Chloé Reddy

MUMT 303, New Media Production 2

Professor Yaolong Ju and Professor Travis West

April 6 , 2020

An Exploration Into the Hopalong Fractal

Introduction

In this paper, I will discuss my exploration of the Hopalong Fractal and how it can be represented not only visually, but also musically. The visual portions of this project were implemented in Processing, while all audio was produced in MAX. The communication between the two was done using Open Sound Control (OSC).

Background Research

Fractals are patterns and shapes that have the property of self-similarity. They are made up of infinite versions of themselves with the basic building block being made up of smaller versions of itself. It is most commonly found in mathematics, mainly in geometry and algebra, but it exists naturally in the physical world. Some examples of common fractals are the Koch Snowflake, the Sierpinski Triangle, and the Mandelbrot Set (e.g. see *Fig. 1*). Examples in nature include snowflakes, river deltas, and leaves (e.g. see *Fig. 2*).

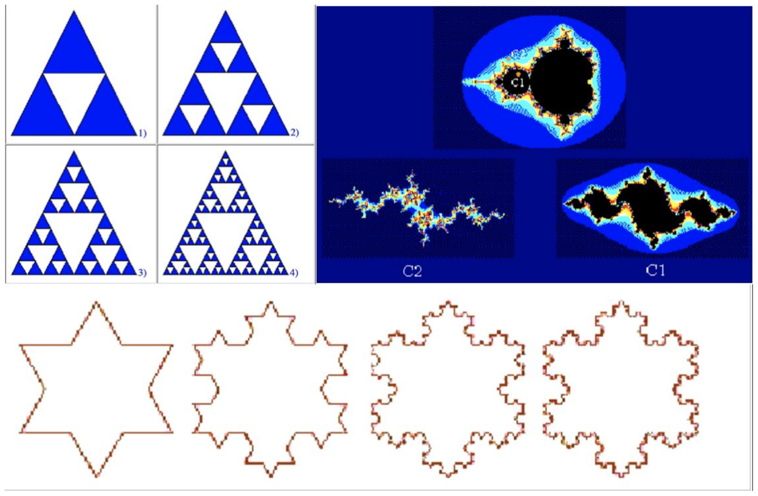


Figure 1. Koch Snowflake (bottom), Sierpinski Triangle (top-left), and the Mandelbrot Set (top-right). *("Fractal")*

Figure 2. Fractals in nature. Snowflake (left), River Deltas (middle), and leaves (right). *("17 Captivating Fractals Found In Nature")*

Fractals are almost always represented visually. It is easy to conceptualize them in this familiar form because that is how they exist in the physical world. But many musicians and mathematicians have found ways to translate fractals into music. One example comes from musician and youtuber Adam Neely, who took a rhythmic approach, using the first four measures of Smash Mouth's "All Star". Neely uses the fact that rhythms played very quickly can be perceived as notes to synthesize the excerpt using only samples of itself.

Another, more melodic, approach comes from Ville Pulkki, who created a musical representation of the Koch Snowflake, entitled "Kuusi (Soundflake)". Each iteration of the fractal is given a different orchestral voice whose note is based on the height of that iteration in time.

Perhaps the most comprehensive implementation comes from Spanish pianist and composer, Gustavo Díaz-Jerez, who created a program called FractMus. The program allows users to select from eleven different fractals and fifteen predefined scales and it algorithmically generates a piece of music based on the chosen parameters. The piece can then be played back with up to sixteen voices.

The Hopalong Fractal

After researching many different types of fractals, I settled on the Hopalong Fractal to use for this project. I found it the most interesting, because it is a family of fractals, rather than a fixed shape. The Hopalong Fractal is defined as an infinite sequence of cartesian points generated by the equation:

$$x_{n+1} = y_n - sign(x_n) * \sqrt{(|b*x_n - c|)}$$

$$y_{n+1} = a - x_n$$

The variables *a*, *b*, and *c* can be any real numbers and the function *sign(x)* returns 1, -1, or 0 depending on the sign of *x*. Depending on the values of these variables, the fractal will appear differently (e.g. see *Fig. 3*). Due to the three inputs to this equation, I thought it would lead to a broader exploration of this fractal, as I could create a program that would produce an output as the *a, b,* and *c* values change.

