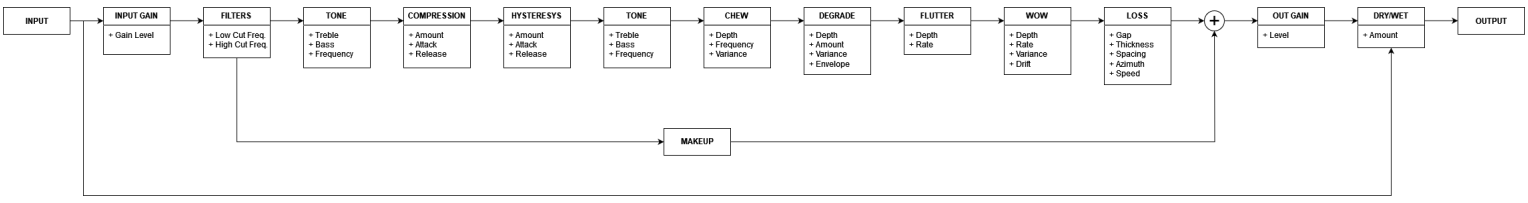
CMLS HW2 - Analysis of a JUCE plugin

Xenharmonics Group:
Baroli Gabriele
Ferrando Alessandro
Mauri Noemi
Passoni Riccardo

Chow Tape Model - by Jatin Chowdhury

ChowTape is an analog tape machine physical model. It's a plugin developed with the JUCE framework which simulates the qualities and effects of a real analog audio tape.

The next figure is showing the signal flow of the plugin. Each block is explained in detail in the following parts. Each audio processing block's implementation and working principles will be explained in the rest of the report.



Main Controls

Input Gain controls the gain level before the signal goes into the rest of the plugin. **Output Gain** controls the level coming out of the plugin.
Input and Output gain are part of the *GainProcessor* class which is responsable of getting the new value of gain when it is changed and set the new gain.

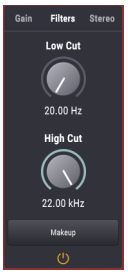
Dry/Wet allows users to choose how much of the signal is processed by the plugin.
It is executed by the *DryWetProcessor* class which outputs a buffer that consist of a mix of a wet and dry buffer with different gain. The sum of each gain is equal to 1.
Mix Group: When using ChowTape on multiple channels in a mix, you can synchronize parameters between plugin instances belonging to the same mix group. Plugin instances that are part of the same mix group will share the same parameters. It is implemented by the *MixGroupConstants* and *MixGroupsShareData* classes which generate different groups of plugin instances and change the parameters simultaneously.

Bypass

The BypassProcessor class is used to remove the effect from the audio processing chain (controlled visually with a yellow power button in the instrument), this is done by declaring a bypass variable in ChewProcessor.h, which is then used in its implementation (.cpp) file. Its method processBlockIn is called at the beginning of the processBlock in order to determine (based on the boolean return value of the method) whether or not to perform the necessary calculations needed to apply the effect. If calculations are performed the output is routed through its processBlockOut method.

Input filters

Input filters apply a **low-cut** and **high-cut** filter to the input signal before it is passed on to the rest of the plugin. The cutoff frequencies of the two filter are controlled by two knobs.



The Makeup control allows the signal cut out by the input filters to be added back to the output of the plugin. Makeup be useful by allowing sub-bass frequencies to pass through the plugin unaffected.
In the header file, *InputFilters.h* 3 AudioBuffer are initialized for the low cut filter, high cut filter and the makeupBuffer which consists in a switch on/off for the filter effect.

```
1 AudioBuffer<float> lowCutBuffer, highCutBuffer, makeupBuffer;
```

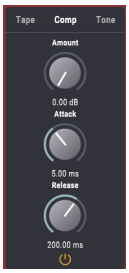
The 2 filter are then initialize by the class LinkwitzRileyFilter which performs througth *processSample* function the filter operation on a single sample at a time and returns both the low-pass and the high-pass outputs.

```
1 LinkwitzRileyFilter<float> lowCutFilter;
2 LinkwitzRileyFilter<float> highCutFilter;
```

In the *InputFilters.cpp* file the ranges for the 2 filters are defined: 20-2000Hz for the low cut filter and 2000-20000Hz for the high cut filter. The processBlock performs the filtering and the makeup operations, getting the value of the cut frequencies thanks to the gettingValue() function.

