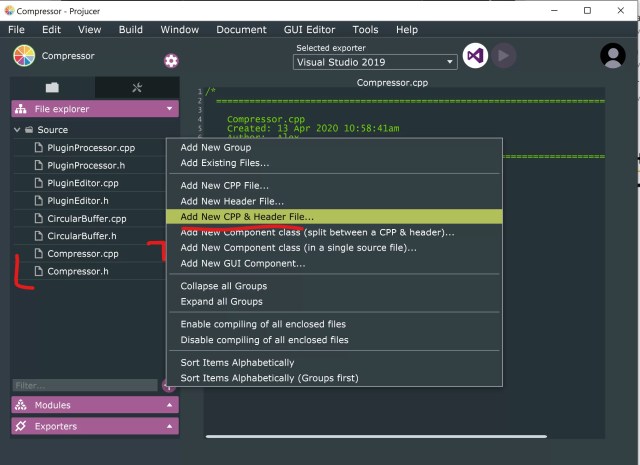
Build working compressor with attack/release controls and a soft/hard knee selector.

Create Compressor plugin with Projucer. Copy **CircularBuffer** files from last tutorial to source folder. Download at [github.com/aRycroft/JuceTutorial5](https://github.com/aRycroft/JuceTutorial5)



Select*Add New CPP & Header File…* & create **Compressor.cpp** and **Compressor.h** files**.**

**Compressor Class**

A screenshot of a computer

Description automatically generated with medium confidence

Add the following base code to **Compressor.h**.

#include "CircularBuffer.h"  
#pragma once  
class Compressor {  
public:  
private:  
};

Add these in private section of class. *tav*is averaging time used when calculating *rms.*

CircularBuffer buffer;  
float tav, rms, gain;

For now compressor class has one public function & constructor that takes no arguments.

Compressor();  
float compressSample(float data, float thresh, float ratio, float attack, float release, float knee);

A screenshot of a computer program

Description automatically generated with medium confidence

**Compressor.h**should look like this,

Create definitions for the two functions in **Compressor.cpp.**

In **Compressor()** constructor initialise CircularBuffer, set *tav*, *rms* & *gain* to default values.

buffer = CircularBuffer(150, 20);  
tav = 0.01;  
rms = 0;  
gain = 1;

**How Compressors Work**

3 main operations for compressor; how loud incoming signal is, how much to reduce gain by, smooth gain transition to avoid artefacts.

**1. Gain Detector**: We used peak detector to determine when to activate limiter. Now we find **RMS**value of waveform. **RMS**is continous power of waveform over time.

A screenshot of a computer

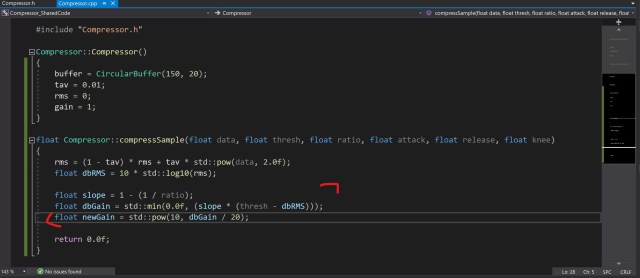
Description automatically generated with medium confidence

rms = (1 - tav) \* rms + tav \* std::pow(data, 2.0f); //1  
float dbRMS = 10 \* std::log10(rms); //2

//1 Adjusts the rms value depending on the incoming signal.  
//2 Converts this value to the decibel scale.

**2. Gain reduction**: Users pick [compression](https://audioordeal.co.uk/audio-compression-intro-and-basics/) ratio, how much signal compressed once passes threshold. Convert ratio to slope, 4:1 -> 0.75, 2:1 -> 0.5

Next we multiply **(threshold – dbRMS)** by this **slope** factor. Then check that it is below 0 to make sure the signal is never boosted if the **rms** value is above the **threshold**.



float slope = 1 - (1 / ratio); //1  
float dbGain = std::min(0.0f, (slope \* (thresh - dbRMS))); //2  
float newGain = std::pow(10, dbGain / 20); //3

//1 Calculate slope from ratio  
//2 Find the gain to be applied in db, and make sure it’s less than 0.0f  
//3 Calculate the newGain in linear scale

**3. Smooth transition:** Now we know what gain to apply to the current signal we need to smooth the transition to this new value. We do this using **attack** and **release** times.