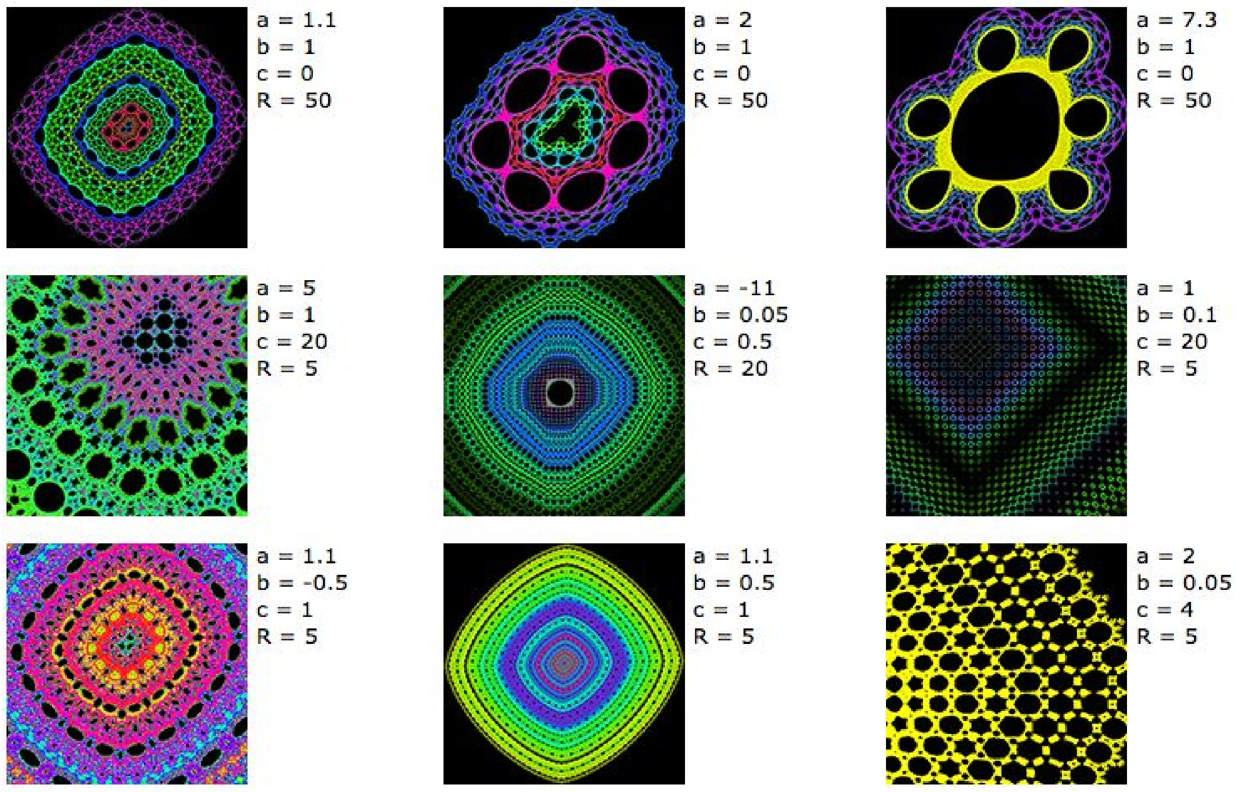


Figure 3. Examples of different Hopalong Fractals. (R represents resolution). *(Lanter)*

Visuals

All the visuals were written in Java and implemented in Processing, which is a software sketchbook mostly used in visual arts and game design. The program starts by randomly generating *a, b,* and *c* floating point values from negative one to positive one. This range was chosen so that, once displayed, the whole fractal would be shown and not just a portion. Then, the *x* and *y* values are iteratively generated for 10,000 points. The points are assigned one of five colours (such that every 2,000 points, the colour changes) and displayed. When new *a, b,* and *c* values are randomly generated, the points then move to their position to display the new fractal.

