Experimental Composition Tools Using the DeJong Chaotic Attractor to Manipulate Audio Using an AU Plugin Implementation

ABSTRACT

The DeJong plugin is an AU plugin that can be used in any DAW on Mac OS which applies the DeJong chaotic attractor equation first proposed by Peter DeJong to incoming audio in stereo. The sonic output is captivating, experimental, and diverse. The plugin can be used as easily as any other common effect plugin, such as distortion and equalization, with the ability to automate parameters in real-time.

1. PRIOR WORK

DrawJong 2.0 [1] is a synthesizer and visualizer created by Matthew Hetrick specifically designed to sonify and visualize chaotic attractors. Hetrick's implementation is a stand alone audio application that synthesizes audio based on the DeJong chaotic attractor but also visualizes the equation by plotting points graphically. This implementation does not affect incoming audio, it synthesizes in real time based on the equation. The synthesis is interesting, but I wanted to shape my plugin in a way that would work with audio and preserve the original audio signal in my implementation.

2. DEJONG CHAOTIC ATTRACTOR

The DeJong chaotic attractor is given as[1]:

$$xn = \sin(a * yn-1) - \cos(b * xn-1)$$
$$yn = \sin(c * xn-1) - \cos(d * yn-1)$$

In Hetrick's DrawJong, xn and yn equal the next point to be plotted on the respective axis with xn-1 and yn-1 being the previously plotted points. Variables a, b, c, and d are parameters that can be constants or dynamic, but Hetrick's implementation used these variables as constants. The images created by this equation are truly beautiful and can be seen in Hetrick's paper.

In my interpretation of this equation for audio, xn is equal to the left channel's output and yn equals the right channel's output. xn-1 and yn-1 represent the previous sample in each channel and I used variables a and b in the plugin replacing variable c with variable a and replacing variable d with variable b.


```cpp
for (int sample = 0; sample < buffer.getNumSamples(); ++sample)
{
    if (stereo == true)
    {
    prevX = prevX + *varCParameter + inputX[sample];
    prevY = prevY + *varDParameter + inputY[sample];

    auto newXDeJongX = (sin(varA * prevY) - cos(varB * prevX));
    auto newYDeJongY = (sin(varB * prevX) - cos(varA * prevY));

    if (inputX[sample] == 0)
    {
        outBufferOne[sample] = 0;
    }

    else
    {
    outBufferOne[sample] = ((newXDeJongX) * 0.25f) * gain;
    }

    if (inputY[sample] == 0)
    {
        outBufferTwo[sample] = 0;
    }

    else
    {
    outBufferTwo[sample] = ((newYDeJongY) * 0.25f) * gain;
    }

    // Update prevX and prevY so that the next time this for-loop starts, they will have the previous
    // sample value stored in them.
    prevX = newXDeJongX;
    prevY = newYDeJongY;
    }
    else
    {
        channelData[sample] = 0;
    }
}
```

Figure 1. The process block method in Juce.

The significance of using a chaotic attractor as an audio plugin is creating a new experimental compositional tool that is easy to use inside of any DAW by anyone regardless of how familiar they are with experimental music making techniques.

3. THE JUCE FRAMEWORK AND PLUGIN DESIGN

I used the open source Juce framework with C++ to build both the visual component and dsp component of this plugin. [2] In this framework, Juce includes a handy method that handles all of the plugins dsp called the processBlock which is shown in figure 1. This method iterates through each audio sample coming through the input with a for loop given as:

for (int sample = 0; sample ¡ buffer.getNumSamples(); ++sample).

This is where I was able to apply processing to each sample. Before I implemented the equation, I created a way to dynamically access previous samples from the input. I created two variables named prevX and prevY to store the current sample from the input so that after each iteration, the samples contain in prexV and prevY are what came before each iteration through the for loop. This was done by implementing the code:

prevX = prevX + *varCParameter + inputX[sample];

prevY = prevY + *varDParameter + inputY[sample];.