Compression

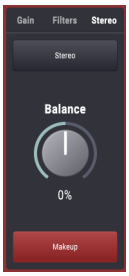
The compression section applies a characteristic compression curve to the signal, which can be usefulfor reducing the dynamic rangeof the signal before going into the hysteresis processing.
Amount controls the level at which the compression curve starts to take effect. At 0 dB, the compression has no effect on the signal. At 9 dB, any signal above -9 dB will start to be compressed.



The **Attack** parameter controls how quickly the compression will start to take effect once the signal enters in the compression region.
The **Release** parameter controls how quickly the compression backs off once the signal is no longer in the compression region.
In the header file, *CompressionProcessor.h* the amount, attack and release parameters are defined. In the *CompressionProcessor.cpp* file the range in which the cmpression will take place is defined by the *compressionDB* function. The compression is then performed by the processBlock. The compGainVec contains new compressed gain value while the slewLimiter vector defines the attack and release action. The compression is then applied to the samples by a multiplication by the compression gain vector. A bypass button is used to instantly remove the effect of compression from the signal.

Stereo/Mid-Side

When using ChowTape in Stereo mode, the plugin will process the input signal as a left/right pair. In Mid/Side mode, the plugin will encode and process the input and mid-side mode and then decode the signal back to a left/right stereo pair at the output. A button will let users switch between the stereo and mid-side mode.



The Balance control makes one of the channels louder than the other by panning the signal when in stereo mode or by emphasizing either the mid or side part of the signal.
When in Mid-Side mode the Mid consists of the sum of left and right channel while the side consists of the difference between the two.
The Makeup control allows the signal which was attenuated by the Balance control to be amplified by the same amount at the output of the plugin.
Everything is implemented in the MidSideProcessor class.

Hysteresis

The Hysteresis Loop is an important tool for understanding the behaviour of a magnetic material. It shows the relationship between an induced magnetic flux density B and a magnetizing force H, and how the magnetization of a material changes as the magnetic field is cycled between positive and negative values.
In a simulation of an Analog Tape, the magnetisation of the tape can be represented with the Hysteresis Loop using the formula $\vec{M}(x,y) = F_{Loop}(\vec{H}(x,y))$, where F_{Loop} is a generalized hysteresis function.
The Mathematical model used to describe the magnetisation behaviour is the Jiles-Atherton model.
The model uses a set of differential equations to describe the magnetization behaviour of the material. These equations are based on a number of parameters that describe the properties of the material.
The plugin allows the user to choose different mathematical method as solver for the Jiles-Atherton model as the Runge-Kutta and the Newton-Raphson.

Hysteresis Loop Implementation

The implementation of the Hysteresis function is done in the *HysteresisOps.h* file. Here a struct is used to store the state ofthe Hysteresis and to start some initial parameters calculations such as the magnetisation saturation.
In the same file the functions *hysteresisFunc* and *hysteresisFuncPrime* implement the Jile-Atherton model as function of the magnetic field of a specific direction and with respect to time.
We can notice that there are some functions which calculate the Langevin method. This method is used to calculate the magnetization of a material as a function of the applied magnetic field, taking into account thermal fluctuations.
In the *HysteresisProcessing.h* provides methods to reset the processing state and to set the sample rate. It also provides a templated *process()* method that takes a SolverType template parameter and a Float parameter and performs hysteresis processing on a single sample using a solver of the specified type and the *hysteresisFunc* function of the *HysteresisOps* class. The solvers implemented are Runge-Kutta 2, Runge-Kutta 4, Newton-Raphson 4, Newton-Raphson 8, and an STN solver.

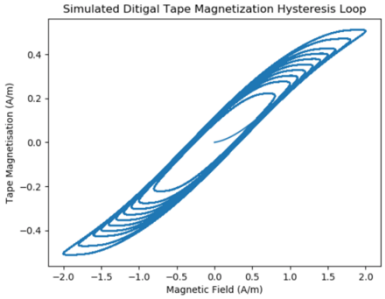


Figure 2: *Digitized Hysteresis Loop Simulation*

In the *HysteresisProcessor.cpp* the setter functions for the Hysteresis parameters are implemented (*setSolver()*, *calcMakeup()*, *setDrive (float newDrive)*, *setWidth (float newWidth)*, *setSaturation (float newSaturation)*, *setOversampling()*).
The method *prepareToPlay()* is called to initialize the processor before it starts processing audio. It resizes the dimension of the parameters with respect to the number of channels.
In the processBlock function the parameters are setted and we can also notice that the input buffer is clipped in order to avoid unstable hysteresis caused by aliasing artifacts at the Nyquist frequency. Those problems are generated by the approximations used in the formulas that are used to build the model.

```
// clip input to avoid unstable hysteresis
    for (int ch = 0; ch < numChannels; ++ch)
    {
        auto* bufferPtr = buffer.getWritePointer (ch);
        FloatVectorOperations::clip (bufferPtr,
                                     bufferPtr,
                                     -clipLevel,
                                     clipLevel,
                                     buffer.getNumSamples());
    }
```