This is done using what is called the arrival steering behaviour. Points act as agents that have a location, a destination, velocity, acceleration, a maximum speed, and a maximum factor, all defined in the class `aPoint`. Given a new destination, the point will move at no faster than the maximum speed, slowly decelerating to "arrive" at it's point. It never truly arrives at its destination due to the constant deceleration, so there is a check to see if it is close enough. If the magnitude of the vector between the location and the destination is less than 0.01, then the point is said to have arrived. Once all the points have arrived, the fractal is completely displayed. Once *a, b,* and *c* values are updated, then the program runs again.

The program can be run automatically or manually. If the user selects it to be run automatically, when all points have arrived, the program is delayed for 100 milliseconds before new *a, b,* and *c* values are generated. Otherwise, the program will wait for a user input before generating new values.

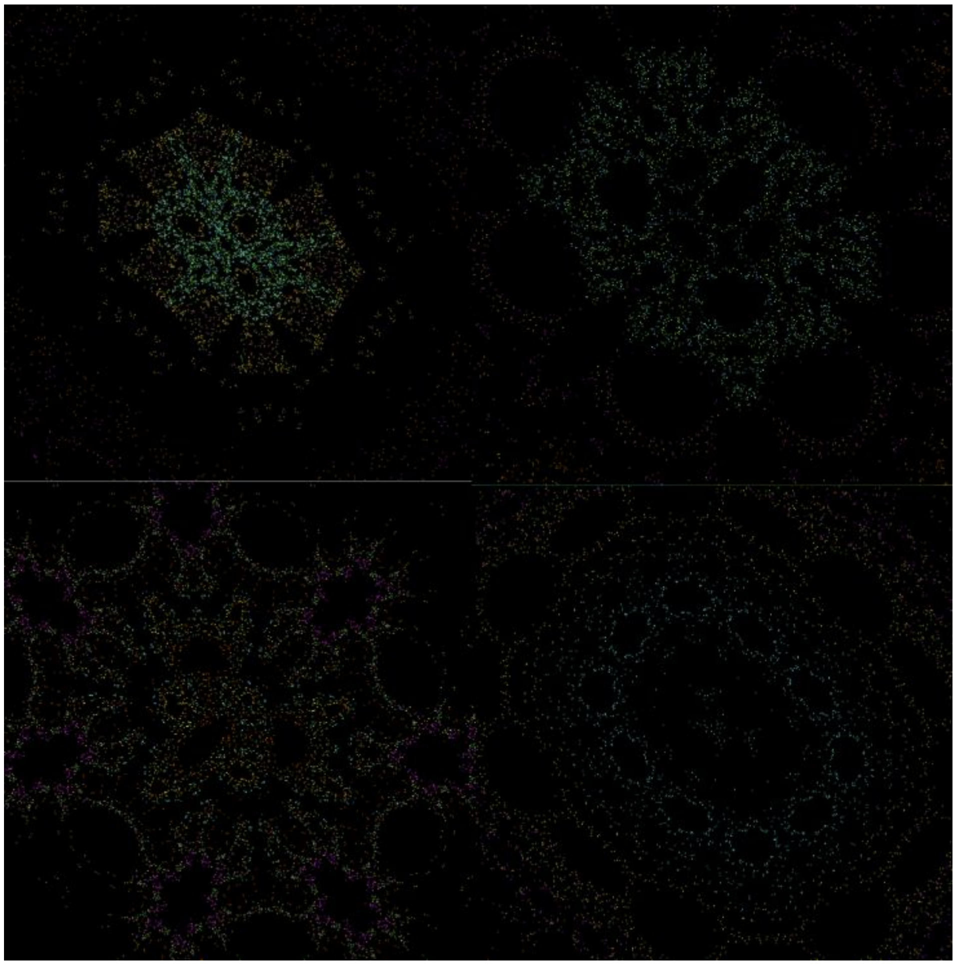


Figure 4. Examples of different Hopalong Fractals randomly generated by the Processing program.

