virtual DOM and updates the "real" browser DOM as necessary. React applications are built from *components* which are reusable chunks of code that render pieces of a web page. They are created with proprieties, or "props", and include JS code that use props, state, and other variables to determine what should be rendered. Of course, they also include the JSX that gets rendered by the browser. React also uses hooks which are special functions only valid in React which handle aspects of the application that do not directly relate to rendered content, such as handling state and performing side effects. In React, we can store data that will change in state variables, and components can react to these state changes to re-render.

In React apps, we often have state that needs to be shared or sent between components. This can easily be passed "downstream" by passing state as props to children components, but it is a little harder to pass state "up" or "across" the *component hierarchy*. The way React deals with *global state* is hard to work with and is not as powerful as we often need. For our application, we used a state management library, Zustand [13]. Zustand allows us to easily share state between components, and allows for a lot of control over when to trigger re-renders based on global state. For styling our application, we used React Bootstrap, which is a style framework [15]. It includes many pre-built React components for simple elements such as buttons, that are already styled nicely. It also provides an easy way to break up the web page and style it appropriately.

As stated earlier, JavaScript is the main language of the web, but it interpreted, so can be quite slow for computationally intensive tasks. To solve this, in 2017 Web Assembly, or wasm, was released. Wasm is a binary code format that can be executed by a web browser, and allows near native speeds on the web. We can take a program in almost any language, compile to wasm, and then run it in a web browser. FractalVoyager uses wasm in many places, and primarily compiles C++ to wasm with *Emscripten*. Emscripten is a compiler toolchain that uses LLVM/Clang to compile C++ and C to wasm to be run in the browser. In addition to generate pure wasm, Emscripten generates a JavaScript module for us to interact with the wasm code. For example, when we are compiling to wasm on the command line with Emscripten, we pass flags to indicate certain ways for Emscripten to generate wasm, including methods to include. These runtime method flags include some that Emscripten provides to work with wasm such as *malloc* and *free* to allocate or un-allocate data on the heap, as well as the C++ functions we defined to call from JavaScript. This way, after Emscripten compiles our wasm, we can load the module, call C++ functions and get the return value, and interact with the wasm memory to use arrays, all within JS code.

3 Implementation

3.1 Fractal Generation on an HTML Canvas

In the initial planning phase when we began to build this application, we determined that a building a web application would be the ideal way to bring the features of FractalStream to an application that was most accessible. Given that, initial exploration involved determining the method in which we would paint a fractal onto a webpage. After briefly trying to learn WebGL and other libraries for graphics in JavaScript, we though HTML Canvas would be all we need for our application. Given this, the first step was to generate the image data array for a fractal, and put it on a canvas. This, is the core of FractalVoyager.

After an HTML canvas is in the DOM, we can *query select* it in plain JS, or use a *ref* hook to access it in react.

```
1 var can = document.querySelector("#myCanvas");
```

```
2 const can = canRef.current;
```

Now, we can get the 2D context which allows us to put 2D image data on the canvas. We can also programmatically set the pixel size by altering the width and height proprieties of the selected canvas element. Then, we can give the canvas context image data arrays to display. These will have length of `pixelsWidth * pixelsHeight * 4`. To generate these arrays, if we have a JavaScript "Uint8ClampedArray" (ex. pixelArr) that represents the RGB color value and transparency of each pixel, we can make an image data array from that by doing `let image = new ImageData(pixelArr, canWidth, canHeight);`, when "canWidth" and "canHeight" are the width and height of the canvas. To put this image on the canvas, we can call the "putImageData" method on our canvas 2D context, with the first parameter being our image data, and the next two being zero to say we are starting the drawing in the upper left corner.

Initially, we tried to generate the image data array for fractals in JavaScript, but since this tasks requires many iterations, it was far too slow for a useable web application. This, is why we used wasm. Say we have a C++ file with two functions (ex. genPixels and Oribt) that we want to call from JavaScript. We can use Emscripten to compile the C++ with "genPixels" and "Orbit" callable from JS, along with the built-in wasm/Emscripten methods: "malloc", "free", and "setValue" useable in JS with the following command.

```
1 emcc -O3 -sSINGLE_FILE=1 -sNO_EXIT_RUNTIME=1
2 -sEXPORTED_RUNTIME_METHODS=['ccall','cwrap']
3 -sEXPORT_ES6=1 -sUSE_ES6_IMPORT_META=0 -sEXPORTED_FUNCTIONS=
4 ['_malloc','_free','_genPixles','_orbit','setValue']  -sMODULARIZE=1
5 -sEXPORT_NAME='createModule' -s ENVIRONMENT='web' -sALLOW_MEMORY_GROWTH
     main.cpp
6 -o main.mjs‘
```

Which generates "main.wasm", and "main.mjs" that allows us to use wasm without coding in it. To use the wasm, we must first import "createModule" from "main.mjs" which we can call as a JS *promise* that returns a Module. We then use `.then(Module)` and write our code to handle interaction with wasm.

To call a function from wasm, we must first "cwrap" it. If our genPixels function has three parameters in C++ that are approximately equal to the "number" type in JS, and returns a "number"; we can initialize genPixels with the following:

```
1 let genPixles = Module.cwrap("genPixles", "number", [
2     "number",
3     "number",
```

```
    "number"
  ]);
```

Now, we can call genPixles as a regular JS function that takes three numbers and returns a number.

In the context of fractal generation, we could use two for loops to iterate through every pixel in the canvas (in JS), then call a function to generate how many iterations a point takes to escape that is in wasm with the above method. This was attempted, but again was too slow. Thus, the fastest possible option using wasm and canvas is to iterate through every pixel in the canvas in wasm, and pass JavaScript back an array representing the pixels. To use arrays with wasm and JS, we must access wasm memory. We first create a pointer to space in memory with a given array length.

```
let pixelsPtr = Module._malloc(
  arrayLength * Uint8Array.BYTES_PER_ELEMENT
);
```

Then, we copy data from the Emscripten heap, which is accessed directly with "Module.HEAPU8" when "Module" is our created wasm module, to a "Uint8Array" is JS.

```
let dataheap = new Uint8Array(
    Module.HEAPU8.buffer,
    pixelsPtr,
    arrayLength * Uint8Array.BYTES_PER_ELEMENT
);
```

Now, we can use either "pixelsPtr" or "dataHeap.byteOffset" as a pointer to our array in wasm memory which can be passed as a number parameter to our function. We are using "Uint8", so the array elements will be 8-bit integers from 0-255, which, conveniently, are the possible values for RGB colors.

Once we call the function from JS and we write to the pointer in C++, thus writing to the wasm memory, we can get the new array off the heap in the following way:

```
let pixelArray = new Uint8ClampedArray(
    dataheap.buffer,
    dataheap.byteOffset,
    arrayLength
);
```

This creates a "Uint8ClampedArray" that can be used to create an image data array which can be painted to the canvas as discussed above. Now, we can create a C++ function to iterate through every pixel in an HTML canvas, calculate a color value for each, put that in an array, pass the array back to JS, then paint the HTML canvas with that array to generate fractals quickly in a web browser. This method was tested to be quite fast (comparatively), so we choose to use this method in our final application.

Given a function that takes a complex number and returns the number of iterations needed to meet a fractal condition or zero if it is never met; the next implementation step is to create a function that can take parameters, and iterate through every canvas pixel and convert each pixel to a complex number, which will creates a desired range of numbers. Another feature of the application is dragging boxes which will zoom in on that portion of the fractal, so we may have a fractal (and thus image data arrays) that do not fill the entire canvas. The approach to implement this was to pass C++ the width and height of the canvas, the width and height of the fractal to be drawn, scale values for width and height, and start values for width and height.

Now, we can iterate through the portion of the canvas that the fractal will be drawn on with the following for loops; where "newCanWidth"/"Height" is the amount of pixels that we will draw the fractal in. Notice that we take the "floor()" of them, since they could be fractional, but we can't have a fraction of a pixel.

```
1 for (int x = 0; x < floor(newCanWidth); x++){
2     for (int y = 0; y < floor(newCanHeight); y++){
3     }
4 }
```

Inside these for loops, we must calculate the complex number equivalent of each pixel. Recall, the HTML canvas pixels start at 0,0 in the upper left, while the complex plane has (0,0) in the middle. The "default" (when scales and starts are 0) section of the complex plane is -1 to 1 for both real and imaginary axis. This calculation can be seen below, where x, y is the current pixel.

```
1 double screen_re = (x - width / 2.) / (width  / 2.);
2 double screen_im = -(y - height / 2.) / (height / 2.);
```

"screen_re" is the real part of the complex number, and "screen_im" is the imaginary part. The real part is along the horizontal axis, so is determined by the width. An x value of 0 should go to the smallest possible real value on our plane. Say we have a width of 2160. $(2160 - 1080)/1080$ will give us -1, as desired. Halfway will give us zero, and all the way 1, as desired. The same can be applied to the imaginary part, but we must multiple the final result by -1 since 0 on the canvas is the *largest* imaginary value, instead of the *smallest* for the real part. This is because negative to positive values in the complex plane go the same direction as the canvas horizontally (more positive the further

right), but the opposite direction vertically (more positive further down in canvas, and opposite in the plane).

We can adapt this simple formula to adjust for zooming in our out. For example, say we want to zoom in by a factor of two. In this case, all of the real and imaginary parts should be divided by two. One approach to do this, is to multiply the pixel value by .5. This will create a range half the size of the original, but also, for the real values, shift the plane to the by -1 *Re*. So, we must adjust for this by subtracting half of the width. This will work for any zooming in our out (for a zoom out: multiply and add half the width). The strategy also allows us to have "start" values to adjust for panning. For example, say the plane should be shifted by -1 on the real axis; we can add half the width to each x value so they start and end is shifted. Now, our final equation to convert a pixel value to a complex value is:

```
1 double screen_re = (((widthScale * x) + startX) - width / 2.) / (width  /2.);
2 double screen_im = -(((heightScale * y) + startY) - height /2.) / (height
    /2.);
```

Scales in the range $0 < S < 1$ will cause a zoom and the plane will have a range < 2. For scales > 1, the range will be > 2. Now, based on user inputs, we can adjust the scales, starts, and widths to represent the canvas as a section of the complex plane anywhere.

The last piece to generate fractals is coloring. In section 2.2, we discussed how to color a point in a fractal along a gradient based the ratio of how many iterations it took the point took to meet the condition and the maximum number of iterations. To include this in our implementation we must pass an array of colors to C++ from JS, as the colors are determined by the user and handled in JavaScript. We can put data on the wasm heap from JavaScript by using the "Module.setValue()" function that takes an address in memory, the value, and the type of data. The easiest way to have an array of colors is to have three arrays: one for the red color values, one for blues, and one for greens. The value in each array at index i is the color value for the specific color (red, blue, or green), for our i^{th} color. We first must allocate space on the wasm heap for each array:

```
1 let redPtr = Module._malloc(numColors * Uint8Array.BYTES_PER_ELEMENT);
2 let bluePtr = Module._malloc(numColors * Uint8Array.BYTES_PER_ELEMENT);
3 let greenPtr = Module._malloc(numColors * Uint8Array.BYTES_PER_ELEMENT);
```

Then, set the values:

```
1 for (let i = 0; i < numColors; i++) {
2     Module.setValue(redPtr + i, reds[i], "i8");
3     Module.setValue(bluePtr + i, blues[i], "i8");
4     Module.setValue(greenPtr + i, greens[i], "i8");
```

```
}
```

In C++, we must look up the indexes in the array for each color of each point, which can be done with the simple function below:

```
int getIdx(int x, int y, int width, int color) {
    int red = y * (width * 4) + x * 4;
    return red + color;
}
```

It takes the x,y value, the width of the canvas, the color to be placed (0 for red, 1 for green, 2 for blue), and returns the index. We can now assign colors to points:

```
int color = ceil((double)iterations*numColors/maxIters);
ptr[getIdx(x, y, width, 0)] = redPtr[color];
ptr[getIdx(x, y, width, 1)] = greenPtr[color];
ptr[getIdx(x, y, width, 2)] = bluePtr[color];
ptr[getIdx(x, y, width, 3)] = 255;
```

When iterations is the number of iterations the specific point took to meet the condition, “numColors” is the number of colors we have, “maxIters” is the maximum number of iterations, and “ptr” is a pointer to the beginning of our pixel array. “Color” gets set to the index of the appropriate color in our colors array.

