Chloé Reddy

MUMT 303, New Media Production 2

Professor Yaolong Ju and Professor Travis West

February 17, 2020

Polar Function Music Visualizer

Introduction

In this paper I am going to explain and talk about how I created a music visualizer that is rooted in Polar functions to create flowers that dance and react to the music by changing size as the changing colours in response to the music. I will talk about my motivation and the research for the project but also the process that I went through and do a breakdown of the patch and how the audio was analyzed and mapped to visual components.

Inspiration and Motivation

The inspiration of this patch came from two places. The first is a song by Tom Misch and Poppy Ajudha song "Disco Yes". As the title describes, the song is disco inspired, with, strong beats, and a funky groove. The second inspiration was the Mac screensavers that react and change size to the music that plays from the computer. Aesthetic and colour wise, the inspiration came from the disco feels of the Tom Misch song, which led to the use of polar roses to create flowers. Instead of trying to create something that fulfilled the image in my mind, I wanted to create a more objective visualizer and see how different songs would react to it.

Background Research

Polar roses were used to create flowers. Polar roses are sinusoids plotted onto the polar coordinate plane. They are expressed in the polar plane by the equation $r = a \cos (k\ \theta)^b$. K represents the amount of petals. Normally, if k is odd, the rose has k petals and if k is even, the rose has $2k$ petals. In this implementation, the roses are always 2k-petalled(e.g. see *Fig. 1*). The coefficient *a* inversely and proportionally affects the size of the function (e.g. see *Fig. 2*). *B* controls how the width of the petals increases proportionally(e.g. see *Fig. 3*).

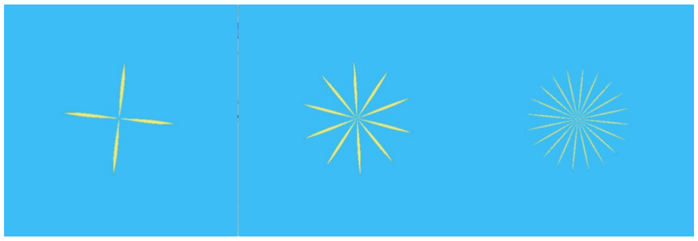

Fig. 1 Roses with an increasing number of petals (from the left: $k = 2$, 5, and 10).

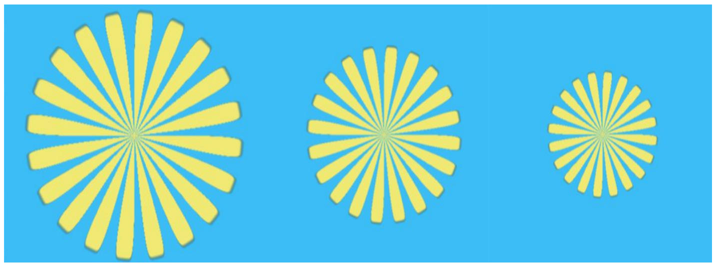

Fig. 2 Roses with increasing a values (from the left: $a = 0.5$, 1, and 2).

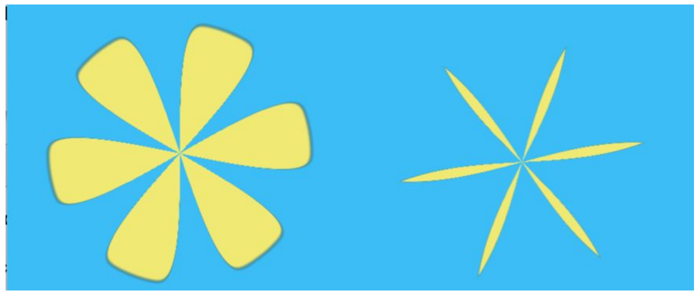

Fig. 3 Two roses with different b values (left: $b = 10$, right: $b = 5$).

Polar roses can also be made using sine instead of cosine in the function. This results in a phase difference of $\pi/2$. When the function includes the sum of a cosine and sine function, the rose starts to have some interesting properties. For example, with the equation $r = a(((\cos (k\,\theta))^b$

+((sin (l θ))^c) + d combines two separate flowers into one (e.g. see *fig. 4*). When the functions overlap, the length of the petals are added together. When they don't overlap, the length of the individual petal stays the same. The constant *d* is the rotation. Adding *d* where $0 \leq d \leq 2\pi$, the flower will rotate to the left by a factor of *d*.