The Patch

The sonification of the fractal was done in MAX. Two different approaches were utilized in the MAX patch. The first was using the *a, b,* and *c,* values that are being randomly generated in the Processing program described above, to create a sawtooth drone (e.g. see *Fig. 4*). Three `saw~` objects create the first three harmonics of the synthesized sound through additive synthesis. The amplitude of each harmonic is controlled by one of *a, b,* and *c*. A user-controlled `kslider` allows the user to select which note they want to hear. To add more complexity to the sound, the drone is passed through a `biquad~` object set to low-pass and whose cutoff frequency is controlled by *a*, gain controlled by *b,* and q-factor controlled by *c*, all scaled to fit an appropriate range for these values. The subpatch `smooth` smooths out the previous value to the current value over seven seconds using a line object to avoid any unpleasant sounds caused by abrupt changes to the values (e.g. see *Fig. 5)*. As the fractal is formed in Processing, the texture of the drone changes as well.

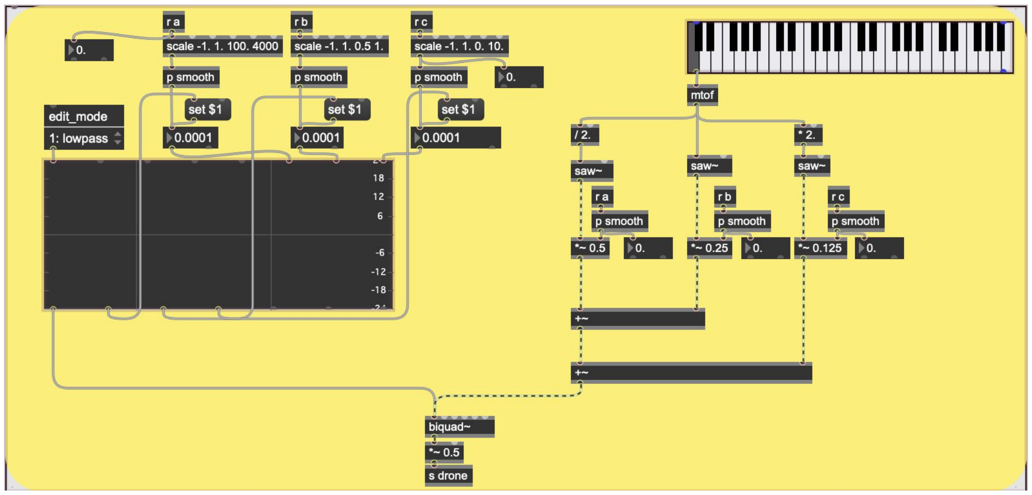


Figure 5. The implementation of the sawtooth drone. The additive synthesis is on the left while the lowpass filter is on the right.