A screenshot of a computer

Description automatically generated with medium confidence

float coeff;  
if (newGain < gain) coeff = attack; //1  
else coeff = release; //2  
gain = (1 - coeff) \* gain + coeff \* newGain; //3

//1 Declare **coeff** variable, if **newGain** is less than current gain set to **attack** variable  
//2 Else set to **release**  
//3 Adjust **gain** based on new gain and coeff

Finally we need to set and get values from our **CircularBuffer**.

A screenshot of a computer program

Description automatically generated with medium confidence

float compressedSample = gain \* buffer.getData();

buffer.setData(data);

buffer.nextSample();

return compressedSample;

**Coding PluginProcessor**

We have working Compressor class, but not using it. Go to **PluginProcessor.h**.

A screenshot of a computer

Description automatically generated with medium confidence

#include "Compressor.h" //1  
  
Array allCompressors; //2

Include new **Compressor** class at top of file, & declare new array of **Compressors**.

Now in **PluginProcessor.cpp**

A screenshot of a computer

Description automatically generated with medium confidence

//Prepare To Play

for (int channel = 0; channel < getNumOutputChannels(); channel++) {

allCompressors.add(Compressor());

}

//Process Block

for (int i = 0; i < buffer.getNumSamples(); i++) {

for (int channel = 0; channel < getTotalNumOutputChannels(); channel++) {  
 auto\* data = buffer.getWritePointer(channel);

Compressor\* comp = &allCompressors.getReference(channel); //1

data[i] = comp->compressSample(data[i], -30.0f, 20.0f, 0.01f, 0.4f, 0.0f); //2   
 }

}

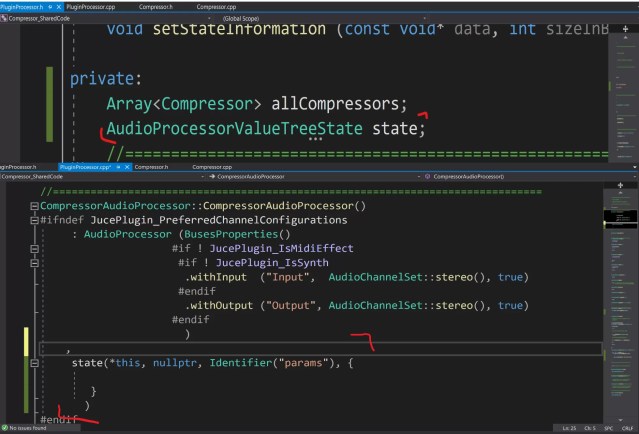
In **PrepareToPlay** create an array of Compressors.

In **ProcessBlock**loop through samples and channels.  
1 Get reference C**ompressor** for the current channel.  
2 Calculate compressed samples with some initial values passed to *compressSample* function.

Build plugin now and you hear the incoming audio being compressed. Try changing the values passed to the compressor class, then rebuilding to hear the difference.

**GUI**

Use **ValueStateTree** to make juce plugin GUI, save & recall plugin state, & add parameter automation in DAW. Editor access & edit it, Processor only reads from it.



//PluginProcessor.h

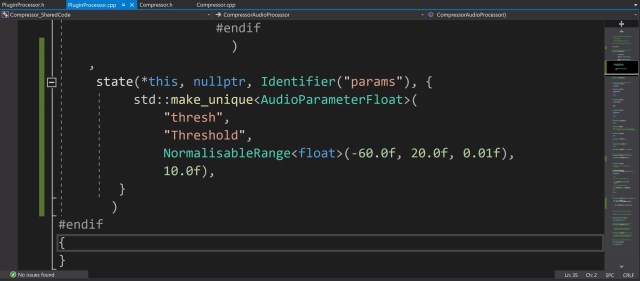
AudioProcessorValueTreeState state;

//PluginProcessor.cpp

, state(\*this, nullptr, Identifier("params"), { })

***BUILD HERE***

Declare **AudioProcessorValueTreeState**named state in **PluginProcessor.h**. Add constructor in AudioProcessor Constructor list & parameters to StateTree in constructor.



std::make\_unique<AudioParameterFloat>(  
 "thresh",   
 "Threshold",   
 NormalisableRange<float>(-60.0f, 20.0f, 0.01f),   
 10.0f),

Creates unique pointer to **AudioParameterFloat**with identifier *thresh*, name *Threshold*, numberRange -60.0f -> 20.0f with spacing 0.01f, default value 10.0f

***BUILD HERE***

Add the following **AudioParameterFloat**parameters using the same method.