After that there are different implementations of the process() method over the audio block: The *processSmooth()* function applies additional smoothing to the hysteresis processor's parameters before processing each sample, while the *process()* function simply processes each sample using the hysteresis processor.

```
template <SolverType solverType, typename T>
void HysteresisProcessor::process (chowdsp::AudioBlock<T>& block)
{
    const auto numChannels = block.getNumChannels();
    const auto numSamples = block.getNumSamples();

    for (size_t channel = 0; channel < numChannels; ++channel)
    {
        auto* x = block.getChannelPointer (channel);
        auto& hProc = hProcs[channel];
        for (size_t samp = 0; samp < numSamples; samp++)
```

```
            x[samp] = hProc.process<solverType> (x[samp]);
    }

    applyMakeup<T> (block, makeup);
}

template <SolverType solverType, typename T>
void HysteresisProcessor::processSmooth (chowdsp::AudioBlock<T>& block)
{
    const auto numChannels = block.getNumChannels();
    const auto numSamples = block.getNumSamples();

    for (size_t channel = 0; channel < numChannels; ++channel)
    {
        auto* x = block.getChannelPointer (channel);
        auto& hProc = hProcs[channel];
        for (size_t samp = 0; samp < numSamples; samp++)
        {
            hProc.cook (drive[channel].getNextValue(), width[channel].
                getNextValue(), sat[channel].getNextValue(), false);
            x[samp] = hProc.process<solverType> (x[samp]);
        }
    }
```

Chew

This section of the instrument's audio processing chain simulates noise and imperfections that could be present in a real audio tape. It can deteriorate over time in real-world recording circumstances and as a result, the tape may produce noise and imperfections that might give the recorded sound more low fidelity qualities.
The settings specifically include options for emulating worn-out, old tape from repeated use as well as tape that has been chewed up by a malfunctioning tape machine. The **Chew Controls** consist of three distinct parameters: *Depth*, *Frequency*, and *Variance*. These controls influence the quality of the chewed up tape from a malfunctioning tape machine. Users can produce subtle or obvious effects by adjusting the Depth of the chew marks on the tape. The spacing between the tape segments that have been chewed up is controlled by Frequency, and the Variance option adds unpredictability to the amount of space between these segments.

The effect is implemented in the files *ChewProcessor.h/cpp* (with the inclusion of *BypassProcessor.h*, *DegradeFilter.h* and *Dropout.h*).
The included header file *Dropout* applies an irregular reduction in volume to random samples in the audio signal (since the power value has a random element to it in the main *processShortBlock*), simulating the effect of a damaged audio tape. The implementation uses a vector of linear smoothed values to control the mix and power parameters for each channel. Importantly, the effect is bypassed if the target mix value is equal to 0. The *process* function uses *dropout()* to calculate the effect for a given sample value and channel. It first computes the sign of the input signal to preserve the phase information of the signal while applying the distortion effect. Next it calculates the absolute value of the input sample and then raises it to the corresponding power value. Finally, the function multiplies the sign of the input signal with the power result to preserve the original phase information and returns the sample.
The *DegradeFilter* header file implements a lowpass filter class to perform further degradation in this particular effect. The *calcCoefs* function calculates the filter coefficients (a0, b0, b1) based on the current frequency value. The *process* function applies the filter to the buffer of samples and *setFreq* allows to change the frequency of the filter when it is called. It uses a second

order IIR low pass filter structure with coefficients that are updated based on the current cutoff frequency.

```
float y = x * b[1] - y * a[1] + x * b[0];
```