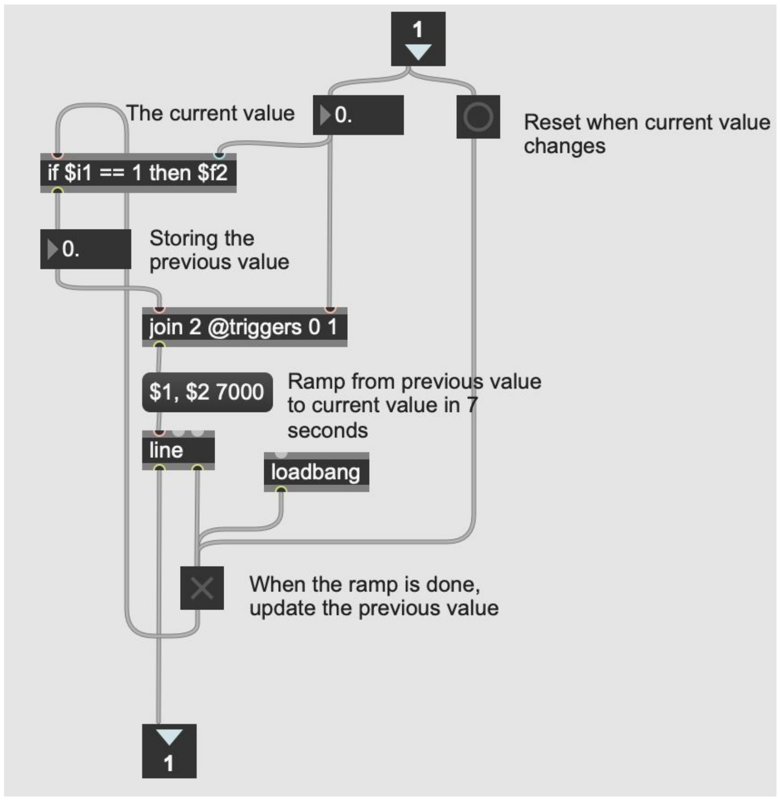
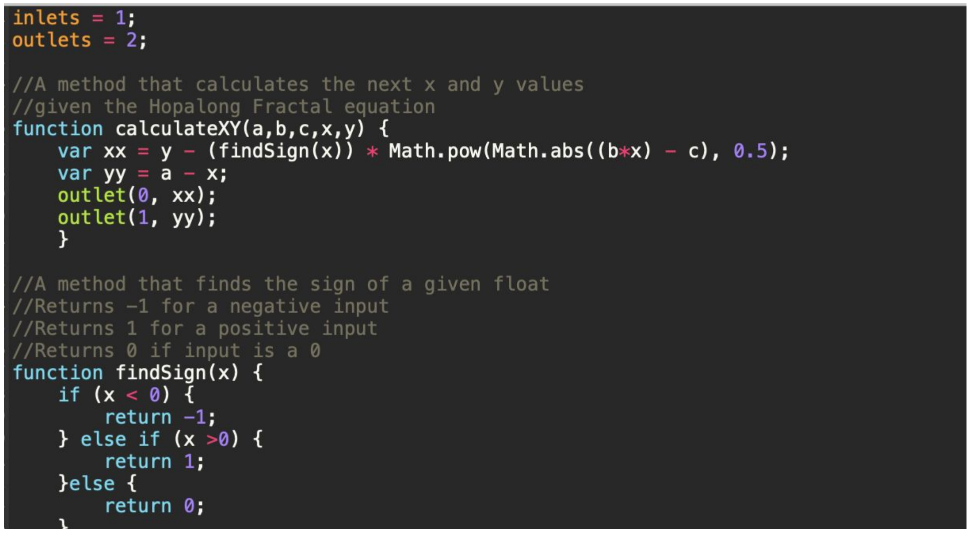

1
The current value
0.
Reset when current value changes
if $i1 == 1 then $f2
0.
Storing the previous value
join 2 @triggers 0 1
$1, $2 7000
Ramp from previous value to current value in 7 seconds
line
loadbang
When the ramp is done, update the previous value
1

Figure 6. The implementation of the `smooth` subpatcher to ramp from the previous value to the current value.

