# tSaturator: A Vintage Tape Saturation Plugin

**Deanna Turner**
Northeastern University
Boston, MA
turner.de@northeastern.edu

**ABSTRACT**

tSaturator (pronounced "saturator") is a VST/AU tape saturation plugin built with the JUCE C++ framework. The interface allows users to control the amount of saturation, the drive of the signal applied before saturation, the wet/dry mix, and the amount of hiss. tSaturator also includes a spectrum analyzer, allowing users to see what the plugin is doing to their audio in real time. This project explores creating an effect plugin based on audio antiquing in a modern real-time audio environment. It is a much more simplified version than a full blown tape machine emulation, given the time and resource constraints imposed by such a project. The intent is also to practice good coding practices across a larger-scale project.

## 1. INTRODUCTION

Tape saturation is a type of harmonic distortion that is produced by magnetic tape when the input voltage exceeds the tape's ability to record. The signal gets compressed and distorted non-linearly, and thus odd harmonics are added to the signal. Essentially, this combination of compression and distortion creates a "warm" sound that is characteristic of most analog gear.

### 1.1 History of Distortion and Tape Saturation

*1.1.1 Early years of Distortion*

The use of distortion can be traced back to the early 1950s, but the progenitor of the effect is debated heavily. The first widely accepted use of distortion in a commercial recording is "Rocket 88" by Jackie Brenston and Ike Turner (1951). The story goes that guitarist Willie Kizart poked holes in the speakers of his amplifier, but there are other versions of the story that say he dropped the amp while on tour and that's how the sound was born. Howlin' Wolf's guitarist is also credited with cranking up his Fender Deluxe amp on "How Many More Times" (1951) to the point that it caused the tubes in the amp to be overdriven [1].

From there, further "mistakes" in the studio contributed to the development of distortion in popular music, such as the accidental fuzzed bass tone of "Don't Worry" (1961) which led to Glenn Snoddy's creation of the Maestro Fuzz Tone pedal. Keith Richards' use of the pedal on the Rolling Stones' 1965 hit "Satisfaction" caused sales to soar and another pedal to be born: Tone Bender, an improved version of the Maestro Fuzz Tone with more bass and sustain [1]. Most pedals around this time were still fuzz, until about

the 1970s, when a new form of distortion was created: hard clipping, which used an op-amp. With that, we get pedals like the MXR-Distortion+, Boss DS-1, and more going into the 80s and 90s [2].

*1.1.2 Tape Saturation*

While reel-to-reel tape machines were invented much earlier, they revolutionized the music industry in the 1950s when crooners like Bing Crosby invested in tape machine companies such as Ampex. These earlier tape machines were made with vacuum tubes, but eventually switched to solid state circuits and direct drive motors through the 60s and 70s. Tape machines started phasing out by the 80s with the advent of digital recording technologies such as DAT cassettes and hard disks [3].

The unique thing about tape on reel-to-reel machines is that there's a lot of idiosyncrasies that contribute to their sound: the chemical makeup of the tape, the construction of the head mechanism that feeds the tape through the machine, the thickness, etc. While they can be considered defects, they have been used in artistic ways, and thus, plugins like these were born.

**1.2 Technical Considerations**

*1.2.1 Distortion Transfer Functions*

Tape saturation can be simulated digitally by applying a few different transfer functions to the signal, some to control the distortion, and some to control the compression. The paper by Välimäki et. al on audio antiquing illustrates some transfer functions that can be used to model nonlinear distortion.

$$y = \frac{\tanh(x(n)B_{\text{loud}})}{\tanh(B_{\text{loud}})}$$

**Figure 1.** Nonlinear distortion on loud passages [4]

$$y = \text{sign}(x(n))\text{abs}(x(n))^{B_{\text{soft}}}$$

**Figure 2.** Nonlinear distortion on soft passages [4]

x(n) is the input signal, y(n) is the output signal, and $B_{soft}$ and $B_{loud}$ are set loudness factors (the study in this paper sets $B_{soft}$ to 2 and $B_{loud}$ to 5). These formulas are for modeling general audio antiquing, but are being adapted for usage in creating tape saturation in this project. Special consideration needs to be taken with these functions though, as aliasing will occur if left untreated. To combat this, either oversampling or low pass filtering can be performed on the output signal [4]. This project utilizes oversampling at a rate of 2x to combat the effects of aliasing.