Now, we have method for generating and displaying fractals in C++ (with wasm) and JavaScript on the web. In order to abstract the function to generate pixel data for as much user interaction as possible, we included additional parameters than the ones needed for colors and plane position. They are as follows:

- “type”: the type of fractal we are generating (0 for parameter plane, 1 for dynamical plane)
- “fixed_re”, “fixed_im”: *Re* and *Im* parts of a complex number used to fix the c value to generate a dynamical plane fractal from parameter plane.
- “maxIters”, “epsilon”, “minRadius”, “maxRadius”: these set the maximum iterations the fractal loop can go for, when to say the difference in magnitude between two complex numbers is small enough to call them the same, the magnitude to call a number 0, and the magnitude to conclude that the loop will escape to infinity, respectively.

These are in addition to the other parameters mentioned before:

- “startX”, “startY”: shifts to effect the start position of the plane

- “newCanWidth”, “newCanHeight”: the section of the entire canvas that the fractal will be drawn on
- “width”, “height”: dimensions of canvas in pixels
- “widthScale”, “heightScale”: scales
- “*ptr”: pointer to big array of pixel data
- “*redPtr”, “*greenPtr”, “*bluePtr”: pointers to the various color arrays.

The one aspect of fractal generation that has not yet been mentioned, but is worth considering, is generating orbits. Recall, orbits are simply the values of points in dynamical plane fractals after each iteration. To generate them, we can alter our fractal iteration function to set a value in an array after each iteration. We implemented an additional exported C++/wasm function to generate an orbit for any given point. It assign values of an orbit as double precision floats to the wasm heap for access is JS. The following is the lines that are added to a generic fractal generating function (just after the loops definitions) to make it for orbits, where “i” is our current iterations, beginning with 1, and “z” is our current complex number being iterated. We also must add the initial values of z before entering the loop.

```
1 ptr[i*2-2] = real(z);
2 ptr[i*2-1] = imag(z);
```

Recall, we are using the `#complex.h` C++ library, which has a built in functions to get the real and imaginary parts out of a complex number.

We can retrieve this data and free the memory from JavaScript the same way we did for “genPixels”, but now have an array of floats which represent an orbit in the form of alternating real and imaginary parts for each iteration. Now, we must convert these complex numbers into places in our canvas, so we can draw a line to represent the orbit of a point. The first step in this process is to reverse the equation to convert a canvas point to a complex number. Recall, this was the equation for width: `re = (((widthScale * x)+ startX)- width / 2.)/ (width /2.)`. We don’t need to worry about the start values (as we do this later), so can solve for x to yield: `x = (re * width)/ 2 + width / 2`. Using this equation, we created a helper function in JS that takes the real and imaginary part of a complex number, along with the height and width of the a canvas, and returns the canvas values that represent the complex number as follows:

```
1 const complexToCanvas = (re, im, width, height) => {
2     return [(re * width) / 2 + width / 2, (-im * height) / 2 + height / 2];
3 };
```

Using this function, we can build up a new array of canvas points by iterating through the array with JavaScript's ".forEach()" array method:

```
orbitArr.forEach((val, idx, arr) => {
    if (!(val === 0 && arr[idx + 1] === 0) && idx % 2 === 0) {
        newOrbit.push(
            complexToCanvas(val, arr[idx + 1], canWidth, canHeight)
        );
    }
});
```

Where "orbitArr" is our initial orbit array returned from C++. Notice, we only add to the array on even numbered elements, and use the element after it in the computation for the imaginary part.

Now, we must put this array of canvas points onto the canvas, which is achieved using the canvas 2D context methods ".moveTo()", ".lineTo()", ".beginPath()", and ".stroke()". The following code draws an orbit onto a canvas given that "p" is an array of arrays that represent the points to be included, and "ctx" is the canvases 2D context.

```
ctx.moveTo(p[0][0], p[0][1]);
ctx.beginPath();
p.forEach((cords) => {
    ctx.lineTo(cords[0], cords[1]);
    ctx.stroke();
    ctx.beginPath();
    ctx.moveTo(cords[0], cords[1]);
});
```

In this subsection, we have fully described the process of how FractalVoyager generates fractal image data (and orbits) in C++ with wasm, and paints it to the canvas with JS.

3.2 cdScript Language Compilation

For FractalVoyager, we developed cdScript, a complex dynamics scripting language that is based off of the scripting language used in FractalStream, which was based off the language used in FractalASM. cdScript implements the the majority of the FractalStream language, and nothing more. The documentation for the FractalStream language can be found here: `https://code.google.com/archive/p/fractalstream/downloads`. In cdScript, users define parameter plane fractals and dynamical plane fractals as scripts that describe what should be done to each pixel. As

stated in section 2.3, it is a declarative, high-level scripting language. In each script, a user defines the function to be iterated, what variable to iterate the function on, and what condition to end the function on.

The language includes variable definitions, many mathematical functions which can be used such as *sin*, and the ability to do certain actions only if the fractal is currently in a dynamical plane or parameter plane. This is useful since if a user defines a parameter plane fractal, the user interface can be used to *fix* the parameter to a complex number in order to generate dynamical plane fractals. cdScript reserves the variables z and c as special key variables. If c appears in the script, cdScript assumes that a parameter plane is being drawing, and will use c as the parameter. So, it will be set to the complex number representation of each pixel. In the parameter plane, z is always set to 0 initially, as a best guess for the critical point for the function. Thus, z is typically used in the iteration function, but the language allows for other variables to be used. In the dynamical plane, z is set to the value of each pixel, so is, once again, typically the variable used in the iterated function. If there is not c in the script, a dynamical plane fractal will be generated.

The basic structure of the language closely follows the basic structure of the grammar that we implemented in ANTLR to begin the compilation process. We will describe the grammatical structure, and in doing so, give more insight into the structure of a cdScript. Recall, lower case rules are parser rules, and upper case rules are lexer rules. The top-most rule which is the singular root node of every valid cdScript is: `script: (command '.')+;`. It can be seen that a script is a sequence of commands separated by periods, the following code snippet is the command rule grammar:

```
1 command: 'set' variable 'to' expression #SET_TO_COM |
2          ('block' | ':') (command)+ 'end' #BLOCK_COM |
3          'par' command #PAR_COM |
4          'dyn' command #DYN_COM |
5          if_then #IF_THEN_COM |
6          loop #LOOP_COM
7 ;
```

So, commands can set a variable to an expression, be a list of commands, do a command only in parameter plane or dynamical plane (par and dyn), be an if/then, or be a loop. Another noticeable feature to this grammar snippet is the "#TEXT" after each rule. These tell ANTLR to create separate rule nodes for every option of the rule, instead of just one for the rule. To continue with our example, we will further investigate the loop rule, since that is the base of all scripts in cdScript. Here is the grammar for a loop:

```
1 loop : 'do' (command)+ 'until' condition #LoopDo |
2          ITERATE expression 'on' variable 'until' condition #LoopIterateOn |
```

```
3          ITERATE expression 'until' condition #LoopIterateEmpty |
4          'repeat' n 'times' command #LoopRepeat
5  ;
```

These are the four types of valid loops in cdScript. The "ITERATE" styles are most commonly used in practice. The "#LoopIterateEmpty" option assumes that the variable that the loop will be iterated on is z, while the other version requires the variable that will be iterated on to be declared. As an example, let's consider the common Mandelbrot set representation of coloring values of "c" that cause $z \mapsto z^2 + c$ to escape to infinity. "c" is our parameter, and "z" is our critical point to be iterated. Thus, we can use the shorthand iterate loop, `iterate z^2+c until z escapes.` If `z^2+c` is our expression, and `z escapes` is our condition. We can also use the longhand, `iterate z^2+c on z until z escapes.`, which will produce the same result. Notice, we don't need to set a value for "z" since the initial "guess" of 0 by cdScript works for the Mandelbrot set. If we did need to reset the starting value for "z", we would use "par set z..." since if we hard set "z", then when switching to a dynamical plane, "z" will always get reset from the screen value to the set value, and our fractal will be one color.

In our example in the previous paragraph, we used many expressions and condition ("z escapes", two necessary building blocks of a cdScript script. Let's continue our discussion with the grammar for a condition.

```
1  condition : expression (GT | LT | GT EQUALS | LT EQUALS | EQUALS) expression
       #COMP_COND |
2       expression 'escapes' #ESCAPES_COND |
3       expression 'vanishes' #VANISHES_COND |
4       expression (STOPS) #STOPS_COND |
5       condition (OR | AND | XOR) condition #COMB_COND
6  ;
```

The basic comparison operators are supported such as $>$, $\leqslant$, exclusive or, and so on. In addition to these basic operators, we have "escapes", "vanishes", and "stops". "Stops", is written above as a lexer rule because having lexer rules allows us to check if that rule is in a rule context when compiling. This is useful for stops as it is a tricky condition to compile to C++. The stops condition is met when the value of an expression is within a user defined epsilon value of the value of that expression in the previous iteration. "Escapes" is met when an expression's value is greater than a maximum radius defined by the user, signalling that the expression will eventually escape towards infinity. "Vanishes" is the same as "escapes" except for converging to 0, and within the defined *minimum* radius.

Now, let's consider the grammar for an expresion.

```
1  expression: (PLUS | MINUS)? atom #SIGNED_ATOM_EXP |
```

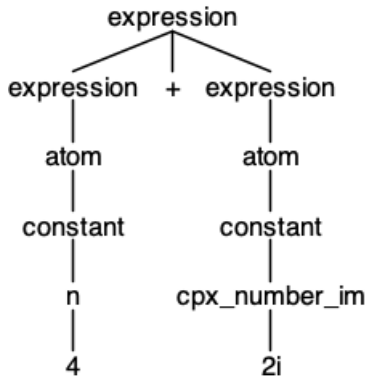


Figure 12: The parsing of "4+2i" by cdScript. Notice that there is no concept of complex numbers in parsing, only real and imaginary numbers.

```
2               expression POW n #POW_EXP |
3               expression TIMES expression #TIMES_EXP |
4               expression DIVIDE expression #DIVIDE_EXP |
5               left=expression PLUS right=expression #PLUS_EXP |
6               expression MINUS expression #MINUS_EXP |
7               cpx_function LPAREN expression RPAREN #CPX_FCN_EXP|
8               LPAREN expression RPAREN #PAREN_EXP
9   ;
```

This defines the recursive nature of an expression, and allows for mathematical functions such as + and *sin* to be used. *sin* and other more compilated functions are defined in the parser rule "cpx_function". The grammar reveals that the building blocks of expressions are atoms, which here can be assigned to a positive or negative value. An atom is a variable or constant. A variable is a lexer rule defined with regex and ANTLR's version of regex: `VARIABLE : ('a' .. 'z')+ ;`. This defines a variables as a sequence of characters. Also, cdScript is case-insensitive, so any sequence of characters is valid here no matter capitalization. A constant is either a rational number, or a rational number followed by an i to define an imaginary number. The approach we took to handle complex numbers, was to only have a concept of real numbers and imaginary numbers separate when parsing, then in compilation they are combined. This can be seen with the above grammar, as $4 + 2i$ will get parsed as two atoms which are expressions, then added as seen in figure 12.