The second approach taken was to utilize the *x* and *y* values to create a melody. Using the *a, b,*and *c* values generated in Processing, *x* and *y* values are recalculated in a javascript script called `calculations.js` in the patch (e.g. see *Fig. 6*). The sound used for the melody was once again created using two triangle waves and additive synthesis in a subpatch called `synthVoice` (e.g. see *Fig. 7*). The user has three options for how they want to hear the melody. The first option has the *x* value as the frequency and *y* as the note duration. The second option has *y* as the frequency and *x* as the note duration. The final option converts the cartesian points into polar coordinates using a `poltocar` object, where *r* becomes the frequency and *θ* is the note duration. Each frequency value is scaled logarithmically using a linedrive object and the absolute value of the note duration is multiplied by 300 to convert into a significant amount of time in milliseconds. Then, using a simple ADSR (attack, delay, sustain, and release) envelope implemented with a `function` object, the note is played. Once a note has finished playing, the next x and y values are generated using `calculations.js`.



```
inlets = 1;
outlets = 2;

//A method that calculates the next x and y values
//given the Hopalong Fractal equation
function calculateXY(a,b,c,x,y) {
    var xx = y - (findSign(x)) * Math.pow(Math.abs((b*x) - c), 0.5);
    var yy = a - x;
    outlet(0, xx);
    outlet(1, yy);
    }

//A method that finds the sign of a given float
//Returns -1 for a negative input
//Returns 1 for a positive input
//Returns 0 if input is a 0
function findSign(x) {
    if (x < 0) {
        return -1;
    } else if (x >0) {
        return 1;
    }else {
        return 0;
    }
```

Figure 7. `Calculations.js`, which returns the next *x* and *y* values based on the Hopalong Fractal equation.

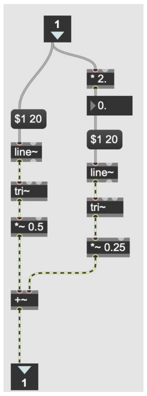


Figure 8. The subpatch `synthVoice` that implements the additive synthesis for the melody voice.

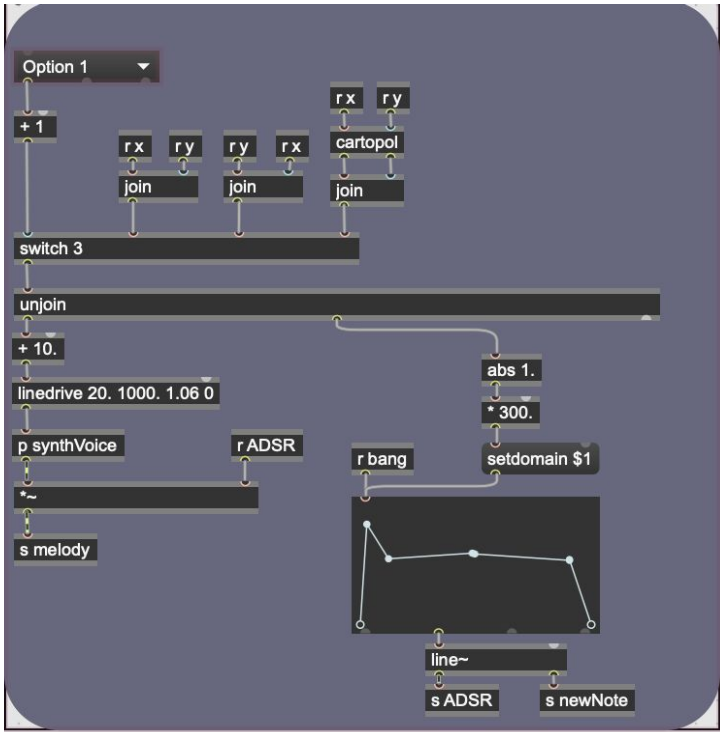


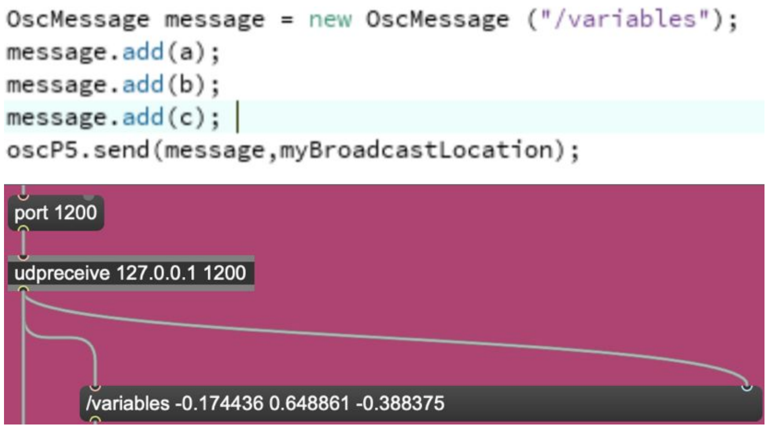
Figure 9. The implementation of the melody in the MAX patch, including the three different options for frequency and note duration as well as the ADSR envelope.