*1.2.2 Oversampling*

According to the Nyquist-Shannon sampling theorem, for a sample rate $f_s$ of 44.1kHz (which is used in the code for this project), we can represent sounds up to $f_s$ / 2, or 22.05 kHz, where 22.05kHz is the Nyquist frequency. When sounds above the Nyquist frequency are recorded, they "fold over" into frequencies we can hear but don't intend to have in the recording, and this is known as aliasing. To combat this, oversampling can be used. Oversampling is a process by which the sample rate is

temporarily increased, increasing the Nyquist frequency and the bandwidth of the signal. Processing runs on the signal as normal (introducing new harmonics), then the signal is low-pass filtered at the original Nyquist frequency, and finally the signal is "downsampled" back to the original sample rate [5].

*1.2.3 Hiss*

Magnetic tape, like most other analog gear, produces a small amount of hiss due to its imperfections. This can be added by observing the dominant frequencies in actual tape hiss, and then filtering white noise to have a similar spectrum with an EQ curve. The best choice is to find the response curve of a specific tape machine, which Jack Endino had plots of various tape machines to choose from.

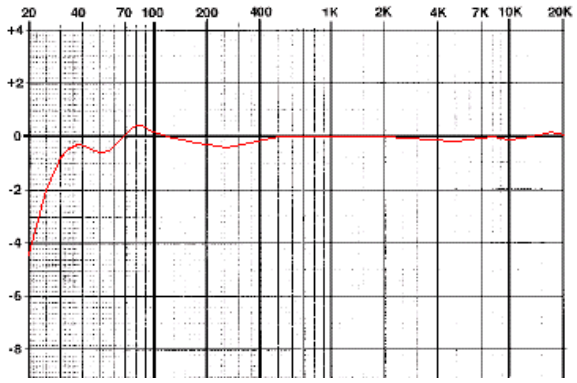


**Figure 3.** Frequency Response Curve of the Ampex ATR-102 half-inch two-track, with extended low frequency heads, 30 IPS [6]

The Ampex ATR-102 was a machine considered to be the basis for modeling the specifics of the saturation for this project, but this would require extensive physical modeling and access to that kind or machine or its manuals, both of which were far beyond the scope of the project. Hence, its frequency response curve was selected to be the basis of the EQ curve for the project's hiss feature to make it somewhat more realistic.

*1.2.4 Wow, Flutter, Bias*

Wow, flutter, and bias are other controls typically part of a tape saturation application used to further enhance the realness of the sound. Wow refers to the slow shifts in pitch usually caused by the varying rotational speed of the tape. Flutter refers to faster fluctuations usually caused by imperfections in the tape itself [4]. Bias is an additional signal, either AC or DC, applied to the tape to improve its fidelity.

**1.3 Study of Current Market Plugins**

Observing plugins that are currently on the market is helpful to informing the design of a new product. The plugins studied for this project were Waves J37-Tape, Kramer Master Tape, RC-20 Retro Color, Lifeline Expanse, Reels Lo-fi Tape, Sketch Cassette II, and Goodhertz Tupe. In this way, the design can be informed by what makes pre-existing products successful, while also making sure to create something that doesn't already exist.

*1.3.1 Waves J37-Tape*

This is a digital emulation of Abbey Road Studios' Studer J37 tape machine, which is an all valve tape machine. There's a lot that can be controlled by this plugin, namely wow, flutter, bias, speed, and the tape formula (that helps shape the characteristic sounds of the machine). These are three tape formulas [7]:

- EMI TAPE 888 (early ‘60s) – Considered “lo-fi” and “grainier” compared to the other formulas, with more distortion between 1 kHz and 8 kHz
- EMI TAPE 811 (mid to late ‘60s) – Offers better high frequency response and slightly less distortion than the 888 formula.
- EMI TAPE 815 (early ‘70s) – Delivers flatter high frequency response and less distortion than the 811. Recommended when minimal coloration is desired.

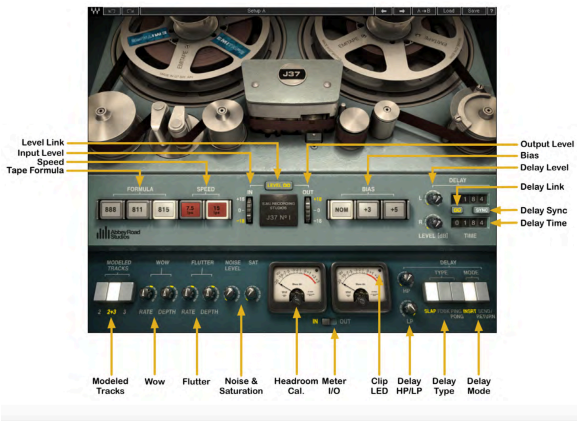


**Figure 4.** Waves J37-Tape UI [7]

*1.3.2 Kramer Master Tape*

The Kramer Master Tape is a combination tube and tape saturation plugin designed with the help of Eddie Kramer, who worked with Led Zeppelin and Jimi Hendrix. This plugin is an emulation of the tube recorder in Olympic Studios. Controls include wow, hiss, flutter, speed, bias, delay, feedback and flux (which controls tape saturation, compression and harmonic distortion simultaneously).