With our grammar for atoms and expression, we can see how a variable such as z is an expression, so `iterate z^2+c until z escapes.` is a valid script using rules discussed above. Once, the grammar is defined in ANTLR, the next step is to use the ANTLR tool to generate a parser/lexer in our target language, C++. Once we have ANTLR

installed and the "antlr" command in our classpath, we used the following command to generate the parser/lexer and the defaults for the visitor pattern from our grammar, "Fractal.g4": `antlr4 -Dlanguage=Cpp Fractal.g4 -visitor -no-listener`. This generates C++ files such as "FractalVisitor.cpp/h" and "BaseVisitor.cpp/h", which create the default visitor that performs a depth-first traversal of the nodes by visiting each one. On top of these files, we created "main.cpp" and "MyVisitor.h". "Main.cpp" contain the methods that will be compiled to wasm and called from JavaScript. One is "cgen()" that takes a string that is the script to be compiled, source-to-source compiles the script to C++, then returns the length of the code. The following code snippet takes the input, creates the ANTLR lexer, uses the lexer to generate tokens, parses those tokens to create our parser tree, gets the root node of this tree (always "script"), then initializes our visitor class.

```
1 antlr4::ANTLRInputStream input(stream);
2 FractalLexer lexer(&input);
3 antlr4::CommonTokenStream tokens(&lexer);
4 FractalParser parser(&tokens);
5 FractalParser::ScriptContext* tree = parser.script();
6 myVisitor visitor;
```

These use the built-in methods from the ANTLR run time library, which, becuase we are compiling this to wasm, must also be compiled to wasm.

Once this is set up, we have generated an abstract syntax tree, so can begin the next step of language generation; either interpretation or compilation. Originally, we tested an interpreter for this language by using the visitor nodes to pass and edit data, but this was far too slow, so we built a code generator to C++ that will be described. Throughout this process, we used C++ stringstreams, which allow programmers to concatenate strings easily with the follow syntax: `bigLoops << "EMSCRIPTEN_KEEPALIVE " << fcnName` when "bigLoops" is a stringstream and "fcnname" is a regular C++ string. We can then create a C++ string from a stringstream by simply uses the ".str()" method. Recall, the C++ we are generating will is the fractal generating code described in sections 2.2 and 3.1, and will eventually be compiled to wasm.

The first step to our code generation is to generate all of the lines which do not change based on the script passed in. These include the pixel to number conversions, the function definitions, the coloring method, and many other sections described in section 3.1, and seen in appendix C. This straightforward, and we just add text to stringstreams. Next, we must traverse our AST with ANTLR to generate the code specific to our script. Recall, ANTLR automatically generates a visitor parser which visits each rule node, so to build up a the code, we will add to a global stringstream at each parser node.

We will explain this process with an example, the one we have been following throughout the report: `iterate`

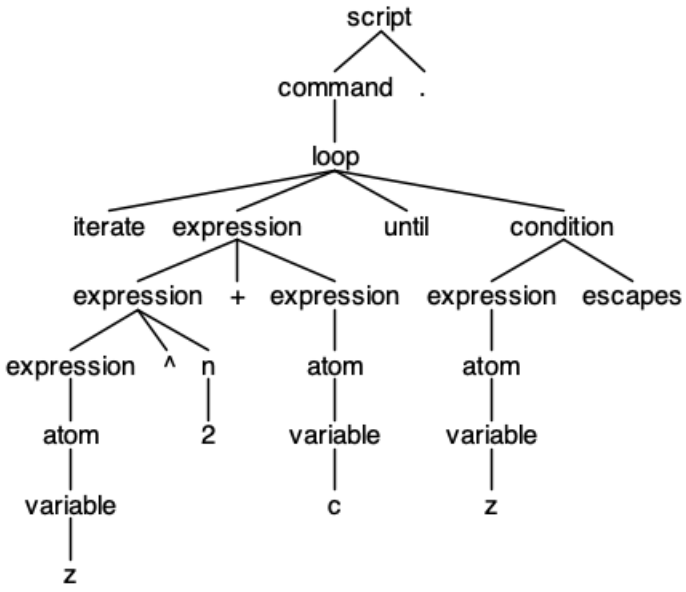


Figure 13: cdScript parse tree for `iterate z^2+c until z escapes.`

`z^2+c until z escapes.`. First, consider the parse tree generated by ANTLR for this script in figure 13, based on our grammar. Note that this is the parse tree, and is not a true representation of the AST generated by ANTLR, but will help as a visual aid for the compilation process. The first node which will be visited is script, which needs no override since it will visit all children by default. Next, we visit a command, specifically, a "LOOP_COM" command. This also needs no overriding because we don't need to add anything to the output for this command. Next, we visit a "LoopIterateEmpty" loop rule. This will need to be overridden, as we need to add to the compiler output. This can be achieved by adding a method to our "MyVisitor.cpp" file. Below is how we override the default function for a rule node. The default is simply: `return visitChildren(ctx)`, which continues the depth-first traversal.

```
1 virtual antlrcpp::Any
      visitLoopIterateEmpty(FractalParser::LoopIterateEmptyContext *ctx)
      override {
2 }
```

Loops become quite complicated to code generate with the addition of the stops command, so we will briefly describe the process for "LoopIterateEmpty" node, assuming there is no stops command and no orbits which need to be generated. This will vary from our actual implementation.

First, we can add the for loop declaration to our stringstream named "output": `output << "for(int i = 1; i < maxIters; i++){\n"` where "maxIters" is a parameter in our eventually function. Next, since we know this function will be iterated on "z", we can add `output << "z = ";`. Now, we want to assign the value of the expression

to “z”, so we can visit the expression, which will add to our output: `visit(ctx->expression());`. We are working with “ctx”, which is passed to the function, and represents the rule with associated built-in ANTLR methods. Here, we are accessing the expression in our context. Next, we close the line started by the expression with a “;”, and start an if condition that will hold our condition: `output << ";\nif("`. Now, we can visit our condition, which will populate the if within our condition: `visit(ctx->condition())`. Then, close our if and add a return of i, which will return the current number of iterations if the condition is met: `output << ") {\nreturn i;\n}\n\n"`. Now, we have the structure of a loop which runs for the maximum number of iterations, does the expression on our given variable, “z”, and returns “i” (the number of iterations) if our condition is met. We don’t need to visit any children of our loop rule since we handled all actions by accessing specific children with our context.

When we call `visit(ctx->expression());`, the evaluation of our expression is added to the output. The top-most expression for loop expression is `z^2+c` which is a PLUS_EXP. Below is the overridden plus expression method. Notice how we simply recursively visit the children and add a plus sign between them.

```
1   virtual antlrcpp::Any visitPLUS_EXP(FractalParser::PLUS_EXPContext *ctx)
        override {
2     visit(ctx->expression(0));
3     output << "+";
4     visit(ctx->expression(1));
5     return ctx;
6   }
```

The left expression is a POW_EXP, which is overridden with this method:

```
1  virtual antlrcpp::Any visitPOW_EXP(FractalParser::POW_EXPContext *ctx)
       override {
2      int n = stoi(ctx->n()->getText());
3      if(n == 0) {
4          output << "1";
5      }
6      for(int i = 0; i < n; i++) {
7          if(i != 0) {
8              output << "*";
9          }
10         visit(ctx->expression());
11     }
```

```
12      return ctx;
13  }
```

Here, we get the power we are computing from taking the integer value of the string in our “n” rule, then looping up to that number and outputting the expression inbetween multiply signs. When an atom expression gets visited, we simply output the text of the variable, or text of the number. We also check if the variable is simply “c”, and if so, set a global boolean variable so we know this initially draws a parameter plane fractal. So the expression visited here will just output “z”. It can be seen that the same happens for the right side of the plus, so lets move on to our condition.

```
1   virtual bool visitESCAPES_COND(FractalParser::ESCAPES_CONDContext *ctx)
        override {
2       output << "abs(";
3       visit(ctx->expression());
4       output << ")␣>␣maxRadius";
5       return false;
6   }
```

The abs(), or magnitude, of the expression is outputted, here will be “abs(z)”, and then “> maxRadius” is added. This simply checks if the magnitude of our expression (“z”) is greater than “maxRadius”, meaning it escaped and we should return the iteration number. This completes our code that involves traversing the tree for our simple script. All together we have:

```
1  for(int i = 1; i < maxIters; i++) {
2      z = z*z+c;
3      if(abs(z) > maxRadius) {
4          return i;
5      }
6  }
```

We also need to add our variable definitions, so we prepend `"std::complex<double> z(z_re, z_im);\n"` to the beginning of the stringstream, along with `"std::complex<double> c(c_re,c_im);\n"` if we are in the parameter plane. Also, we must add “return 0” to the end of the stream, meaning if we make it through the entire for loop, the condition was not met and it will be colored black. This completes our code generation for the main “genPixels” function, but the “genOrbit” function must also be compiled. As stated in section 3.1, to generate an orbit, we simply add to an array after each iteration of the loop. Our method for implementing this was to do an additional pass

through our tree with a boolean set that indicates we are now code generating an orbit. The same exact code is generated, except after a for loop declaration, we add to an array based on the value of the variable being iterated, and the current index. Given this method, orbits currently are only supported for the two loops with iterate, and does not work for the other two loops. A full output of the compiled script can be found in the appendix.

Recall that this process is written in C++, but is running in a React app and web browser. Thus, we must compile all of our files, along ANTLR generated ones, and even the entire ANTLR runtime library to Web Assembly (make file for this can be found in the source code, which was modified from an existing repository [16]). The exported methods to wasm are "cgen()" which takes a script and returns the length of the code, and "getCgen()" which takes a pointer, populates the array with the code, and returns the initial type (0 for parameter, 1 for dynamical). This is called in React using a custom hook: "cgenHook.js". From here, a user can generate a dynamical plane fractal from parameter plane by fixing a "c" value, or generate an orbit in a dynamical plane fractal by picking a value for "z". One glaring issue with the currently flow is that we have C++ code, which is simply a string, sitting in the browser with no way to run it. As stated in the introduction, this C++ code gets compiled to wasm which will live in a virtual file system by a tool called emception.

3.3 C++ to Web Assembly In-Browser Compilation

To compile C++ code to wasm, outside of the browser, we used Emscripten, inside the browser, we use emception. This is a tool created by Jorge Prendes which includes clang/llvm and binaryen (the C++ compiler tools Emscripten uses) compiled to Web Assembly. There are many intricacies to this complicated tool that will not be discussed here. At a very high level, it uses many tools which are compiled to work in the browser, then builds all of these tools to binary files. As the emception repository stands (https://github.com/jprendes/emception), it is a C++ code editor running in the web. One can edit the way the C++ code gets compiled with various Emscripten command line arguments. Kevin Angstadt then took this repository and rebuilt it to make the compilation process the main focus by stripping the user interface and tweaking the tool (and thus build). Now, we can easily write a string representing C++ code to the emception virtual file system, then use an Emscripten command to compile it to wasm, and receive that compiled wasm which can be dynamically imported in JavaScript.