“ratio” 1.0f -> 20.0f  
“knee” 0.0f -> 24.0f  
“attack” 0.01f -> 500.0f  
“release” 0.01f -> 2000.0f

A screenshot of a computer

Description automatically generated with medium confidence

 std::make\_unique(  
 "thresh",   
 "Threshold",   
 NormalisableRange(-60.0f, 20.0f, 0.01f),   
 10.0f),  
 std::make\_unique(  
 "ratio",  
 "Ratio",   
 NormalisableRange(1.0f, 20.0f, 0.01f),   
 2.0f),  
 std::make\_unique(  
 "knee",   
 "KneeWidth",   
 NormalisableRange(0.0f, 24.0f, 0.01f),   
 0.0f),  
 std::make\_unique(  
 "attack",   
 "Attack",   
 NormalisableRange(0.01f, 500.0, 0.01f),   
 100.0f),  
 std::make\_unique(  
 "release",   
 "Release",   
 NormalisableRange(0.01f, 2000.0f, 0.01f),   
 500.0f)

***BUILD HERE***

Now our statetree is setup in **PluginProcessor**we need to pass it into **PluginEditor.**

A screen shot of a computer program

Description automatically generated with low confidence

In **PluginEditor.h**, declare pointer to an **AudioProcessorValueTreeState** called params.

Also change the constructor to receive an **AudioProcessorValueTreeState**object.

A screenshot of a computer program

Description automatically generated with medium confidence

In **PluginEditor.cpp** edit constructor to receive state and point **params** to the state object.

A screen shot of a computer program

Description automatically generated with low confidence

Back in **PluginProcessor.cpp**change editor constructor to include passing statetree.

***BUILD HERE***

Now we passed stateTree to editor we need to create sliders to control values in tree.

In **PluginEditor.h** create this typedef at top of file.

*typedef AudioProcessorValueTreeState::SliderAttachment SliderAttachment;*

This saves writing AudioProcessorValueTreeState::SliderAttachment every time.

A picture containing text, software, multimedia software, graphics software

Description automatically generated

Declare these sliders, labels and sliderattachments in **PluginEditor.h**. A slider attachment will allow values in the stateTree to be changed by a slider.

Slider threshSlider, slopeSlider, kneeSlider, attackSlider, releaseSlider;  
Label threshLabel, slopeLabel, kneeLabel, attackLabel, releaseLabel;  
std::unique\_ptr<SliderAttachment> threshAttachment, slopeAttachment, kneeAttachment, attackAttachment, releaseAttachment;

Add following to **PluginEditor,**declare private in header file as it won’t be called elsewhere.

void CompressorAudioProcessorEditor::addSlider(String name, String labelText, Slider& slider, Label& label, std::unique\_ptr<SliderAttachment>& attachment) {

addAndMakeVisible(slider);

attachment.reset(new SliderAttachment(params, name, slider));

label.setText(labelText, dontSendNotification);

label.attachToComponent(&slider, true);

addAndMakeVisible(label);

}

***BUILD HERE***

Use this to quickly add sliders, labels and attachments to the GUI.

A screen shot of a computer screen

Description automatically generated with low confidence

We put this code in Editor constructor to add sliders to GUI & attach them to stateTree.

addSlider("thresh", "Threshold", threshSlider, threshLabel, threshAttachment);  
addSlider("ratio", "Ratio", slopeSlider, slopeLabel, slopeAttachment);  
addSlider("knee", "Knee", kneeSlider, kneeLabel, kneeAttachment);  
addSlider("attack", "Attack", attackSlider, attackLabel, attackAttachment);  
addSlider("release", "Release", releaseSlider, releaseLabel, releaseAttachment);

Finally set the bounds of the slider components in the **resized()**function.

A picture containing text, software, multimedia software, graphics software

Description automatically generated

threshSlider.setBounds(100, 0, 200, 50);  
slopeSlider.setBounds(100, 50, 200, 50);  
kneeSlider.setBounds(100, 100, 200, 50);  
attackSlider.setBounds(100, 150, 200, 50);  
releaseSlider.setBounds(100, 200, 200, 50);

We’re finished coding the **PluginEditor** but we’re not using the values in **PluginProcessor.**

A screen shot of a computer

Description automatically generated with medium confidence

//PluginProcessor.h  
std::atomic<float> \*threshParam, \*slopeParam, \*kneeParam, \*attackParam, \*releaseParam;  
//PluginProcessor.cpp -> PrepareToPlay  
threshParam = state.getRawParameterValue("thresh");  
scopeParam = state.getRawParameterValue("ratio");  
kneeParam = state.getRawParameterValue("knee");  
attackParam = state.getRawParameterValue("attack");  
releaseParam = state.getRawParameterValue("release");

***BUILD HERE***

Declare these pointers in header file and set them to values in stateTree in prepareToPlay.