**Figure 5.** Kramer Master Tape UI [8]

*1.3.3 RC-20 Retro Color*

This plugin is an effects rack that aims to emulate general vintage recording gear. The modules of interest here are noise, wobble (controls wow and flutter) and distort (tape saturation setting).


**Figure 6.** RC-20 Retro Color UI [9]

*1.3.4 Lifeline Expanse*

This is an effects module with built-in tape saturation. There are tape, tube, fuzz and rectify algorithms, multiband drive, and controls for tone, mix, and frequency smoothing. The module of importance is the “dirt” module, shown below in Figure 7.


**Figure 7.** Lifeline Expanse "dirt" Module UI [10]

*1.3.5 Reels Lo-Fi Tape*

This is a tape emulation plugin based on an old Japanese tape machine. There are controls for harshness (mid-range harmonic distortion), wow, flutter, crosstalk (between left and right channels), hiss, and drop (volume loss due to physical imperfections).


**Figure 8.** Reels Lo-Fi Tape UI [11]

*1.3.6 Sketch Cassette II*

Sketch Cassette is different from most of the other plugins, since it is an emulation of lo-fi cassette tape. The principles of controlling the type of tape, saturation, hiss, wow and flutter are the same though, and users can also control the "age" of the tape as well, further enhancing the sound with antiquing.

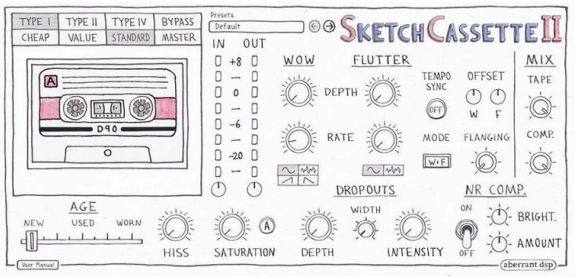


**Figure 9.** Sketch Cassette II UI [12]

*1.3.7 Goodhertz Tupe*

Tupe is inspired by 70s analog gear such as tape machines and mixing boards. There are four analog tape models to choose from (2 track hi, 2 track lo, C90 hi and C90 lo), and there is also an optical compressor/limiter. This one is a bit different in that there is less control over things like wow and flutter directly, but there are still controls for drive, bias, and noise.

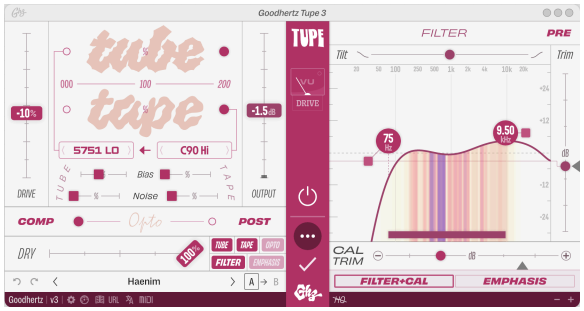


**Figure 10.** Goodhertz Tupe UI [13]

*1.3.8 Takeaways*

There are a number of controls shared across most of these plugins, namely saturation, drive, mix, bias, wow, flutter, and hiss. The main focus was implementing the first three parameters (saturation, drive, mix), and hiss was a bonus fourth feature that would add an

extra level of complexity to the project. A lot of these UIs also are made to look like a tape machine. For the purposes of this project, the tSaturator UI will not directly resemble a tape machine, as its digital signal processing is not technically built to resemble one specific machine. Additionally, that kind of UI work is too large for the scope of this project, but is mentioned in the "Future Modifications" section found later in this paper.

### 1.4 JUCE Framework, and Developing Real-Time Plugins

JUCE is an open-source C++ framework for developing cross-platform audio plugins. It provides a set of abstracted classes and interfaces that allows developers to be able to create real-time audio plugins as a VST (primarily for Windows and Linux operating systems), an AU (for MacOS), or AAX (for ProTools specifically, but requires some more configuration). Using JUCE makes interacting with incoming audio much easier to work cross platform.

## 2. DESIGN PROCESS

Once the background research was done, and the core functionality parameters were finalized (saturation, drive, mix, hiss), other ways of making the UI unique, but not just a jumble of knobs, were explored. This is why the addition of the spectrum analyzer was so crucial. This was the closest to getting a VU meter without having to actually implement one, as both elements give visual cues to users on what is happening sonically within the plugin The analyzer is bypassable, since rendering raw FFT data is taxing on the CPU. Before any coding was started, the UI design was sketched out on paper, as shown in figure 11.