To integrate this process with React, we stored a built version of Angstadt's modified emception in the public folder of our React app and created three custom hooks. One to initialize emcepiton, one to compile code with it, and one to run that compiled code. To initialize it, we create a *web worker*, then wrap that web worker with *Comlink*, and initialize the worker. The wrapped worker is then saved in a global variable, so it can be accessed by the other hooks. The basic idea behind a web worker is that it runs asynchronously and in the background of an application. Comlink makes these easier to work with. Once emception is initialized, we can compile code with it. Below is a

simplified version of the hook to compile code with emception.

```
const useCompileCode = (code) => {
    const ready = useCompileStore((state) => state.ready);
    const setContent = useCompileStore((state) => state.setContent);
    useEffect(() => {
        const compileCode = async () => {
            try {
                await emception.fileSystem.writeFile("/working/main.cpp",
                    code);
                const cmd = `emcc -O3 -sSINGLE_FILE=1 -sNO_EXIT_RUNTIME=1
                    -sEXPORTED_RUNTIME_METHODS=['ccall','cwrap']
                    -sEXPORT_ES6=1 -sUSE_ES6_IMPORT_META=0
                    -sEXPORTED_FUNCTIONS=['_malloc','_free','_genPixles','_orbit',
                    'setValue']  -sMODULARIZE=1 -sEXPORT_NAME='createModule'
                    -s ENVIRONMENT='web' -sALLOW_MEMORY_GROWTH main.cpp -o
                    main.mjs`
                ;
                const result = await emception.run(cmd);
                if (result.returncode == 0) {
                    const content = await emception.fileSystem.readFile(
                        "/working/main.mjs",
                        { encoding: "utf8" }
                    );
                    setContent(content);
                }
            }
        }
        if (ready) {
        compileCode();
        }
    }, [code]);
};
```

The “useCompileCode” hook first obtains a global state variable, “ready”, and a global state function, “setContent”. Then, a “useEffect” is used, which is a React hook that runs every time any of the variables in the array at the end of it change. In this case, our useEffect runs every time “code” changes. This is needed because these custom React hooks exist at the top level of react components, so cannot be called conditionally. So, when the passed in value of code (which is the C++ source code string) changes, we enter the useEffect. Inside the useEffect, there is an additional async function that first writes the code to the virtual file system. Then, we define the Emscripten command which is the same one used in section 3.1. Now, we use the global emception variable (which is a web worker) to run the command. If the compilation succeeds, we will get a return code of 0, then read the .mjs file to interact with the wasm off of the virtual file system, and call “setContent()” with it as a parameter. What this does is set a global state variable to the contents of the .mjs file, is still simply as a string. We only call this function to compile code if the “ready” global state is true, which is set to true when emception is initialized.

The final hook used to interact with emception is the “useGenPixels” hook. It, along with most custom hooks in FractlVoyager, is set up like “useCompileCode” with a function nested in a useEffect. “useGenPixels” takes all of the parameters that get passed into the “genPixels” or “genOrbit” function described in the previous two sections. One step before being able to call the two functions is to import the wasm .mjs module which is currently stored in “content” as a string. This is done with the following lines of code:

```
1     function doimport(str) {
2     const blob = new Blob([str], { type: "text/javascript" });
3     const url = URL.createObjectURL(blob);
4     const module = import(/* webpackIgnore: true */ url);
5     URL.revokeObjectURL(url);
6     return module;
7 }
8 const createModule = (await doimport(new Blob([content]))).default;
```

In the above code snippet, we create a “blob” object which contains the .mjs module content, create an url for the blob, then import the module by doing a JS dynamic import of the created url. One import step is to add the “webpackIgnore” string into the import statement which makes it so when webpack bundles all of JS files, it does not try to do any special optimization or bundling to this line which would cause it to break. After we have the module, we take the “.default” of it, which is the “createModule()” function described in section 3.1. This function can be called, then “genPixels” and “genOrbit” will be cwraped.

After genPixels and genOrbit have been cwrapped, they are called with the given parameters, and the hook eventually returns either image data (for parameter and dynamical plane fractals) or an orbit. This is done the same

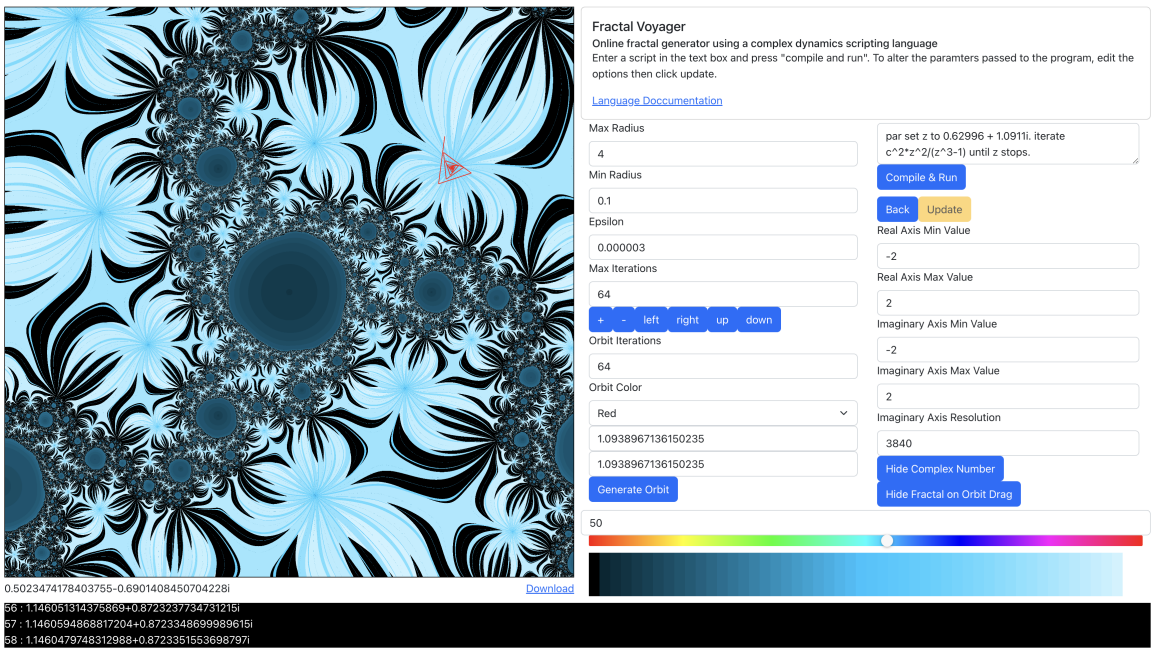


Figure 14: A screenshot a FractalVoyager, with a dynamical plane fractal based on the entered parameter plane fractal displayed, along with an orbit

Figure 15: The terminal output after a dynamical plane fractal is generated based on a parameter plane fractal

as described in section 3.1 by accessing the wasm memory heap.

3.4 Web Application

Now that we have discussed our language compilation process (3.2), the code which is produced that generates fractal image data and orbits (3.1), and how we can use these in a React app (3.3), we will discuss the user interface that was built to bring these aspects together in a user-friendly way.

Figure 14 is a screenshot of the user interface of FractalVoyager. In this UI, after emception is initialized, the "Compiled & Run" button becomes clickable. Users enter scripts and click the button which then go through the process of being compiled by our code generator and then emception, which creates the wasm module and is run to generate a fractal image. Throughout this process, information is given to the user through the terminal component seen in the lower section of the screen. The terminal dumps the emception compiler output and tells the user if there is a failure in compilation, along with giving information such as the orbits or what complex number was fixed for a dynamical plane fractal, as in figure 15.

If a parameter plane fractal is generated, the user can click a point to fix that point for a dynamical plane fractal,

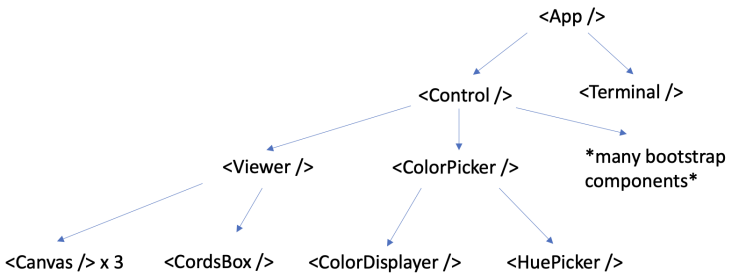


Figure 16: Simplified version of FractalVoyager's component hierarchy

or enter the complex number in the boxes (one for *Re* part and one for *Im*) and click "Generate Dynamical Plane". That button figure 14 says "Generate Orbit" because that is what a click on the fractal or number in the boxes does when the dynamical plane is displayed. When the fractal has yet to be generated, the button is grayed out along with the input fields to enter the numbers. The orbit is drawn on the canvas as a sequence of red lines, and the complex numbers are dumped to the terminal. Users may explore the fractal by dragging boxes on the canvas which zoom in on that box, by using the zooming and panning buttons in the middle of the left column of options, or by resetting the axes values on the right. The min and max axis values change to reflect box zooms. A user may also change the color gradient by using the slider, and change the amount of colors used by changing the number above the slider. The parameters that are passed to the fractal function ("maxRadius", "minRadius", "epsilon") can be edited as well, along with the amount of iterations to do an orbit for, and the color of the orbit line.

Whenever any of the inputs boxes, excluding the complex number for generation boxes or the color slider/number, change, the yellow update button becomes un-grayed out and a click updates those changes to the fractal. Other features include being able to download the fractal image as a ".png", a button that makes a drag on an orbit make the dynamical plane fractal go away (a useful feature in teaching complex dynamics), a back button to render the previously drawn fractal or orbit, and the complex number representation of the pixel that the mouse is hovering over to be displayed.

To best understand the implementation of the user interface, consider figure 16 which is a visual representation of FractalVoyager's component hierarchy.

Instead of explaining the props and state of each component, we will go into some detail on the state, props, and functions that the application uses to take a user entered script and options to render a fractal on the canvas. In doing so, we will describe many of the methods that were used for other components in FractalVoyager. To begin, the control component handles the call to our "cgen" hook which compiles the cdScript to C++ by using our C++ wasm compiled language generation. The functionality of this is implemented with a useRef and useState hook. There is a state variable, "script", defined as: `const [script, setScript] = useState(null);`, and a ref called

“inputRef”. Refs allow programmers to directly access HTML elements in React, and when their value changes, it does not trigger a rerender. State variables, on the other hand, hold any value and trigger a rerender when they change. Consider the below code snippets which are the JSX for the script box and the "Compile & Run" button.

```
<Form.Control
    as="textarea"
    ref={inputRef}
    type="text"
    placeholder="Enter Script"
    id="script-area"
></Form.Control>
```

```
{compileReady ? (
    <Button
        variant="primary"
        onClick={() => {
            setScript(inputRef.current.value);
        }}
    >
        Compile & Run
    </Button>
) : (
    <Button
        variant="primary"
        disabled
    >
        Compile & Run
    </Button>
)}
```

The first, is the HTML to render the text box. It is a Bootstrap-React component, and has a defined ref of “inputRef”, so “inputRef” will refer to that HTML element. The second uses a turner expression; if the state variable “compileReady” (set to true when emception is initialized) is true, they we allow the button to be clicked, and if not, we gray it out. If the button is clicked, we set the state of “script” to the current value of the text box. When this

state value changes, a rerender is triggered. Since our "useCgen" hook exists at the top level of the component, it will get called every rerender. Inside the cgen hook, the actual function to compile code only gets called when the script value changes (achieved with a useEffect), so even if we have rerenders where the script doesn't change, we will only compile code when we need to.

After "useCgen" uses wasm to compile the script to C++, a global state variable changes that "useGenPixels" in bound to (achieved by simply using the variable in it), so "useGenPixels" reruns. There is a few small sets in between, but that is generally how we go from a script to image data. Now, let's consider how we handle updates is user options to affect the fractal, along with the general state management strategy of holding the options data and the parameter data for the "useGenPixels" hook.