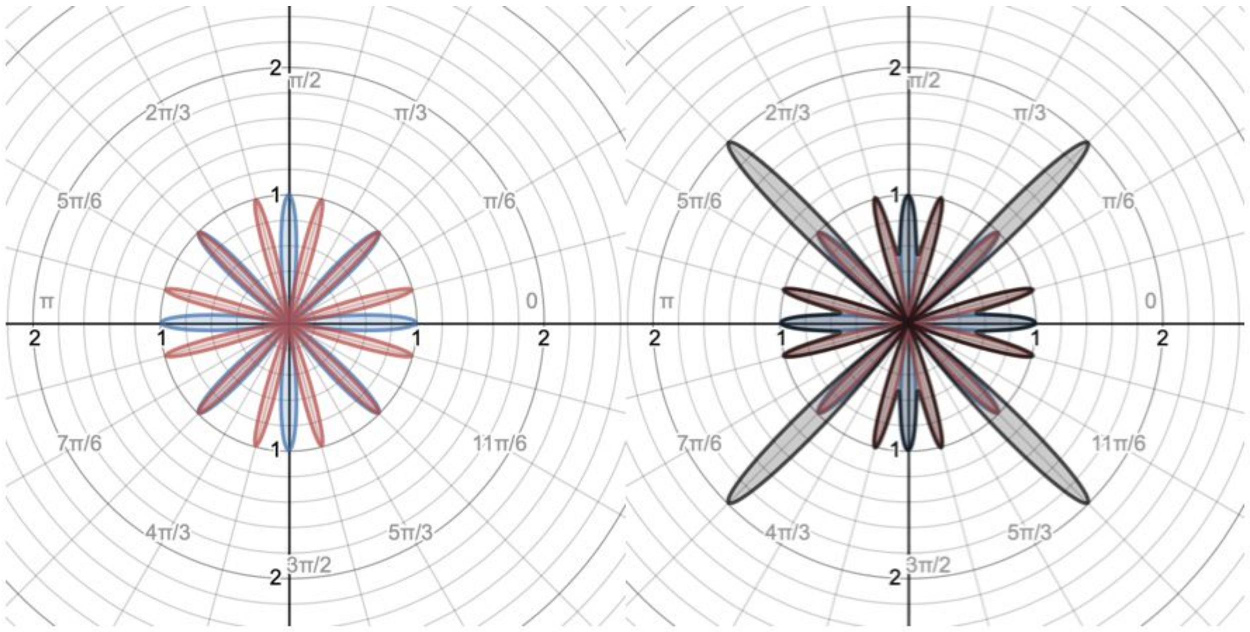

Figure 4. Representation of a cosine (red) and sine (blue) polar functions plotted. The black shows the result of adding the two functions together.

"Desmos-Graph." *Desmos.com*, www.desmos.com/calculator/ucthjyrrbv.

The Patch

The patch can be broken down into 3 parts: the creation of the rose, audio analysis, and scaling/mapping.

The first step was creating the rose itself. This was done using jitter objects in Max. The calculation of the polar roses is done using a `jit.gl.pix` object called `flowers`. There are two of these to create two seperate flowers that layer on top of one another. The `flowers` patch implements the equation $r = a(((\cos (k (\theta+d)))^b +((\sin (l (\theta+d)))^c)$ where θ is calculated taking the arctangent of the x and y components calculated from the signed norm of the indices of the matrix (e.g. see *fig. 4*). There are two jit.matrix objects (one for each flower) called `flower1` and `flower2`. They are 600 x 600 matrices that have four dimensions to contain colour information about each pixel. The calculation is performed on each pixel as well as a reverse subtraction. The value is then multiplied by a colour vector (gien in terms of red, green, and blue). The original colours are hardcoded, but they will be shifted. This is then mixed with another colour vector to colour the background of the image.

To change the other parameters of the flower (i.e. *a, k, b, l, c,* and *d*), a `param` object functions within the patch and is changed outside the patch using a message of the form `param nameOfParameter $1.` This allows any value to be controlled from the patch and the roses to be dynamic.

To mix the two flowers together, another jit.gl.pix object is used. It is also in this object that the colours are shifted, controlled by the parameter `hue_shift`, which is a float from 0 to 1. First, the pixel rgb (red, green, blue) value is converted to an hsl (hue, saturation, lightness), and the hue shift is added. Then the value is converted back to an rgb value.