The *Variance* parameter influences the difference in frequency between the chewed parts of the tape, using the *getDry/WetTime* declared in the header file (this decides when the effect is applied in the processing function of the effect):

```
auto varScale = pow (random.nextFloat() * 2.0f, var->getCurrentValue());
return random.nextInt (Range<int> ((int) ((1.0 - pow (*freq, 0.1f)) * sampleRate
    * varScale), (int) ((2 - 1.99 * pow (*freq, 0.1f)) * sampleRate * varScale)));
```

The process function first initializes some variables. It calculates the highest frequency that the effect will process (up to 22000 Hz or 0.49 times the sample rate, whichever is smaller) and the amount by which this frequency will change based on the depth of the effect.
The function then determines whether the frequency of the effect is zero. If that is the case, it resets the mix value to zero and changes the frequency of each filter to the previously determined high frequency. This means that the output is simply the dry signal.
If the frequency is 1, the function sets the mix value to 1 and the depth parameter is multiplied by 3 to calculate the effect's dropout power. Each filter's cutoff frequency is set to a number that is lower than its highest possible value and dependent on the depth.

```
const auto filterFreq = highFreq - freqChange * *depth;
```

The function detects whether or not it is now in a *crinkled* state (damaged tape) if the depth is between zero and one. If so, the function adjusts each filter's frequency and power based on the depth of the effect and a random value. Since the effect sets the mix value to zero and the frequency of each filter to the high frequency value, the output is unaffected by the effect if it is not in a *crinkled* condition. The function sets a counter upon moving from the *crinkled* state to the *not crinkled* state or vice versa.
Finally, the function sets the mix and power values of the dropout and processes the audio buffer with it. Then, the audio buffer is processed by each of the filters using the *process* function of each filter. The function also increments a counter by the number of samples in the input buffer, this is needed in order to keep track of when the chewing effect has to be applied.

Degregation

Tape Degradation Controls include *Depth*, *Amount*, *Variance*, and *Envelope*. Users may produce subtle or dramatic wear effects by adjusting the *Depth*, which regulates the severity of wear on the tape. The wear effect may be controlled more precisely and subtly using the *0.1x* option. With a parameter that is analogous to the tape's age, *Amount* regulates how much it wears out. By adding a time-varying unpredictability to the degradation effect, the *Variance* simulates the random distressing of a real tape. The *Envelope* option allows the user to apply an amplitude envelope to the tape noise.
The effect is implemented in the files *DegradeProcessor.h/cpp* (with the inclusion of *BypassProcessor.h*, *DegradeNoise.h*, *DegradeFilter.h* and *GainProcessor.h*).
The *BypassProcessor* class is used in the same way as the previous effect, as is the *DegradeFilter*.
The *GainProcessor* header implements gain getters/setters and applies a GainRamp to the buffer.
DegradeNoise creates noise to simulate tape degradation. It contains methods for preparing the noise generator (sets the old gain value), setting the current gain, and processing a block of audio samples by adding noise (random values) to each sample. The quantity of noise introduced is determined by the current and previous gain levels, as well as the sample's location within the block.

This effect also applies the effect by dividing the data into smaller buffers (in this case of 2048 samples or less). The function first checks whether to bypass the processing, if bypass is active, the audio data in the buffer is returned as is. If bypass is not active, the audio data is processed further.
The number of channels and samples in the audio buffer is then retrieved, and a sample counter is incremented by the number of samples. If the sample counter is equal to or greater than the predetermined small block size, *cookParams()* function is called to update the processing parameters. The sample counter is the reset to zero.
The level detector object's *process()* function is called to obtain the audio level of the input samples.
A new AudioBuffer is created to hold noise data with the same number of channels and samples as the input buffer, and it is cleared. The noise data is then created using a separate *noiseProc* object's *processBlock()* on each channel.
The signal level is then multiplied by the noise data if an envelope of the noise is enabled.
The noisy signal is written to the buffer, the resulting data is then passed to a *DegradeFilter* object for additional degradation processing.

Loss Filter

This section of the plugin emulates the real frequency response of a tape machine, which has a strong dependence on the playhead conditions: the playhead gap, its distance from the tape, the tape thickness and its speed are the parameters taken into consideration. Another effect considered is the possibile misalignment between the playhead and the tape, which causes a stereo *widening* effect.