There are two state variables which hold the data in our options. One, is the values in the options at any given time and is stored locally in the control component, called "tmpParams". The other, stored in global state, is the values of the options that were last used to generate a fractal, "tmpParamsStore". The value of each control option component is tied directly to the values of "tmpParams" instead of using refs and state. This is achieved by hard setting a components value to the state variable, and changing the state on change. For example, consider the below code snippet for the orbit iterations option component.

```
1 <Form.Label>Orbit Iterations</Form.Label>
2 <Form.Control
3     type="number"
4     value={tmpParams.orbitNum}
5     onChange={(e) =>
6         setTmpParams({ ...tmpParams, orbitNum: e.target.value })
7     }
8 ></Form.Control>
```

When any option is changed, the new values in "tmpParams" are compared with the values in "tmpParamsStore", and if any of them are different, we allow the update button to be clicked. When the update button is clicked, the values of "tmpParamsStore" are updated, along with this, another local state variable, "params", is updated. The "params" state variable holds the values of the parameters that will get passed to "useGenPixels", which are the parameters used in our C++/wasm fractal generating program. There is a middle step to convert what we have in options to what is needed for the parameters. Mainly, we need to convert the min and max values for the real and imaginary axes to the scales and starts values mentioned in section 3.1 that our generating function takes.

Once the conversions are done, we pass the values of "params" as props to our viewer component. Inside the viewer component, we have a state variable, "genPixelsParams" which stores all of the parameters for "useGenPixels".

There is a useEffect which triggers on change of the props, and updates “genPixelsParams”. This additional step is needed as there are actions/state inside the viewer component which effects the parameters. So, “genPixelsParams” is not solely dependent on the props. When “genPixelsParams” changes state internally in the viewer component, functions are called and useEffects are triggered which updates the global “tmpParamsStore”, which then triggers a rerender of the control component. In the control component, there are useEffects for the values of “tmpParamsStore” that update the values of “tmpParams”, thus changing the values in the options. The main place to see this process is a box zoom triggerring an update of the min and max axes values.

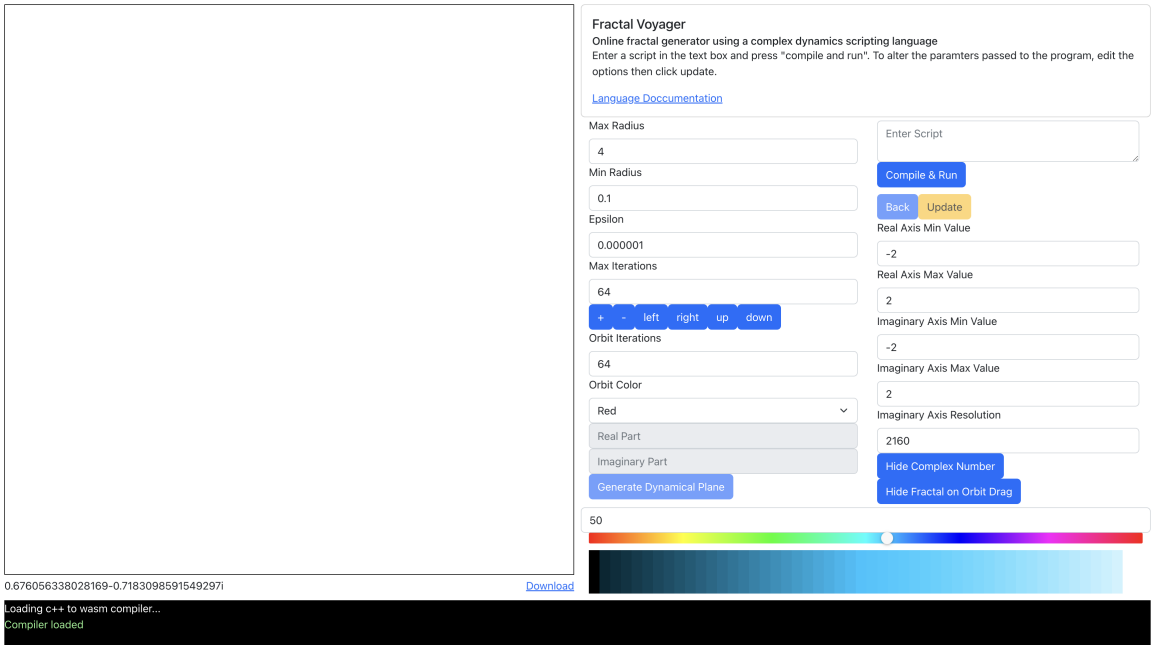
At the top level of the viewer component, there is a call to the “genPixeles” hook with all the “genPixelsParams”. Thus, this reruns every time the viewer component gets rerendered, but inside the hook there is a useEffect which makes it so the re-generation of image data/orbits only happens when the parameters change (much like our “useCompileCode” hook). Once our “genPixels” hook runs, it returns the image data or orbit to the viewer component. There is a function to draw the fractal by painting the canvas (as described in 3.1) which gets passed down as a prop to the canvas component for the fractal canvas. This function includes the variables which is returned from “useGenPixels”, and since the function is a prop, the variable in it changing triggers a re-render of the canvas, and thus a redraw, painting our new fractal.

The above discussion of props and state involved in updating and drawing a fractal is a slight simplification of the process. We also have to deal with what type of plane we are drawing, along with the handling the back button to go to the last fractal or orbit drawn. This is achieved with a state variable that is basically a stack of parameters. There are a few other intricacies we must handle, which mostly come down to a couple small state variables and more useEffects. The other main process involved in the overview of drawing a fractal is handling box zooms, which will not be discussed.

The rest of the React components are built similarly to the two described above, as is the flow of the overall application. See appendix A for the entire source of the application, well commented.

4 Worked Example

In this section, we will go through an example of using FractalVoyager with a simple script. In doing so, we will further explain the features of the application, building off of section 3.4. When the app is first opened, we will get the below screen. Once “Compiler loaded” gets logged, the “Compile and Run” button will be clickable, and a user can enter a script. Below is a screenshot of the app after the compiler (emception) has been loaded.



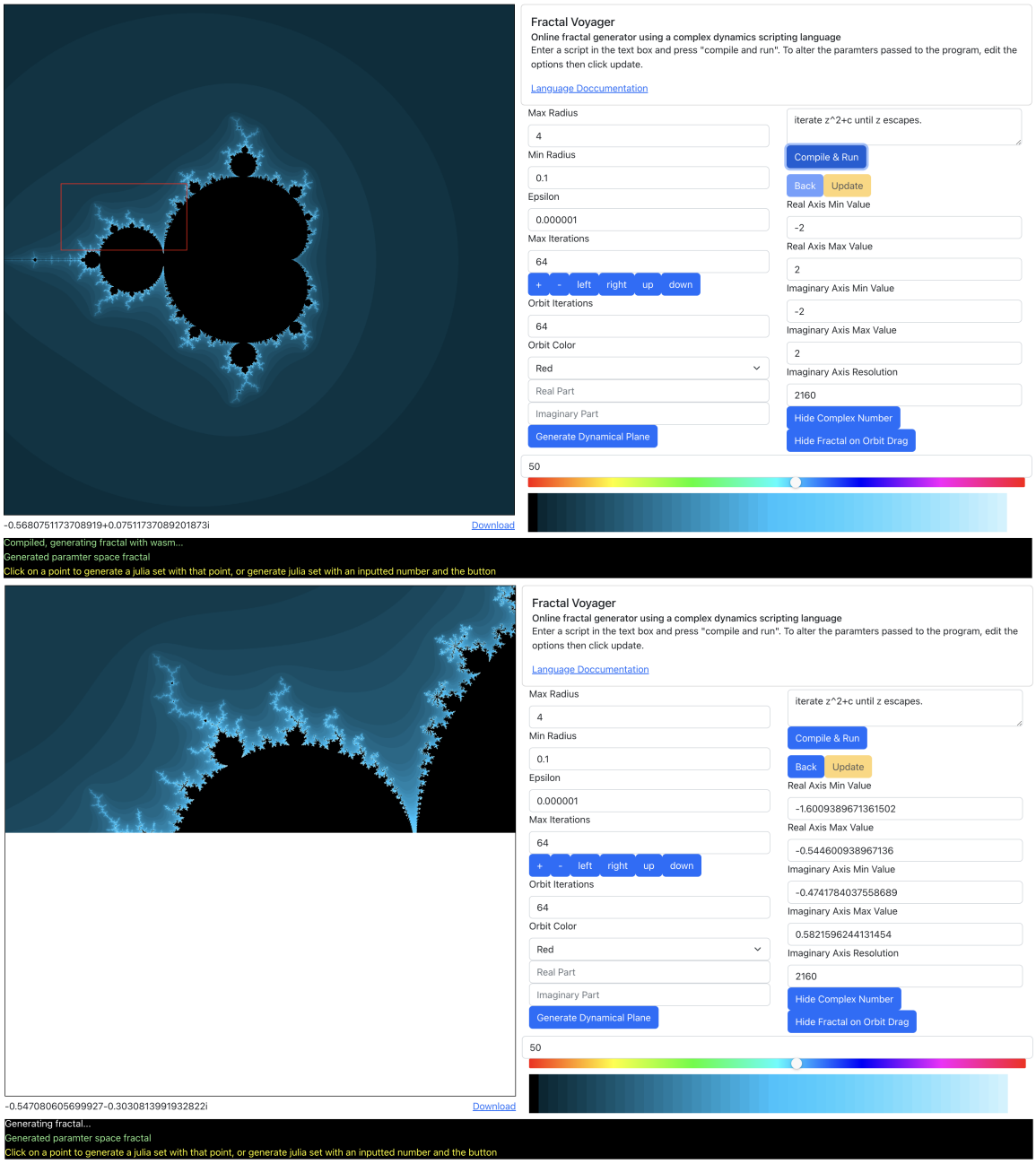
Then, we can enter a script in the box, and press compile. The output of the in-browser c++ to wasm compiler will be logged, and the fractal will appear on the screen with some additional terminal logging to provide useful hints. After the fractal is on the screen, a user can:

- Zoom and/or pan
- Generate an orbit or dynamical plane fractal (depending on what type was just generated)
- Click the show/hide buttons to show/hide the complex number equivalent of the pixel that our cursor is over or the dynamical plane fractal when an orbit is drawn and dragged
- Edit any of the options on the screen and click “update” to alter the fractal image
- Click the back button go back to the previously drawn fractal
- Download the current fractal image
- Enter a new script

Let’s begin by discussing zooming and panning. There are three distinct ways to zoom and/or pan the fractal:

1. Box zooming
2. Zooming and panning buttons
3. resetting axes (by changing the min and max values then clicking “update”)

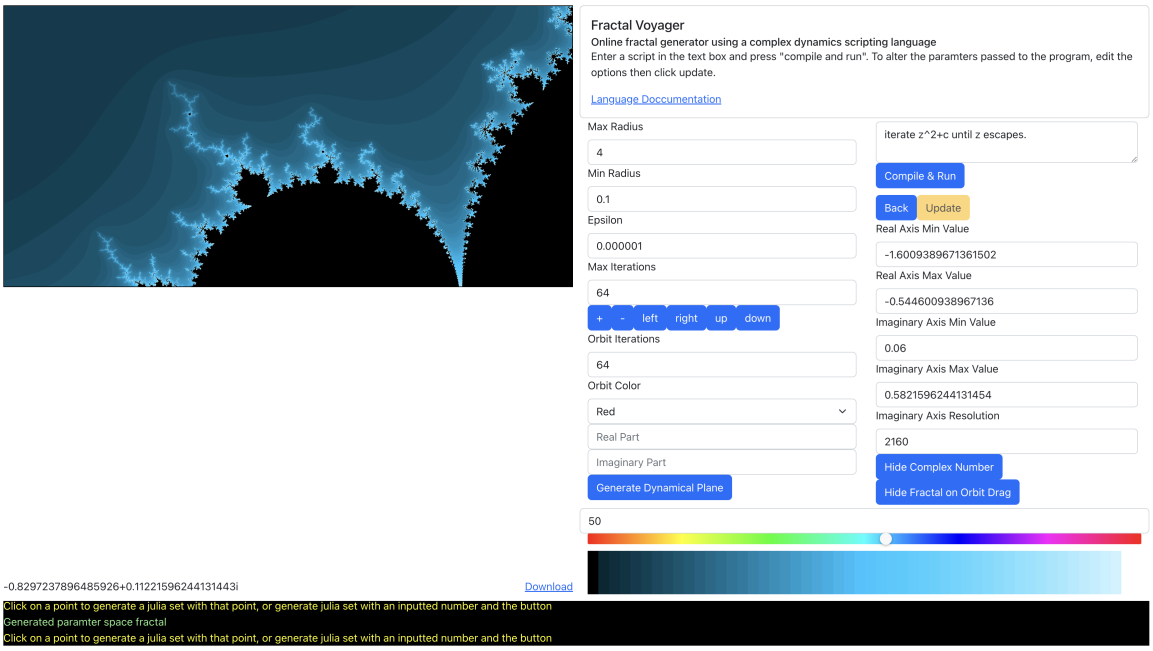
To box zoom, we can click and drag a red box on the fractal, which will zoom in on that particular section, while keeping the original canvas size. So, the axes values in the options panel will be reset. Notice, these values will be of the *entire* canvas, not just the area that the fractal is drawn on. See the below images for a mid box zoom, and the resulting image using the typical Mandelbrot script: `iterate z^2+c until z escapes.` with the default options.