The user can select to hear: just the drone, just the melody, or the drone and melody together. The patch also includes the controls that tell Processing how the user wants to run the program. There is a `trigger` object to select whether the program runs automatically or manually. The default is manually. There is also a `button` that can be pressed to regenerate new *a, b,* and *c,* values, when being run manually.

Connecting MAX and Processing

Once the Processing program is running, it can be controlled by MAX using the Open Sound Control protocol (or OSC). There is a Processing library called oscP5 which can be downloaded and installed to allow OSC to be used in Processing. Using `oscP5.send()` in Processing and `udpreceive` in Max, the randomly generated *a, b,* and *c* values can be sent to MAX through port 1200 (e.g. see *Fig. 10*).

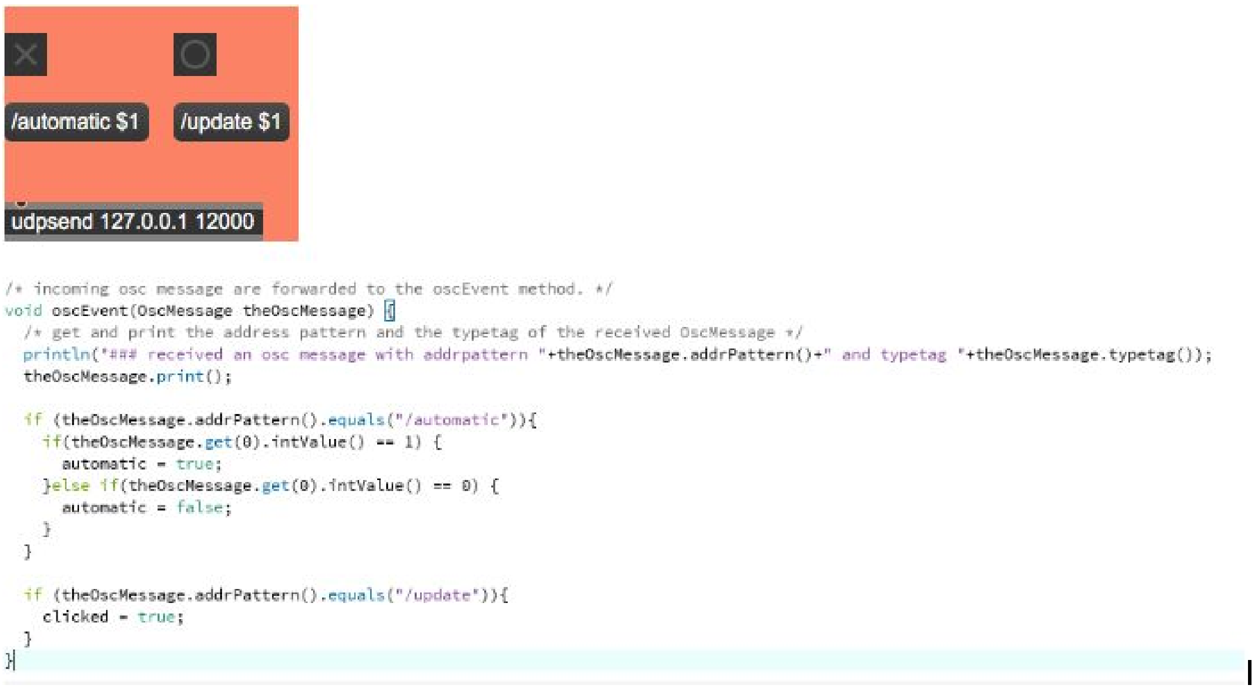
MAX will send one of two OSC messages to Processing through port 12000. The first message defines whether or not the program will run automatically. If the program is running manually, the second message signals Processing to update the values of *a, b,* and *c*. These messages are received by a method called `oscEvent()` in Processing. The global variables `automatic` and `clicked` are updated and the program runs accordingly (e.g. see *Fig. 11*).



```
OscMessage message = new OscMessage ("/variables");
message.add(a);
message.add(b);
message.add(c);
oscP5.send(message,myBroadcastLocation);
```