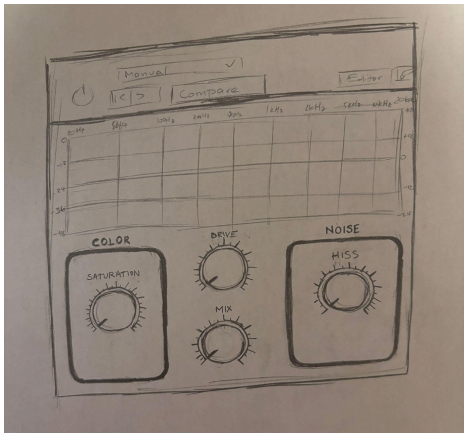


**Figure 11.** Original Paper Sketch of tSaturator UI

The motivation behind the different knob sections was really looking ahead to future iterations of this project, ones in which wow, flutter and bias controls would be a part of the plugin. Color and noise categories make sense in this way, as these controls fit into these categories. Each knob is labeled with the parameter it controls, and is made to resemble knobs on analog gear with the tick marks. Keeping the spectrum analyzer up top catered to familiarity in other plugins available on the market (not just tape saturation ones). The labels on the analyzer are as follows: the x-axis is frequency (in Hz) from 20Hz to 20kHz, and the y-axis is gain (in decibels) from 0dB to -48dB . This is a range in which much spectral information will be visible. This paper sketch was then moved into a more high-fidelity application, ProCreate, in order to refine the design so as to have a guide when starting to build the actual plugin. This is where the analyzer bypass button was added, as well as a branded title.


**Figure 12.** Iterated Design of tSaturator UI

This iterated design was used to construct the final UI in the plugin itself. Some tweaks were necessary, the most noticeable of which was removing the global bypass (power) button as seen above, since this feature gets included when the plugin is exported as an AU or VST. Additionally, in the final version, the scale of decibels on the y-axis was changed from -36dB to +12dB because it made it easier to see more data. Even so, this range may still require tweaking in the future, as the loudest settings usually go beyond the scope of what the graph can handle.

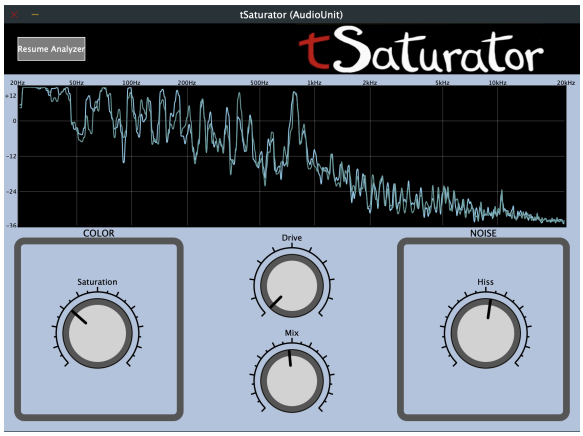


**Figure 13.** Final tSaturator UI

## 3. PLUGIN ARCHITECTURE AND DEVELOPMENT PROCESS

This project was a way to not only hone my understanding of developing real-time audio applications, but apply some of the principles learned in my computer science classes, namely the SOLID principles that are a cornerstone of object-oriented programming. SOLID stands for **s**ingle responsibility, **o**pen-closed, **L**iskov substitution, **i**nterface segregation, and **d**ependency inversion. The principles that will be highlighted against this project's design are the single responsibility and the open-closed principles. The single responsibility dictates that every class should have one and only one responsibility. The open-closed principle states that software should be open to extension, but closed for modification, meaning that new features are easier to add to the existing code (without having to do major refactoring) [14].

### 3.1 Plugin Architecture and DSP Overview

By the nature of JUCE's default plugin project architecture, the code is split into a `pluginProcessor` and a `pluginEditor` so that the signal processing and UI elements can live in separate places. `PluginProcessor` is the default (and root) processor that JUCE plugins use. Each part of the processing chain (saturation, drive, mix, and hiss) are constructed to be their own extension of the `ProcessorBase` class, in order to keep the code modular and uphold the single responsibility principle in the design. Each knob on the frontend

controls only the processor it is hooked up to, and each processor is a `Listener` of one and only one parameter knob.

The `ProcessorBase` class, part of JUCE's DSP module, exposes three helpful methods:

- `prepare`, which prepares the given processor with a `ProcessSpec`. The `ProcessSpec` contains information about the sample rate, maximum block size and number of channels of a context it will be called with. It is also where other pre-processing initialization (such as setting the mix rule, oversampling, preparing the EQ for the hiss, etc.) happens.
- `process`, which is basically where all the signal processing magic happens. Usually, this deals with working on some input block and transforming it as necessary (such as applying the distortion transfer function) and then returning the output block.
- `reset`, which resets the parameters of the processor.