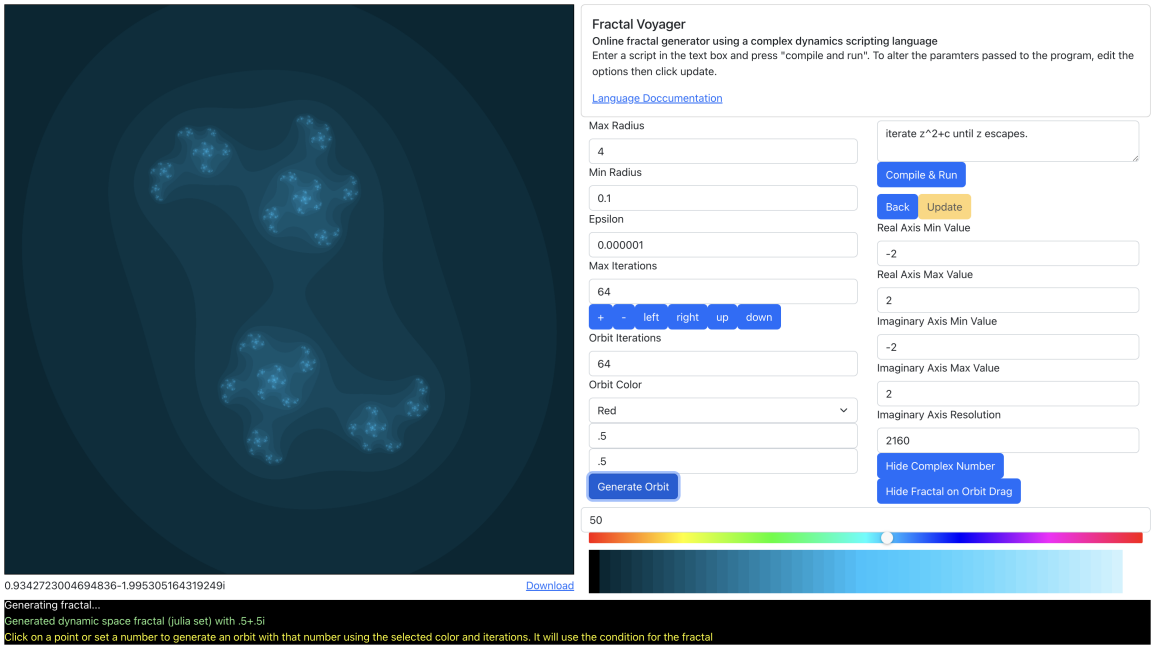
To zoom and pan with the buttons, simply click on them. We do not need to click the update button; any click on the buttons will directly effect the fractal and will reset the axes values. Zooming and panning this way will keep the fractal taking up the entire canvas, and will not reset the canvas size.

The other way to zoom and/or pan is by altering the axes values. To do this, change the min/max values in the

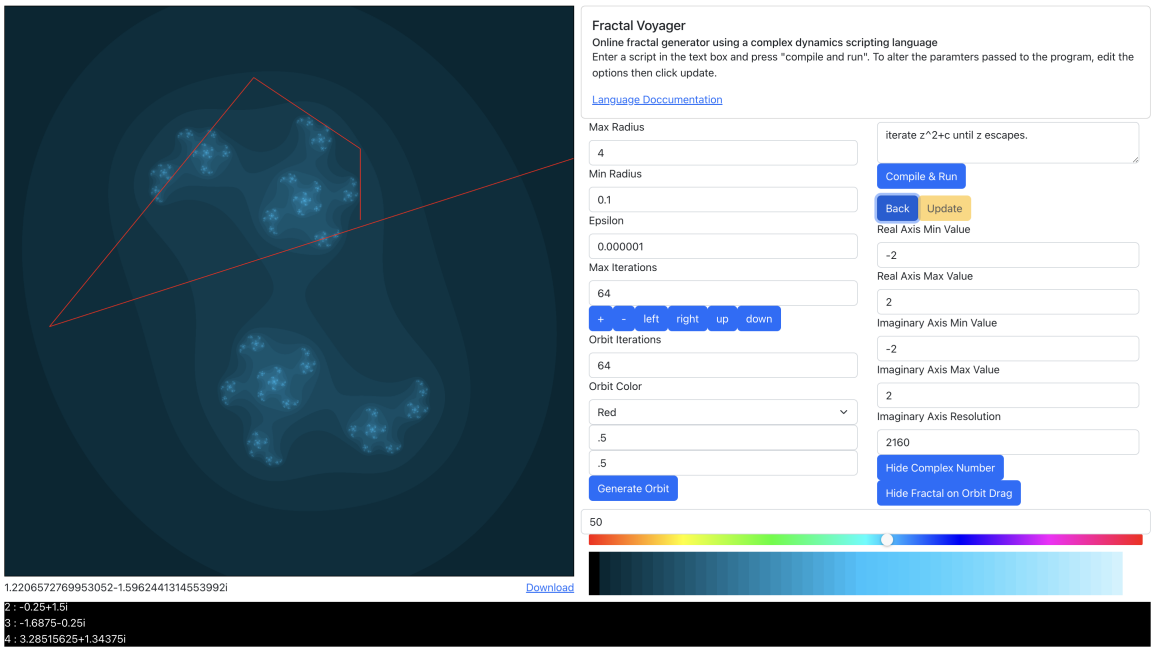
control panel. Any other zooming that has happened will reset these values, so this can be done at any point. After we have edited the values, we must click “update” for the fractal to update. A key distinction should be noted for this type of zooming/panning. Altering these values will reset the *canvas size*, not just the area the fractal takes up. Thus, if you change the aspect ratio here, the canvas size will be reset to that aspect ratio. This is useful when downloading fractal images, as if we download an image after a box zoom, the area that the fractal is not in will still be in the downloaded image. Instead, we can alter the axes values to change the aspect ratio of the canvas and get a resized downloaded “.png” of the fractal. Below is an screenshot of the application after a canvas resize by resetting the axes values.



Now, lets consider how to generate an orbit or dynamical plane fractal. In a parameter plane fractal, such as our Mandelbrot script, we can click on a point to fix that complex number equivalent as a value of “c”, or use the boxes in the lower left. To use the boxes, enter the real part and imaginary part of the complex number to fix as “c”, and click “Generate Dynamical Plane”. With either of these methods, a dynamical plane fractal will be generated based on our original script and our new “c”. This is either the complex number equivalent of where we clicked, which can be seen if “Show Complex Number” has been clicked, or by default, below the fractal. Or, it will be based on the number we entered. This “c” will also be logged in the terminal. As an example, if our script is `iterate z^2+c`, and we enter “.5+.5” in the box, our new fractal will be equivalent to the script `iterate z^2+.5+.5i`. See the below image for this example.

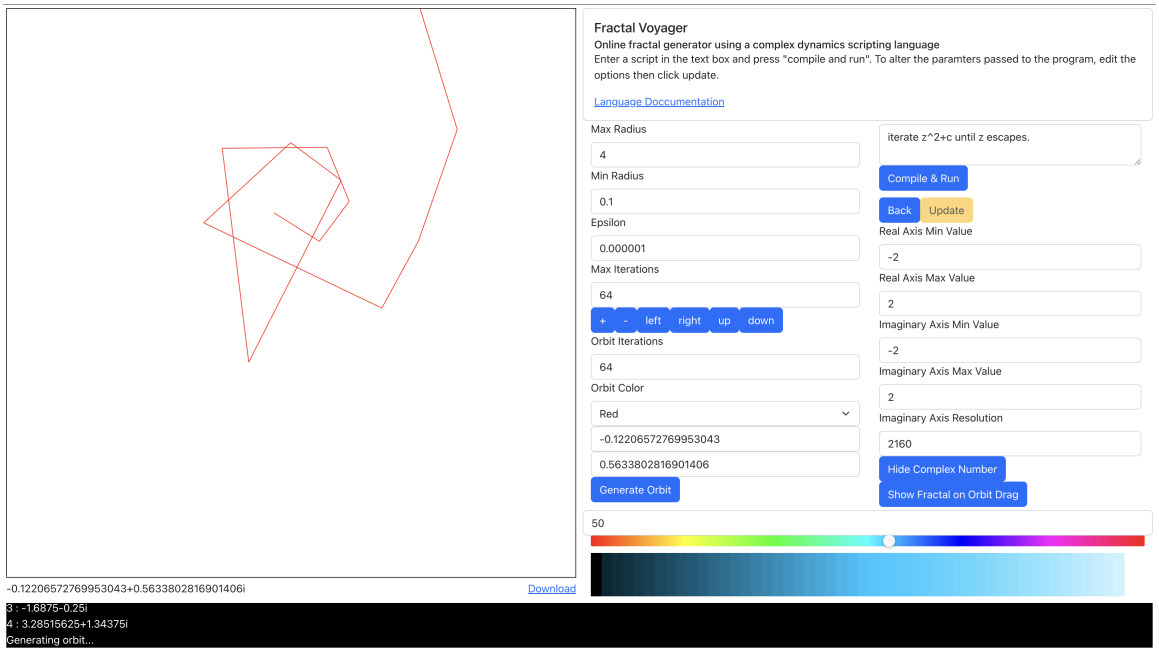


In a dynamical plane fractal, we can do the same actions as above (clicking or entering numbers) to generate an orbit. The orbit will show up on the screen as a red line, and be logged to the terminal. It should be noted that *currently* this orbit is calculated by simply entering the number as the "z" value in our script (either the original or with the new fixed "c"), and running the same fractal function. Thus, it will use the same condition as the fractal. In practice, this means that the orbit will be 4 iterations in length if that point took 4 iterations to meet the condition. See the below example of our previous dynamical plane fractal with an orbit for .5+.5i generated and drawn.



We can also click and drag these orbits. If that is done, the orbit will no longer be outputted to the terminal. In

addition, if we are dragging an orbit, the “back” button will go back to the drawn dynamical plane with no orbits instead of the last drawn orbit which is the functionality for clicked orbits. If we want to hide the dynamical plane fractal when dragging orbits (a useful feature in teaching complex dynamics), we can click “Hide Fractal on Orbit Drag“. Below is an example of this. Notice that the complex number orbit *seed* can be seen both below the fractal, and in the box for generated orbits.