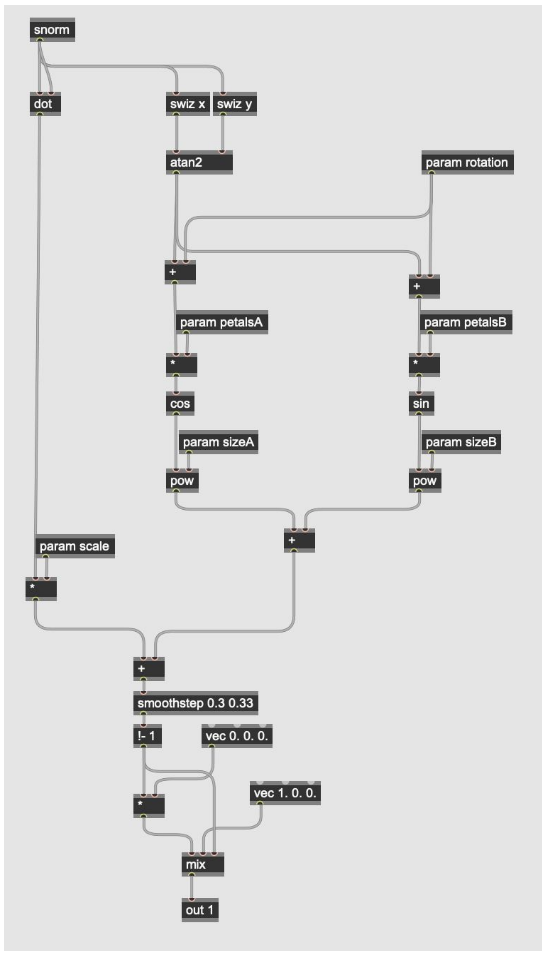

Figure 4. The `jit.gl.pix flowers` object

Finally, the flower has to be rendered. This was done with a `qmetro` object set to have a period of 16.67 milliseconds. Then, using 2 `jit.gl.render` objects, a `jit.gl.videoplane` object, and a `jit.window` of size 600 x 600, all called `flowers`, the image can be displayed in a popup window or in full screen.

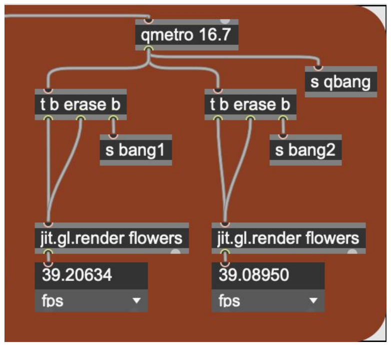

Figure 5. The rendering portion of the patch.

Then comes the audio-processing portion of the patch. A user can choose to upload an audio file or use the open-source Soundflower extension to use the audio that would normally be outputted from their laptop. Created by Matt Ingals, Soundflower can be downloaded and installed by a user and allows them to change the routing of the audio signals. Max can recognize Soundflower as an input and music can then be played from software such as iTunes or Spotify without having to download individual audio files to play on Max.

After the user chooses an audio input, its signal average is analyzed in two ways: with the absolute value and the rms value. The rms evaluation takes the square root of the average value squared. -0.5 is subtracted from the rms evaluation to add a bigger difference between the two averages and then both values are scaled in the patch `scaleAmp` (e.g. see *fig 6*). Using zl.stream and zl.median, it averages the last two values that it receives and then scales it with an inverse relationship. A slide object is used to smooth out the values.

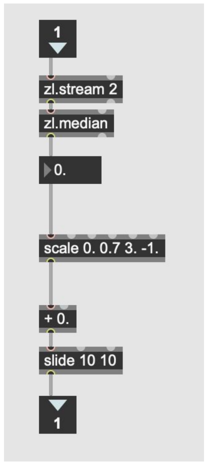

Fig 6. The patcher `scaleAmp`.

The signal is also analyzed to get information about the magnitude of different parts of its spectrum in the patch `Magnitude`. Using the `pfft~` object called `spectrum1` (e.g. see *fig. 7*). This patch takes the signal and, using the fast fourier transform, divides the spectrum into 8 "chunks", where the magnitude of each can be separated. Then, in `Magnitude`, the magnitudes are averaged in pairs (ie.e the two lowest chunks' magnitudes are averaged). This generates four separate magnitude values to use. These values are then scaled using a `zmap` object, which ignores values outside of the set range -127 to -20.

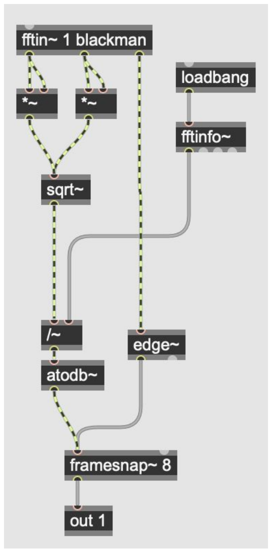

Fig 7. The `pfft~` patcher `spectrum1`.