The `Listener` class registers callbacks when a parameter is changed, and provides one crucial method for doing so:

- `parameterChanged`, which fetches the value of the provided parameter given its ID (which is unique), and replaces it with the new value provided to it.

The `PluginProcessor`'s member variables include one of each of the four processors and an `FFTAnalyzer`. This is used to convert the audio data from the output buffers into FFT data for the spectrum analyzer to visualize.

*3.1.1 Data Flow in the Application*

Before any audio is run through the application, `PluginProcessor`'s `prepareToPlay()` method is called. This ensures that any pre-playback setup is done at the right time. It is here that a `ProcessSpec` is declared for the processors to use, and also where the `maximumBlockSize` (512), `sampleRate` (44.1kHz) and `numChannels` (2) are set. Each audio processor is prepared with the `spec` object. Audio is then run through the application via the `PluginProcessor`'s `processBlock()` method, which chunks the audio into smaller buffers. This is the central engine which runs the plugin. The buffers are converted into `AudioBlock`s, which are passed to a `ProcessContextReplacing` object. This is a JUCE object that is used to feed a series of `AudioBlock`s to the `Processor`s and call their respective `process` methods. It is here where the `process` methods are called on each of the `DriveProcessor`, `SaturationProcessor`, `HissProcessor`, and `MixProcessor` in that order. The processors are in this order because of the way that certain processors need to work with the input block (mix* and saturation), while others need to work on the output block (mix* drive, hiss). The mixing processor must work on both the input block (dry signal) and the output block (wet signal), hence its inclusion in both categories. This is why `MixProcessor` is fed a copy of the dry input block before any processing is done, to ensure it works

properly. The FFTAnalyzer is also fed samples from this method, so that it can continually process the audio data. From there, the (split) dry audio flows into MixProcessor and DriveProcessor. The audio from DryProcessor then flows into SaturationProcessor, then HissProcessor, and finally MixProcessor (with the final wet signal). Once all of the saturation related processing is done, the FFTAnalyzer is fed samples from the output block, converted to a buffer, so that the analyzer shows the effect of the processing on the audio signal. This loop happens every

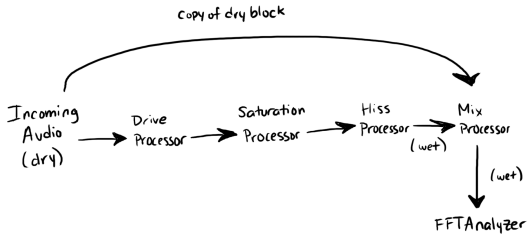


**Figure 14.** Data Flow Chart for tSaturator

*3.1.2 DriveProcessor*

The DriveProcessor is first in the chain, as it controls the added gain applied to the input signal before being sent to the SaturationProcessor. The drive knob registers values on a scale from 0-10, which is mapped to a scale of 0dB to +12dB. The method mapDriveScaleToDecibels does this, as shown below.

```
mapDriveScaleToDecibels(float drive)
{
  float gainInDecibels =
juce::jmap(drive, 0.0f, 10.0f, 0.0f,
12.0f); // 0dB to +12dB range
  curGain =
juce::Decibels::decibelsToGain(gainInDeci
bels);
}
```

The resulting value is then stored as the current gain used as a multiplier for the amount of gain applied. The process method applies gain in steps based on what the current and previous gain values looked like (which are stored in the processor as member variables). If the current gain is the same as the previous gain, then just the curGain value is used to apply gain. curGain is also manually set to be 0 at first, otherwise occasionally the plugin is opened and everything is extremely loud.

*3.1.3 SaturationProcessor*

The SaturationProcessor, which is second in the processing chain, is basically the meat and potatoes of this project. It does two main things: applies the distortion transfer function to generate saturation via a JUCE WaveShaper, and then applies a small amount of compression via a JUCE Compressor to make it tape saturation. These two steps are combined into one JUCE ProcessorChain, which allows for ease of calling process() on multiple processors in sequence. It makes the code a lot cleaner and modular.

First, in the prepare() method, the compressor is set with a gentle curve. This took some experimenting, and ultimately while I wanted to implement a 9dB soft knee, that was not possible to do with the

way JUCE's `Compressor` class operates. I tried to smooth the values, but that doesn't work. The settings for the compressor are as follows:

```
processorChain.get<SaturationChainPositio
ns::COMPRESSION>().setRatio(1.f);

processorChain.get<SaturationChainPositio
ns::COMPRESSION>().setAttack(30.f);

processorChain.get<SaturationChainPositio
ns::COMPRESSION>().setRelease(50.f);

processorChain.get<SaturationChainPositio
ns::COMPRESSION>().setThreshold(-30.f);
```

The compressor is initialized, and then the `Oversampler` is prepared with an oversampling factor of 2, and a `HalfBandPolyPhaseIIR` filter.