The other main features of FractalVoyager lie within editing the options to update the fractal image. These options are:

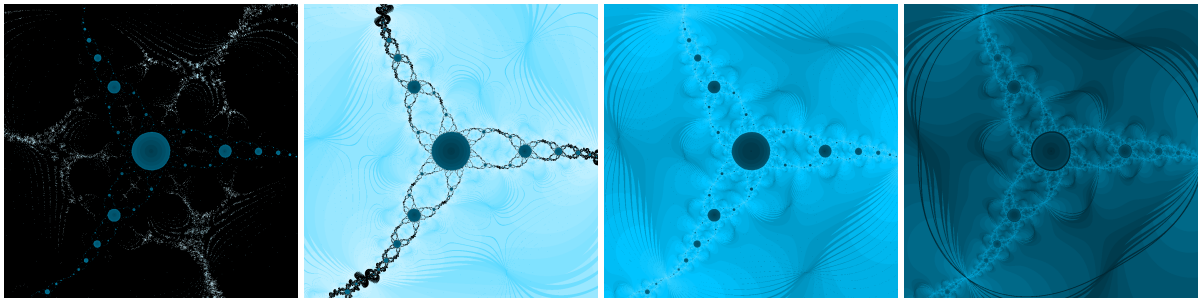
- Max radius
- Epsilon
- Max iterations
- Orbit iterations
- Orbit color
- Min and max values of the axes
- Imaginary axis resolution
- Number of colors
- Color

Before discussing individual options, let's discuss the high level overview of all them. They have default values which are pre-populated in the app and are used in generation of the fractal. If we wish to edit any of the values, we can change them in the boxes/slider. Once they are changed, the "Update" button can be clicked, and once clicked, the fractal is updated with the new options.

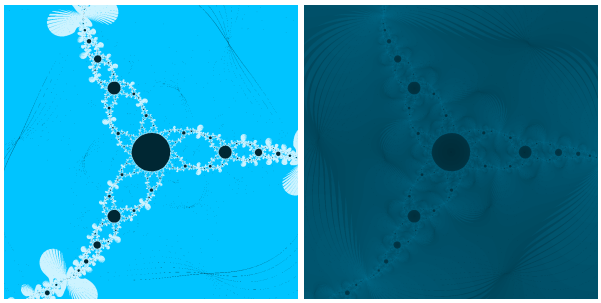
We already discussed the min and max values of the axes, so let's begin with the effect of the "Imaginary Axis Resolution". This is the number of pixels along the imaginary axis (vertical), and will automatically determine the number of pixels along the horizontal axis based on the aspect ratio. For example, if we have 2160 pixels for the value and the ratio is 1:1, the real axis will also have 2160 pixels. If the aspect ratio is 2:1, we will have 4,320 pixels along the horizontal axis if our imaginary axis value is still 2160. In practice, this increases or decreases the quality of the fractal image.

The max and min radius, epsilon, and max iterations values effect the "accuracy" of the fractal. These are the parameters that will be passed into the fractal generation function as discussed in subsections 3.1 and 2.2. Recall, the max radius is the value at which a variable is consider to have "escaped" (or diverged to infinity), min radius is the value at which a variable is consider to have "vanished" (or converged to zero), and epsilon is the minimum difference between two numbers to consider them the same which is used in the "stops" command. Thus, max radius only has an effect in an "escapes" script, min radius only has an effect in a "vanishes" script, and epsilon only has an effect in a "stops" script.

In practice, increasing the accuracy of these parameters effects the coloring. For example, in our `iterate z^2+c until z escapes` script, increasing the value of max radius will make the fractal have more light colored areas than before. This is because points which took, ex. 4 iterations before, to escape, now still take 4 but this is a smaller ratio of the maximum iterations so will be colored lighter. Changing the epsilon value can be particularly useful as it often can reveal interior coloring. For example, consider the dynamical plane fractal below generated with `par set z to 0.62996 + 1.0911i. iterate c^2*z^2/(z^3-1)until z stops.`, a "c" value of -1.1032863849765258-1.6150234741784038i, and epsilon values of 0.0000000000000001, 0.000000000001, 0.000001, and 0.01, respectively (from left to right, top to bottom). Notice that as we decrease the accuracy, we reveal interior coloring that gives insight into the nature of the function by showing patterns that points stop iterating at the same speed along.



The number of colors option increases the amount of colors used, which can also help reveal some internal coloring, but does not have as much of an effect changing the epsilon, max radius, and min radius values. It should be noted that if the number of colors used is over the max iterations, the max iterations will actually be the number of colors. To see an example of how tweaking the number of colors effects the fractal, consider the below two images from our example above with an epsilon of 0.000001 and with 4 colors and 200 colors, respectively.



We can also change the color used by sliding the color slider then clicking update, along with changing the orbit color and number of iterations, with the relevant options. Another import aspect of the application is the back button. This will re-generated the previously drawn fractal or orbit; no matter if options changed, there was a box zoom, we switched to dynamical plane, or we switched to orbit. If the back button is clicked, whatever was on the fractal viewer one time before it was clicked we be re-generated and displayed.

This concludes our walk-through of using the application, but we encourage users to "play around" with the application themselves to best understand it. Also, refer to previous sections, particularly 3.4, 3.1, and 2.2 to gain further insight into the features of the app and their effects.

5 Evaluation

Below are some example scripts that have been tested in FractalVoyager.

- Parameter Plane
  - `iterate z^2+c until z escapes. iterate z^2+c until z escapes.…`
  - `par set z to 0.62996 + 1.0911i. iterate c^2*z^2/(z^3-1)until z stops. par set z to 0.62996 + 1.0911i. iterate c^2*z^3/(z^4-1)until z stops. …`
  - `set a to -1/2 + c. set b to -1/2 -c. iterate z - (z-1)*(z-a)*(z-b)/ ((z-a)* (z-b)+ (z-1)* (z-b)+ (z-1)* (z-a))until z stops.`
- Dynamical Plane - Newtons Method

- `iterate z - (z^3 - 1)/ (3 * z^2)until z stops. iterate z - (z^4 - 1)/ (4 * z^3)until z stops.` …
- `iterate z -.5*( (z^3 - 1)/ (3 * z^2))until z stops.` (changed value of a)
- `iterate z - ((z^3-2*z+2)/(3*z^2-z))until z stops. iterate z - ((z^4-2*z+2)/(4*z^3-z))until z stops.` …
- `iterate z - ((z^6+z^3-1)/(6*z^5+3*z^2))until z stops. iterate z - ((z^5+z^3-1)/(5*z^4+3*z^2))until z stops.` …
- `iterate z - (sin(z)/cos(z))until z stops.`

6 Future Work

Fractal Voyager is currently at a state in which the application is useable and mathematicians wishing to study complex dynamics can use it to visualize and interact with parameter and dynamical plane fractals and orbits. However, there are many improvements needed to make the app more robust, and many additions which would increase the scope of research that could be achieved with the application. To begin, lets discuss the current issues with the application that should be addressed to improve user experience. One bug is the panning and zooming buttons are somewhat unfinished. Currently, the panning buttons function well when only exploring a fractal with the panning and zooming buttons. However, if a user does a box zoom, the panning buttons do not function properly. This is because the state management hasn't fully been handled. The state used for the axes' min and max values, along with the initial work done with the panning and zooming, should be a sufficient guide to handle this state. Along the lines of these buttons, the zooming buttons do not function properly if the user is zooming anywhere not centered at the origin. This is due to an error with the equation used to calculate the new starts and scales, which will be able to be solved after some time considering how the app handles box zooms and pans.

Another buggy section of the app is the terminal output. The terminal output does not always behave the same when it should. For example, we get a "generating fractal..." when there is a box zoom, but not when the fractal axes are changed. The main issue with these terminal outputs are the functions to write to them are asynchronous, but behave oddly sometimes. We haven't had the time to figure the intricacies of building out this terminal; to do so, one should investigate Zustand and React rendering further. Another small bug is that when a user clicks down outside of the fractal, then releases in the fractal, it still registers as a click on the fractal. This is because we are using "mouseUp" events, and do not limit the "mouseDown" events to inside the viewer component. There are some other bugs which are mainly corner cases of ways the user can edit the fractal. Overall, FractalVoyager should be thoroughly bug tested and refactored.

Along with fixing bugs, the styling of the application should be improved. Currently, it is set up to push down the terminal component when the options overflow the page. Instead, when the options are going to overflow its parent container, the options should collapse into a modal or something similar. The other issue with styling is with the options and header. They should expand and contract with screen size, but do not. One issue which is not exactly a bug but should be addressed is with screen resizing. Currently, the styling of the viewer component and the options component works well, but gets messed up when a user's screen is resized. This fixes itself when the user clicks something like "Show/Hide Complex Number" which causes a rerender and updates the styling. There should be a hook or function implemented which triggers on screen resize and forces a rerender.

Fractal Voyager's current handling of orbits needs improvement. Once an orbit is drawn on the screen, the zooming, panning, and editing options no longer works. At the very least, these options should be not allowed once an orbit is drawn. The current issue is that the orbit and image data is stored in the same variable, so, once an orbit is drawn, the last draw fractal simply sticks around on the screen. Any updates will go towards to the parameters used to generate the orbit, which do not effect the image. A solution to this issue would be to hold the two types of variables separately, but this would increase the complexity of state in the application. Another aspect of the orbit that needs to be improved, is adding the functionality for the orbit iterations option (currently unimplemented). This can be done simply by adding it as a parameter into the cgened orbit function just like max iterations or any other parameter is.

The above discussion covers the majority of the immediate work which should be done to improve the user experience of the application, but as stated, Fractal Voyager should be fully bug tested as we most likely missed some issues, and didn't mention a few small ones. One small issue is user actions start to not work after the app has been doing lots of actions for a while. This is most likely because cleanup functions should be implemented on the useEffects. useEffects can return cleanup actions, and FractaVoyager doesn't currently have these implemented. We are not sure if this will fix the issue, but is certainly a good start to investigate this odd behavior.

Once a sufficient amount of the above issues have been fixed, or possibly before, Fractal Voyager should be hosted on the internet. This can be done through GitHub actions, and hosted on GitHub pages. One current issue with this is that the wasm files which handle the cdScript to C++ compilation live in the "src" folder of the React app. Thus, when we run a production build of the app (npm run build), we get an error message. This error message could also be due to the dynamic import for the emception virtual files. The wasm ".mjs" files being in the src folder does, however, cause an issue which is when the files are freshly built: "npm start" fails on first try. It will work on second try though.

Now that the issues with the app have been discussed, let's consider additions to the already implemented sections of FractalVoyager that would improve user experience. Mainly, the language implementation should be updated to allow for better error checking. Currently, if ANTLR is passed a malformed script, we simply get an error message

in the terminal component with no hints to the user on what is incorrect. ANTLR, by default, gives some hints to what the error is in the terminal, and there are certainly ways to use ANTLR for more advanced error handling. In addition to error handling, there are many cases where users can enter a valid script that doesn't make sense. There should be an additional error checking phase, almost like type checking, that will make sure the script makes sense to generate a fractal. There are also cases where our C++ code generation will output C++ code that is malformed, but a valid script was passed in. Here, we get emception's error output to the terminal, which will tell a user what is wrong with the C++, but the user has no way of editing this. Either, the code generation should be improved, or a code editor could be implemented which would allow users to edit the C++ code after it is generated. In summary, the code generation should be improved to be more robust, along with the error checking. Eventually, there could even be syntax highlighting for cdScript.

The cdScript language is not quite to the level of supporting all the features of the original FractalStream language. One requested feature that exists in FractalStream, is the ability for the language to automatically detect cycles, and color accordingly. For example, if a certain value of "z" causes an orbit to get caught on a 3-cycle (repeating the same points), this could be considering "stopping" if allowed by the user. Thus, the point for "z" would get colored black (or another color if requested), if "stops" is the command used. Another requested feature is to allow for multiple colors. This could be built into the UI and/or the language, with conditions on, say "z", that determine the coloring. For example, if our condition is "z stops", we could have additional conditions such as if z ends up being 1, we color that point along a red color gradient. FractalStream also has the ability to detect fixed points that z is going towards, and color the fractal like the above without the need for user commands. These would additions to the language grammar, language compilation, and UI. Using our current method for the stops command and how we pass colors between JS and wasm is a good start to implementing these.