The last component of the audio signal to be analyzed is the energy, which is done in the patch `energy`. It contains a `pfft~` object also called `energy`, that functions similarly to spectrum1, but it sums all of the magnitudes of the spectral parts to get the total energy of the spectrum. This value is then passed to the patch called hueShift, where it is scaled and smoothed using a `zmap` and a `slide` object, to avoid “jumpy” values.

The number of petals are randomly generated each time a new song starts, or when the amplitude is 0. For aesthetics, the number of petals is between 3 and 15 and both flowers have the same number of petals.

The rotation of the flowers is hardcoded and controlled by a `clocker` object and scaled by .04. One flower rotates clockwise while the other rotates counter-clockwise, again, done for aesthetics.

Once all of these values are calculated, scaled, and smoothed, they can be mapped to different parts of the patch to affect the visuals. Table 1 shows where values are computed from and what they affect in the patch. With all the values either hardcoded, randomly generated, or calculated, the visuals and the music can be turned on, and then the user can enjoy watching the visuals react without having to do anymore work.

Table 1

Information about the mapping of values in the patch.

Name of Attribute in the Patch	$r = a\ ((\cos\ (k(\theta+d)))^{b} + (\sin\ (l(\theta+d))^{c})$	What It Maps To	How It's Generated
petalsA1	*k* (flower1)	The division of petals the cosine function of flower1	Randomly generated
petalsB1	*l* (flower1)	The division of petals the sine function of flower1	Randomly generated
sizeA1	*b* (flower1)	The size of the petals from cosine in flower1	Average of the lowest spectral chunks
sizeB1	*c* (flower1)	The size of petals from sine in flower1	Average of the mid-lowest spectral chunks
Scale1	*a* (flower1)	The size of flower1	Rms average
petalsA2	*k* (flower2)	The division of petals the cosine function of flower1	Randomly generated
petalsB2	*l* (flower2)	The division of petals the sine function of flower2	Randomly generated
sizeA2	*b* (flower2)	The size of the petals from cosine in flower2	Average of the mid-highest spectral chunks

Observations and Limitations

After numerous tests and iterations of this patch, I came to a few conclusions,the first being that it is very jumpy. The amplitude changes a lot and as a result the flower looks jittery and almost broken. But, if the values are smoothed out too much, the reactions to strong beats and musical cues are lost. The patch shoots for a middle ground, so that the flower is still responsive, but not so responsive that it is unpleasant to watch. That being said, the visuals work really well for songs with strong beats, such as rap and dance songs. It works less for songs with little amplitude variation, such as softer folk tunes.

Also, certain petal number combinations work better and are more responsive than others. Since they are randomly generated, it will be different each time a song is played. This could be fixed by hard coding petal number combinations, but would allow for less variability.

This patch has interesting effects when certain numbers are aliased, such as petal numbers, scale, and petal width. But, when the patch switches between aliased and non-aliased values too quickly, it is also uncomfortable to watch. It works for long amounts of time, so it could be possible to implement some sort of system that would allow for a slow movement between aliased and non-aliased values to achieve different effects.

The colours in the patch are also extremely limited. The original colours are hardcoded in the jit.gl.pix patches and then they are just shifted. It could be possible to have individual colours affected by different parts of the spectrum. The problem that arose was that unattractive colour combinations were very quick to come by, so the hardcoded solution seemed to be more aesthetically pleasing.

Conclusion

The goal of creating a music visualizer that could react to music played straight from one's laptops was very much achieved. Though it is limited in its implementation, those limitations come from the need of making sure it is aesthetically pleasing rather than complex but an eyesore. In the future, it could be interesting to add more flowers. The process of creating a flower could be put into a poly~ object and then a user could decide how many flowers they might want to create. Different parts of the spectrum could control different parameters of individual flowers. The flowers could even start to move as they dance around. This patch is a simple starting point for what has the potential to be a complex music visualizer.

Sources

Berman, Shelly, and Jo Ann Fricker. "GSP4 Polar Roses." *LMSTD*, www.lmtsd.org/cms/lib/PA01000427/Centricity/Domain/116/Polar%20Roses.pdf.

Ingalls, Matt. "Mattingalls/Soundflower." *GitHub*, 6 Apr. 2019, github.com/mattingalls/Soundflower.