In `process()`, the incoming audio block is upsampled, and then the saturation value from the knob is scaled. The knob values range from 0-100, so this must also be mapped to an appropriate scale so there isn't too much or too little saturation. This value is used as a multiplier for the output of the signal once passed through the transfer function. `getSaturationMultiplier()` is the method that converts the 0-100 saturation value to a more workable number:

```
inline float getSaturationMultiplier()
{
  return std::pow(saturation / 100.f,
2.0f);
};
```

This took some experimenting to get right, as at first there was way too much saturation happening (the upside of this was at least there was confirmation the saturation was being applied to the signal). This multiplier is then applied to every sample in the block, which must be copied as an input block is read only. This also must happen *before* the `WaveShaper` processes the block. The upsampled block is then transferred into a `ProcessContextReplacing` object, which allows `process()` to modify it directly, and then the chain processes distortion and then compression. The distortion transfer function is the one from Figure 1 of the first section of this paper, as shown below:

```
void setDistortionTransferFunction
(const
juce::dsp::ProcessContextReplacing<float>
&context) {
  auto& waveshaper =
processorChain.get<SaturationChainPositio
ns::DISTORTION>();

  waveshaper.functionToUse = [] (float x)
{
  return (std::tanh(x) * 5.f) /
(std::tanh(5.f));
  };
}
```

There briefly was a gain reduction step happening after the processing was done, as the signal was getting far too loud, but other adjustments were made so that wasn't necessary anymore. Once the processing is complete, the block is downsampled to its original sample rate.

*3.1.4 HissProcessor*

The `HissProcessor` is the third processor in the processing chain. In this step, white noise is generated via a random number generator where the gain of the noise is controlled by the values coming out of the

Hiss knob. The generated noise is then passed through an 8-band equalizer to make its response curve similar to that of the Ampex ATR-102 machine, shown back in figure 3. The hiss knob registers values from 0-100 as a percentage, and thus must also be converted to properly affect the gain on a scale from -60dB to -12dB:

```
void HissProcessor::setGain(float
hissPercentage)
{ float targetGain =
juce::Decibels::decibelsToGain(juce::jmap
(hissPercentage, 0.0f, 100.0f, -60.0f,
-12.0f));
 curGain = targetGain;
}
```

The equalizer consists of low and high cut bands, low and high shelves, and four peak bands. They are set to the following settings to follow the Ampex ATR-102 response curve:

```
struct HissResponseCurveSettings
{ float lowCutFreq { 30.f },  lowCutSlope
{ 36.f };
  float lowShelfFreq { 75.5f },
lowShelfGainDecibels { -2.3f };
  double lowShelfQuality { 1.0 },
highShelfQuality { 1.3 }, lowCutQuality {
0.75 }, highCutQuality { 0 };
  float bandOneFreq { 50.f },
bandOneGainDecibels { -3.2f },
bandOneQuality { 3.6f };
  float bandTwoFreq { 85.f },
bandTwoGainDecibels { 2.f },
bandTwoQuality { 3.2f };
  float bandThreeFreq { 250.f },
bandThreeGainDecibels { -1.7f },
bandThreeQuality { 2.f };
  float bandFourFreq { 5000.f },
bandFourGainDecibels { -0.5f },
bandFourQuality { 1.8f };
  float highShelfFreq { 0 },
highShelfGainDecibels { 0 };
  float highCutFreq { 0 }, highCutSlope {
6.f };
};
```

The white noise is generated by the following function:

```
void HissProcessor::processBlock(float
*buffer, int numSamples)
{
        for (int n = 0; n < numSamples;
++n)
        {
        float noise = (random.nextFloat()
- 0.5f) * curGain;
        buffer[n] += noise;
        }
}
```

Noise is processed per channel, with the EQ applied after the noise is generated.

*3.1.5 MixProcessor*

The `MixProcessor` is the final processor in the chain, as it requires all other processing to have finished on the wet signal side, but also requires being given a copy of the input block *before* any other processing is applied. This is done by the following line:

```
mp.setDryBlock(inBlock);
```

This instantiates the `MixProcessor` (mp) with blocks from the original dry input that is run through `PluginProcessor`, to ensure that the correct dry signal is being mixed with the wet signal. In `setDryBlock()`, the dry block must be copied into a buffer every time a new block is processed, otherwise the processor does not get the most current audio samples and the application will break.