The additions of the above features would create an extremely useful application that is very similar to FractalStream, but on the internet. In addition to these features, there are many other uses for FractalVoyager that could increase its scope. For example, there has been interest in a feature that would allow users to draw a line across the canvas in a parameter plane, and get many Julia set fractals out of it. This would allow users to do 3D printings of Julia sets along a line, which could produce interesting results. We are sure there are many other uses for this application and believe this initial development has laid the groundwork for it to exist on the web.

7 Conclusions

This report has shown how we implemented a fractal generating web application which uses a complex dynamics scripting language (cdScript), along with providing the necessary background to understand this implementation. There are two distinct types of fractals that our application generates: parameter plane fractals and dynamical plane

fractals. Parameter plane fractal generating functions include two parameters, “z” and “c”; “z” is iterated starting at a critical point, and “c” changes with respect to the complex plane. Dynamical plane fractals only include one parameter, “z”, which changes with respect to the complex plane and is iterated. We can generate fractals based on these functions by determining if values across the complex plane, when included in a function, causes the iterated parameter to eventually meet a particular condition. We color each point black if the condition is never met, or along a color gradient based on how quickly the condition is met. Scripts in cdScript, at minimum, include a function to iterate and a condition to generate fractals.

FractalVoyager is implemented as a React application that uses wasm, HTML canvas, and ANTLR. ANTLR is used to create a code generator which takes cdScripts and converts them to C++ code which can generate a fractal based on the script. This process is compiled to wasm and used within the React application. Then, this C++ code is compiled to wasm in the browser with a version of Emscripten compiled with wasm, called emcpetion. The newly generated wasm code to generate fractals is called from React, and React receivers an image data array representiting the fractal which it paints to the HTML canvas. Users may also generate orbit’s for points in a dynamical plane fractal, and can also generate dynamical plane fractals from the parameter plane by fixing a value of “c”. The parameters used to generate the fractal may also be altered to obtain a slightly different fractal, and all of these actions do not require a recompile of the script to wasm. FractalVoyager, as it stands, is a usable application that lays the groundwork for additional development to create a powerful complex dynamics fractal application on the web.

8 Acknowledgements

I first want to thank my faculty advisor Kevin Angstadt for the tremendous amount of work, help, and patience throughout the process of building this application. His immense computer science knowledge and skills were vital to the success of this project. Also, I want to thank Dan Look for the inspiration of this project, and answering countless questions about complex dynamics.

References

[1] Kevin Angstadt. *Programming Languages Class (CS 364)*. `https://myslu.stlawu.edu/~kangstadt/teaching/spring21/364/`. St. Lawrence University, Spring 2021.

[2] baeldung. *Methods of Depth First Traversal and Their Applications*. `https://www.baeldung.com/cs/depth-first-traversal-methods`.

[3] Ruslan’s Blog. *Let’s Build A Simple Interpreter. Part 7: Abstract Syntax Trees*. `https://ruslanspivak.com/lsbasi-part7/`.

[4] Paul Bourke. *Newton Raphson Fractals*. http://paulbourke.net/fractals/newtonraphson/. 2019.

[5] Dakota Bryan. *Fractal Voyager*. https://github.com/FractalVoyager/FractalVoyager. 2023.

[6] UC Berkeley EECS Dept. *Complex Plane Image*. https://ptolemy.berkeley.edu/eecs20/sidebars/complex/polar.html.

[7] mdn web docs. *Using the Document Object Model*. https://developer.mozilla.org/en-US/docs/Web/API/Document_object_model/Using_the_Document_Object_Model.

[8] Dan Look. Personal Conversation. May 2023.

[9] nLab. *Simple Mandelbrot Image*. https://ncatlab.org/nlab/show/Mandelbrot+set.

[10] M. Noonan. *Fractal Stream*. http://pi.math.cornell.edu/~noonan/fstream.html. Cornell University, 2008.

[11] Karl Papadantonakis. *Fractal Asm*. http://pi.math.cornell.edu/~dynamics/FA/index.html. Cornell University, 1999.

[12] Terence Parr. *antlr4*. https://github.com/antlr/antlr4/tree/master. 2023.

[13] Poimandres. *Zustand*. https://github.com/pmndrs/zustand. 2023.

[14] Jorge Prendes. *Emception*. https://github.com/jprendes/emception. 2023.

[15] react-bootstrap. *React-Bootstrap*. https://github.com/react-bootstrap/react-bootstrap. 2023.

[16] Arthur Sonzogni. *ANTLR Cmake Starter*. https://github.com/ArthurSonzogni/ANTLR-cmake-Emscripten-starter. 2020.

A Source Code

Full source code at the time of this writing is available at: https://github.com/FractalVoyager/FractalVoyager commit d24dec98854fea132d13ccf1a776287d75c31746

B cdScript Grammar

```
1 grammar Fractal;
2 /* Parser Rules */
3 script: (command '.')+;
4 command: 'set' variable 'to' expression #SET_TO_COM |
5         ('block' | ':') (command)+ 'end' #BLOCK_COM |
```

```
6          'par' command #PAR_COM |
7          'dyn' command #DYN_COM |
8          if_then #IF_THEN_COM |
9          loop #LOOP_COM |
10        ;
11 loop : 'do' (command)+ 'until' condition #LoopDo |
12         ITERATE expression 'on' variable 'until' condition #LoopIterateOn |
13         ITERATE expression 'until' condition #LoopIterateEmpty |
14         'repeat' n 'times' command #LoopRepeat;
15 expression: (PLUS | MINUS)? atom #SIGNED_ATOM_EXP |
16             expression POW n #POW_EXP |
17             expression TIMES expression #TIMES_EXP |
18             expression DIVIDE expression #DIVIDE_EXP |
19             left=expression PLUS right=expression #PLUS_EXP |
20             expression MINUS expression #MINUS_EXP |
21             cpx_function LPAREN expression RPAREN #CPX_FCN_EXP|
22             LPAREN expression RPAREN #PAREN_EXP
23             ;
24 condition : expression (GT | LT | GT EQUALS | LT EQUALS | EQUALS) expression
       #COMP_COND |
25             expression 'escapes' #ESCAPES_COND |
26             expression 'vanishes' #VANISHES_COND |
27             expression (STOPS) #STOPS_COND |
28             condition (OR | AND | XOR) condition #COMB_COND
29             ;
30 if_then : 'if' condition 'then' command #IF_THEN |
31           'if' condition 'then' command 'else' command #IF_THEN_ELSE
32           ;
33 atom : constant | variable ;
34 constant: cpx_number_re |
35           cpx_number_im |
36           n ;
```

```
37 variable : VARIABLE ;
38 n : POS_INT ;
39 cpx_number_re : NUMBER ;
40 cpx_number_im : CPX_NUMBER_IM ;
41 cpx_function : EXP | COS | SIN | TAN | COSH | SINH | TANH | RE | IM | BAR |
       ARG | LOG | SQRT ;
42 /* Lexer Rules */
43 POS_INT : ('1'..'9') ('0'..'9')* ;
44 NUMBER : '.' ('0' .. '9')+ | ('0' .. '9')+ ('.' ('0' .. '9') +)?;
45 CPX_NUMBER_IM: NUMBER 'i' | 'i';
46 EXP : 'exp' ;
47 COS : 'cos' ;
48 SIN : 'sin' ;
49 TAN : 'tan' ;
50 COSH : 'cosh' ;
51 SINH : 'sinh' ;
52 TANH : 'tanh' ;
53 RE : 're' ;
54 IM : 'im' ;
55 BAR : 'bar' ;
56 ARG : 'arg' ;
57 SQRT : 'sqrt';
58 POW : '^' ;
59 PLUS : '+' ;
60 MINUS : '-' ;
61 TIMES : '*' ;
62 DIVIDE : '/' ;
63 EQUALS : '=' ;
64 GT : '>' ;
65 LT : '<' ;
66 OR: 'or';
67 AND: 'and';
```

```
68 XOR:  'xor';
69 VARIABLE : ('a' .. 'z')+ ;
70 STOPS: 'stops';
71 ITERATE : 'iterate' ;
72 LPAREN : '(' ;
73 RPAREN : ')' ;
74 WS : [ \t\r\n]+ -> skip ;
75 COMMENT : '//' ~ [\r\n]* -> skip ;
```

C Generated Code for "`iterate z^2 + c until z escapes.`"

```
1  #include <math.h>
2  #include <cmath>
3  #include <stdint.h>
4  #include <complex.h>
5  #include<stdio.h>
6  #include <emscripten/emscripten.h>
7  int calcPixel(double z_re, double z_im, double c_re, double c_im, int
       maxIters,
8  double minRadius, double maxRadius, int type, double epsilon);
9  int getIdx(int x, int y, int width, int color);
10 int getIdx(int x, int y, int width, int color){
11 int red = y * (width * 4) + x * 4;
12 return red + color;
13 }
14 int calcPixel(double z_re, double z_im, double c_re, double c_im, int
       maxIters,
15 double minRadius, double maxRadius, int type, double epsilon) {
16 std::complex<double> z(z_re, z_im);
17 std::complex<double> c(c_re,c_im);
18 for(int i = 1; i < maxIters; i++) {
19 z = z*z+c;
```

```
20 if(abs(z) > maxRadius) {
21 return i;
22 }
23 }
24 return 0;
25 }
26 extern "C" {
27 EMSCRIPTEN_KEEPALIVE void genPixles(int type, double fixed_re, double
       fixed_im,
28 int maxIters, double epsilon, double minRadius, double maxRadius, double
       startX,
29 double startY, double newCanWidth, double newCanHeight, int width, int height,
30 double widthScale, double heightScale, uint8_t *ptr, int numColors,
31 uint8_t *redPtr, uint8_t *greenPtr, uint8_t *bluePtr)
32 {
33 for (int x = 0; x < floor(newCanWidth); x++){
34 for (int y = 0; y < floor(newCanHeight); y++){
35  double screen_re = (((widthScale * x) + startX) - width / 2.) / (width  /2.);
36 double screen_im = -(((heightScale * y) + startY) - height /2.) / (height
       /2.);
37 int iterations;
38 if(type == 0) {
39 iterations = calcPixel(0.,0.,screen_re,screen_im, maxIters, minRadius,
       maxRadius, type, epsilon);
40 } else if(type == 1) {
41 iterations = calcPixel(screen_re, screen_im, fixed_re, fixed_im,
42 maxIters, minRadius, maxRadius, type, epsilon);
43 }
44 int color = ceil((double)iterations*numColors/maxIters);
45 ptr[getIdx(x, y, width, 0)] = redPtr[color];
46 ptr[getIdx(x, y, width, 1)] = greenPtr[color];
47 ptr[getIdx(x, y, width, 2)] = bluePtr[color];
```

```
48 ptr[getIdx(x, y, width, 3)] = 255;
49 }
50 }
51 }
52 EMSCRIPTEN_KEEPALIVE void orbit(double fixed_re, double fixed_im,
53 double clicked_re, double clicked_im, int maxIters, double minRadius,
54 double maxRadius, double_t *ptr, double epsilon, int orbitNum){
55 std::complex<double> z(clicked_re, clicked_im);
56 std::complex<double> c(fixed_re,fixed_im);
57 for(int i = 1; i < maxIters; i++) {
58 ptr[i*2-2] = real(z);
59 ptr[i*2-1] = imag(z);
60 z = z*z+c;
61 if(abs(z) > maxRadius) {
62 break;
63 }
64 }
65 }
66 }
```