Adding in the varCParameter and varDParameter, which link to dials on the graphic part of the plugin, allowed me to choose how far back in the prevX and prevY address I can pull from. Finally, I implemented the actual DeJong equation for xn and yn with:

```
auto newXDeJongX = (sin(varA * prevY) - cos(varB * prevX));
auto newYDeJongY = (sin(varB * prevX) - cos(varA * prevY));
```

After each iteration of the for loop, I stored the processed sample in prevX and prevY by implementing:

```
prevX = newXDeJongX;
prevY = newYDeJongY;
```

This is where I am able to pull from previous samples. One difference that one notices in this implementation is that I am using only variables a and b in place of variables c and d from the original DeJong equation. This is because in my testing, I found that having four variables did not make any sonic difference from having only two variables. Figure 2 shows the user interface I created for the plugin.

Figure 2. Current interface for the experimental DeJong distortion plugin.

I named variables a and b as "Color 1" and "Color 2" and prevX and prevY as "Left Portal" and "Right Portal" since they are essentially pulling samples from back in time. Finally, I pass each equation to the left and right output buffer with the code:

```
outBufferOne[sample] = ((newXDeJongX) * 0.25f) * gain;
outBufferTwo[sample] = ((newYDeJongY) * 0.25f) * gain;
```

Each side is multiplied by 0.25 to compensate for the volume increase inherent in using this equation. It is also multiplied by a gain parameter that I have not implemented in this demo. This will serve as an output volume in the finished product.

4. SOUND RESULTS

The sonic effect of this plugin and equation in general is captivating and erratic. Figure 3 shows a comparison between a dry drum loop and the same drum loop passing through the plugin using MaxMSP. [3]

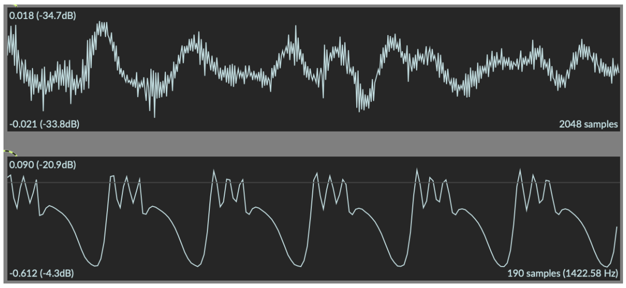

Figure 3. Comparison between a dry drum loop and an affected drum loop.

The top level is the dry drum loop and the bottom level is the affected drum loop. You can see how the dry drum loop looks fairly asymmetrical and dynamic, but the affected drum loop is so distorted that the waveform looks like perfect copies of itself. In terms of timbre, this plugin turns the signal into a thin, bright fuzz that resembles static and feedback. However, changing the color parameters shift the fundamental sound around drastically. The dry signal is still somewhat retained in the affected signal which makes this plugin very dynamic and useful in compositional projects.

5. FUTURE WORK AND DESIGN IMPROVEMENTS

In future conceptions, I would like to add an output volume parameter, connecting to he gain variable, to control the volume of the plugin. I also want to add a wet/dry control to the plugin to change how much of the wet signal and how much of the dry signal passes through to the outputs which is a common control for most effect plugins. [4] I plan on researching new chaotic attractors and fractal maps to create more plugins to use as compositional tools for future projects and research. As it stands, this plugin successfully saves it's state and parameters, is compatible with automation in DAWs, and works successfully with Logic Pro X, [5] Ableton 10, [6] Studio One 4, [7] and MaxMSP [3] on Mac OS.

6. REFERENCES

[1] M. Hetrick, "DrawJong 2.0," 2011.

[2] WeAreRoli/Juce, "Juce Framework," https://github.com/WeAreROLI/JUCE, 2018.

[3] C. 74, "MaxMSP," 2018.

[4] V. Goudard and R. Muller, "Real-time audio plugin architectures," *Comparative study. IRCAM-Centre Pompidou. France*, 2003.

[5] Apple, "Logic Pro X," 2018.

[6] Ableton, "Ableton 10," 2018.

[7] Presonus, "Studio One 4," 2018.