`MixProcessor` is essentially a wrapper of the `DryWetMixer` class that JUCE provides, with the only real additions being the custom copying of the input block to mix as the dry samples, and a percentage conversion function (because the knob values range from 0-100%, and need to be converted to a floating point scale from 0.0f to 1.0f.

**3.2 UI Overview**

There are three main elements to the user interface: the header bar, the spectrum analyzer, and the knob panel.

*3.2.1 Header Bar*

The header bar contains two things: the analyzer bypass button and the plugin title. Pressing the button turns it filled in white at 50% opacity and pauses the spectrum analyzer, in case it uses too much CPU. The text changes from “Pause Analyzer” to “Resume Analyzer”, and vice versa.

*3.2.2 Spectrum Analyzer*

The spectrum analyzer shows the left and right channels of audio in real time and can be paused using the bypass button in the header bar. The y-axis ranges from -36dB to +12dB, and the x-axis ranges from 20Hz to 20kHz. The x-axis, despite being logarithmically portioned out, may still be slightly off-kilter, but there was not enough time to address this unfortunately.

Let it be noted that the spectrum analyzer adapts code from the official JUCE spectrum analyzer tutorial [21]. The files in which this code appears have the proper credit and licensing included, and this code is free to use if those attributions are made, as denoted in the original files from JUCE. The ISC license is included with the project.

*3.2.3 Knob Panel*

The knob panel contains the three different sections: Color, with the saturation knob, the untitled middle section with the drive and mix knobs, and Noise, with the hiss knob. The knobs are styled to be reminiscent of knobs on analog gear.

**3.3 Development Process**

Since this project was my first attempt at creating a JUCE plugin, I looked to various sources and their respective codebases in order to get a better handle on how to work with JUCE, how to structure my code, and just general DSP/saturation concepts. The following projects/repositories (also mentioned in the references) were used as reference material in that regard:

- SimpleMultiBandComp by matkatmusic [15]
- SimpleEQ by matkatmusic [16]
- crispy by spensbot [17]
- AnalogTapeModel by Jatin Chowdhury [18]
- Schrammel_OJD by JanosGit [19]
- viatordsp by landonviator [20]
- PluginGuiMagic by ffAudio [21]
- Official JUCE Spectrum Analyzer tutorial [22]

The development process started with recreating the UI in the JUCE environment. Once the basic UI framework was laid out, the DSP was developed. I worked in the AudioPluginHost that comes with JUCE’s

environment to test and develop the DSP to bring it to the level it needed to be at. The final steps included moving the plugin into a DAW like Logic or Ableton. This uncovered some errors that wouldn't have otherwise been found, namely the channel configuration settings. It works in Logic, but for some reason Ableton doesn't want to pick it up. Packaging the plugin for release was also cumbersome.

## 4. FUTURE MODIFICATIONS

For the purposes of this project, the scope of the parameters implemented was fairly rudimentary. There are a few areas of improvement I have in mind for future iterations of this project: improving the UI, adding more new controls, and adding menu options for existing controls. These modifications were based on prior research conducted, as well as notes made during the development process that just could not be addressed within the timeframe.

### 4.1 Improving the UI

The UI is fairly simplistic, and was a bit of an afterthought as compared to the functionality of the plugin. The UI design was intended to be simplistic, arranging the controls in a way that makes sense, and the spectrum analyzer was included to help visualize the changes the audio goes through. The assets themselves aren't super fancy, and thus I'd like to make the UI a bit more professional looking in a future iteration. This would include finding and making proper assets for the knobs and the background. Additionally, I would like to add the live parameter value updating under the knobs back in. It was there for a time, but between wrangling with the layout and preventing the plugin from crashing altogether, this was not feasible to keep in.

### 4.2 Adding more new controls

To make tSaturator a more comprehensive tape saturation plugin, the next controls that should be added are wow, flutter and bias. These are all controls found in many other saturation and tape saturation plugins, and would make the sound of the plugin more realistic. These were omitted from the initial project because they are time consuming and very math heavy.

### 4.3 Adding menu options for existing controls

Currently, there is only one transfer function dictating the amount of distortion, and one EQ curve affecting the frequency spectrum of the hiss. There's a lot of customizability that can be done with both of these parameters, so in the future I'd do more in-depth research about the distortion and frequency responses of different vintage tape machines, and then allow users to choose between them via a menu. This is akin to the J37-Tape and Sketch Cassette II, both of which allow users to select things like the tape formula.