As we can see from the picture, the GUI lets the user set these parameters through sliders. The speed control is continuous as the others, but its value can be quantized to standard reel-to-reel machines speeds trough suitable buttons below the slider.
The loss effect can be described analytically by this formula:

$$V(t) = V_0(t)[e^{-kd}]\Big[\frac{1-e^{-k\delta}}{k\delta}\Big]\Big[\frac{sin(kg/2)}{kg/2}\Big] \tag{1}$$

Where $V_0(t)$ is a generic sinusoid, k is the wave number, d is the distance between the playhead and the tape, δ is the tape thickness and g is the playhead gap. The wave number is given by:

$$k = \frac{2\pi f}{v} \tag{2}$$

where f is the frequency of the sinusoid and v is the tape speed.
The loss filter is implemented in the following files:

- *LossFilter.h/cpp*, which takes care of the signal filtering through the LossFilter class.
- *AzimuthProc.h/cpp*, which only works if the input audio is stereo, and applies a time misalignmente between the two channels, through the AzimuthProc class.

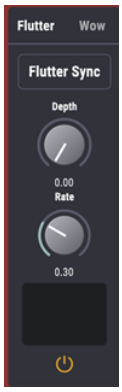
The loss filter is implemented as a variable order (default 64) FIR filter, whose coefficients are obtained through inverse DFT applied to the loss effects described in the above equation (function *calcCoefs*). Another effect embedded in this processor is the so-called head bump effect, modeled by an IIR peak filter with central frequency and gain given by:

```
1 auto bumpFreq = speedIps * 0.0254f / (gapMeters * 500.0f);
2 auto gain = jmax (1.5f * (1000.0f - std::abs (bumpFreq - 100.0f)) / 1000.0f, 1.0f
    );
```

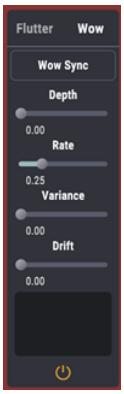
and a constant $Qfactor$ of 2. The function *calcHeadBumpFilter* takes care of setting the filter coefficients. The effects described above are then applied when the *processBlock* function is called.
The disallignment between the two tracks of a stereo input as said is managed by the AzimuthProc class. This effect is achieved using a delay line. When the user drags left or right the Azimuth control, the function *setAzimuthAngle* is responsible for the calculation of the number of samples to delay the signal by. The formula used is $ns = f_s dv$ where f_s is the sampling rate, v is the tape speed and d is the further distance between one edge of the tape and the playhead, given by $d = Wsin(\alpha)$ where W is the tape width, assumed constant and equal to $1/4$ and α is the azimuth angle. The number of samples obtained is smoothed to avoid audio glitches. This effect is also applied when the *processBlock* function is called.

Flutter - Wow effects

A real tape machine is subject to speed fluctuations due to mechanical imperfections. These flutters produce a peculiar effect on the produced audio which is called precisely **Flutter** when they act on a relatively short time window, and **Wow** when their effect acts on a longer period. The resulting sound is distorted in pitch.



The Flutter section of the plugin presents three controls and a visualizer: a Depth and Rate knob, a Sync button, and visual feedback. The Sync button lets the user sync the rate of the modulation to various options, like the tempo of the song or the tape speed control described in the Loss section. When active and with more than zero Depth, the visualizer will plot a pulsating red circle, to help the user understand the effect behaviour. The Wow controls GUI is similar to the previous one, but with two more controls: a *Variance* and a *Drift* slider. These two control the amount of randomness in the phenomenon. We still have a Sync button and a similar visualizer.



The processors used are implemented in the following files:

- *WowFlutterProcessor.h/cpp*
- *FlutterProcess.h/cpp*
- *WowProcess.h/cpp*
- *OHProcess.h*

Here the architecture contemplates a main processor which actively transforms the input audio buffer, and three processes which stage the changes to be applied. The effect is implemented through a delay line, and the total delay to apply to the single audio sample is determined by computation of both the processes.
The Flutter process evaluates the sum of three cosine functions, whose amplitude is related to the Depth parameter, and whose phase is related to the Rate parameter. In particular, the second and third function have a circular frequency twice and three times the first one, respectively.
The Wow process instead evaluates a singular cosine function, whose amplitude is dependent on the Depth paramete, but whose phase presents a multiplicative random contribution informed by the Drift parameter, and an additive parameter given by the *OHProcess* class. This class implements an *Ornstein-Uhlenbeck* process, which describes a mean-reverting Brownian motion. For each sample block a realization of this process is summed to the Wow cosine phase.
The resulting delay is the sum of the values returned by the processes, plus their respective DC offsets. The latter ones will then be removed by a DCBlocker object.