port 1200
udpreceive 127.0.0.1 1200
/variables -0.174436 0.648861 -0.388375

Figure 10. The Processing (top) to MAX (bottom) OSC message implementation.



```
/* incoming osc message are forwarded to the oscEvent method. */
void oscEvent(OscMessage theOscMessage) {
  /* get and print the address pattern and the typetag of the received OscMessage */
  println("### received an osc message with addrpattern "+theOscMessage.addrPattern()+" and typetag "+theOscMessage.typetag());
  theOscMessage.print();

  if (theOscMessage.addrPattern().equals("/automatic")){
    if(theOscMessage.get(0).intValue() == 1) {
      automatic = true;
    }else if(theOscMessage.get(0).intValue() == 0) {
      automatic = false;
    }
  }

  if (theOscMessage.addrPattern().equals("/update")){
    clicked = true;
  }
}
```

Figure 11. The MAX (top) to Processing (bottom) OSC message implementation.

Observations and Limitations

The Hopalong Fractal is interesting because it is an algebraic fractal, rather than a geometric one, so it was hard to identify obvious patterns that indicate a fractal. But, as the melody is played, a pattern emerges. Furthermore, since the *x* and *y* values do not repeat, it is difficult to say if it is a true pattern or not. The frequencies being played over iterations of the pattern are never the same, but differences between them may be too minute to be perceived. Nonetheless, it is an interesting effect. As for the drone, it is interesting to see how different inputs to the equation drastically changes the sound, but it is hard to draw definitive conclusions as to what sounds are produced by different fractals.

I found that my program is limited in three ways. First, there is an option to play the drone and the melody, but it rarely leads to aesthetically pleasing results. This could have been fixed by tuning the melody to a conventional scale. I chose not to do this since I wanted to preserve the significance and the integrity of the values. It is also difficult to differentiate between melodies when the fractal changes. This could be fixed by changing the voice sound or the frequency range with the fractal changes. Finally, there is no way for the user to control the *a, b,* and *c* values, which means that the fractals are always randomly generated. This leads to less control over how the results can be interpreted. This choice was made in the interest of time.

Conclusions

Throughout this project, the initial goal of exploring how fractals can be represented musically was achieved. I learned through this process that there is no right way to represent fractals sonically because the way the dimensions are defined can be implemented in many different ways. For future developments, different implementations can be added to the program to keep expanding the different ways that we can hear fractals.

Works Cited

"17 Captivating Fractals Found in Nature." *WebEcoist*, 7 Nov. 2016, www.momtastic.com/webecoist/2008/09/07/17-amazing-examples-of-fractals-in-nature/.

Díaz-Jerez, Gustavo. "FractMus 2000." *FractMus*, fractmus.com/.

"Fractal." *Fractal*, 2007, www.cs.mcgill.ca/~rwest/wikispeedia/wpcd/wp/f/Fractal.htm.

Lanter, Martin. "LanterSoft." *Hopalong Fractals*, 2014, www.lantersoft.ch/experiments/hopalong/.

Neely, Adam, director. *Musical Fractals*. *Youtube*, 3 Apr. 2017, www.youtube.com/watch?v=mq0z-sxjNlo.

Pulkki, Ville, director. *Musical Fractal "Kuusi" ("Soundflake"), Instrumental Version*. 28 Mar. 2016, www.youtube.com/watch?v=GVK5N7HQf8Y.