## 5. CHALLENGES & THOUGHTS

While the processors developed for this worked with `ProcessContextReplacing` objects, most other audio applications work directly with buffers. This also made streaming the audio data through an FFT very difficult, and a lot of time was spent tweaking the `FFTAnalyzer` (DSP) and `SpectrumAnalyzer` (UI) to be responsive,

not use too much CPU, and not have the lines drawn be super clunky.

Additionally, getting everything to be placed correctly on the UI was a gargantuan task. I spent a long time lining things up pixel by pixel, making sure the code was still structured in a way that did not expose parts of the code to classes that didn't need it while still creating the interface I wanted.

Of course, no coding project would be complete without any gremlins appearing in the code. There were a few times that despite everything appearing as if it should work correctly, the saturation stopped being applied, then was too loud (there used to be a small gain reduction step before the block got downsampled, but it was eventually removed), the drive was improperly initialized (it would be set to 0 but sometimes upon opening the plugin would negative value that created very loud sounds), and when the mix opens set to 100%, nothing can be heard (which makes sense, there is no wet signal to be heard at first).

Overall, this was a good way for me to tie together all of the skills I have learned throughout my college career, as well as get my feet wet for what I hope to work on in the future.

## 6. REFERENCES


[1] "History of Distortion Part 1," *The JHS Show*. https://thejhsshow.com/articles/history-of-distortion-part-1

[2] "History of Distortion Part 2," *The JHS Show*, Nov. 22, 2018. https://thejhsshow.com/articles/history-of-distortion-part-2

[3] "Reel History," *Reel-to-Reel Tech*. https://reeltoreeltech.com/reel-history/

[4] Digital Audio Antiquing—Signal Processing Methods for Imitating the Sound Quality of Historical Recordings. Välimäki et. al, 2007.

[5] "Introduction to Oversampling for Alias Reduction," *Nick Thompson*, Jul. 24, 2018. https://www.nickwritesablog.com/introduction-to-oversampling-for-alias-reduction/

[6] "Response Curves of Analog Recorders," *www.endino.com*. https://www.endino.com/graphs/

[7] "J37 Tape," *waves.com*, 2013. https://www.waves.com/plugins/j37-tape

[8] "Kramer Master Tape," *waves.com*, 2016. https://www.waves.com/plugins/kramer-master-tape

[9] "RC-20 Retro Color - XLN Audio," *www.xlnaudio.com*. https://www.xlnaudio.com/products/addictive_fx/effect/rc-20_retro_color

[10] Lifeline Expanse — Excite Audio, "Excite Audio," *Excite Audio*, 2025. https://www.excite-audio.com/lifeline/lifeline-expanse

[11] "Reels - Analog Tape Plugin with Echo and Tape Stop Effect (VST, AU, AAX)," *AudioThing*, Mar. 27, 2025. https://www.audiothing.net/effects/reels/

[12] "SketchCassette II," *Aberrant DSP*. https://aberrantdsp.com/plugins/sketchcassette/

[13] "Tupe, by Goodhertz," *Goodhertz.com*, 2022. https://goodhertz.com/tupe/

[14] "ArticleS.UncleBob.PrinciplesOfOod," *www.butunclebob.com*. http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod
[15] matkatmusic, "GitHub - matkatmusic/SimpleMultiBandComp," *Github.com*, 2025. https://github.com/matkatmusic/SimpleMultiBandComp
[16] matkatmusic, "GitHub - matkatmusic/SimpleEQ: The code for the SimpleEQ C++ Plugin Project featured on FreeCodeCamp YT channel," *GitHub*, 2021. https://github.com/matkatmusic/SimpleEQ
[17] spensbot, "GitHub - spensbot/crispy: An Audio Saturation Plugin (VST/AU) made with JUCE," *GitHub*, 2020. https://github.com/spensbot/crispy
[18] jatinchowdhury18, "GitHub - jatinchowdhury18/AnalogTapeModel: Physical modelling signal processing for analog tape recording," *GitHub*, Nov. 05, 2023. https://github.com/jatinchowdhury18/AnalogTapeModel
[19] JanosGit, "GitHub - JanosGit/Schrammel_OJD: Audio plugin model of a modern classic guitar overdrive Pedal," *GitHub*, 2020. https://github.com/JanosGit/Schrammel_OJD
[20] landonviator, "GitHub - landonviator/viatordsp," *GitHub*, 2022. https://github.com/landonviator/viatordsp
[21] ffAudio, "GitHub - ffAudio/PluginGuiMagic: Examples for foleys_gui_magic - the styleable plugin gui," *GitHub*, 2023. https://github.com/ffAudio/PluginGuiMagic
[22] "Visualise the frequencies of a signal in real time - JUCE Tutorial - JUCE," *JUCE*, 2024. https://juce.com/tutorials/tutorial_spectrum_analyser/