Pointers don’t directly store float value. They point to memory address where value stored.

We can pass these values into our CompressSample function. We can do maths on attack and release times to convert from seconds to milliseconds and from linear to time scale.

A picture containing text, software, multimedia software, graphics software

Description automatically generated

float at = 1 - std::pow(MathConstants<float>::euler, ((1 / getSampleRate()) \* -2.2f) / (\*attackParam / 1000.0f));  
float rt = 1 - std::pow(MathConstants<float>::euler, ((1 / getSampleRate()) \* -2.2f) / (\*releaseParam / 1000.0f));  
for (int i = 0; i < buffer.getNumSamples(); i++) {  
 for (int channel = 0; channel < getTotalNumOutputChannels(); channel++) {   
 auto\* data = buffer.getWritePointer(channel);   
 Compressor\* comp = &allCompressors.getReference(channel);  
 data[i] = comp->compressSample(data[i], \*threshParam, \*slopeParam, at, rt, \*kneeParam);  
 }  
}

***BUILD HERE***

Now add code to save & recall StateTree, by changing two functions in **PluginProcessor.cpp.**

A picture containing text, software, multimedia software, graphics software

Description automatically generated

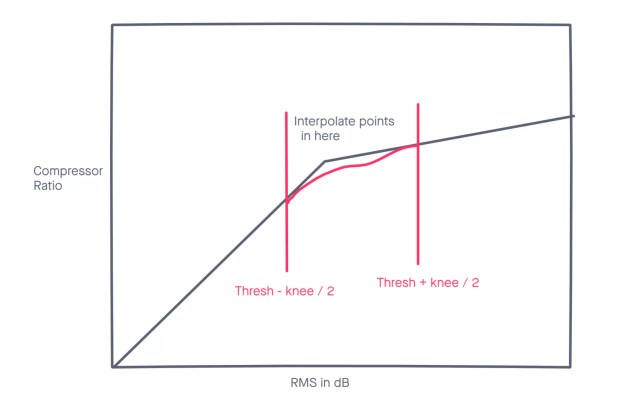
void CompressorAudioProcessor::getStateInformation(MemoryBlock& destData)  
{  
 auto stateTree = state.copyState();  
 std::unique\_ptr<XmlElement> xml(stateTree.createXml());  
 copyXmlToBinary(\*xml, destData);  
}  
void CompressorAudioProcessor::setStateInformation(const void\* data, int sizeInBytes)  
{  
 std::unique\_ptr<XmlElement> xmlState(getXmlFromBinary(data, sizeInBytes));  
 if (xmlState.get() != nullptr && xmlState->hasTagName(state.state.getType()))

state.replaceState(ValueTree::fromXml(\*xmlState));

}

Now Compressor parameters saved when you stop using it & recalled when back.

**Adding a Soft Knee (OPTIONAL)**



Hard knee means when an RMS is detected above threshold, ratio reset instantaneously. Create soft knee by introducing area where slope changed along curve. Gives compressor more natural, subtle effect on the sound.

There are many ways to interpolate between two points but we use lagrange interpolation. First we write a function to do this in **Compressor.cpp.**

A screenshot of a computer program

Description automatically generated with medium confidence

float Compressor::interpolatePoints(float\* xPoints, float\* yPoints, float detectedValue) {  
 float result = 0.0f;  
 int n = 2;  
  for (int i = 0; i < n; i++){  
 float term = 1.0f;  
 for (int j = 0; j < n; j++{  
 if (j != i) {  
 term \*= (detectedValue-xPoints[j]) / (xPoints[i]-xPoints[j]);  
 }  
 }

result += term \* yPoints[i];

}  
 return result;  
}

This function takes array of 2 xpoints, 2 y points & approximates value between them. This value will replace our **slope** value.

A screenshot of a computer

Description automatically generated with medium confidence

if (knee > 0 && dbRMS > (thresh - knee / 2.0) && dbRMS < (thresh + knee / 2.0)) {  
   float kneeBottom = thresh - knee /    2.0, kneeTop = thresh + knee / 2.0;  
 float xPoints[2], yPoints[2];  
 xPoints[0] = kneeBottom;  
 xPoints[1] = kneeTop;  
 xPoints[1] = std::fmin(0.0f, kneeTop);  
 yPoints[0] = 0.0f;  
 yPoints[1] = slope;  
 slope = interpolatePoints(&xPoints[0], &yPoints[0], thresh);  
 thresh = kneeBottom;  
}

This block of code sets up some variables for the interpolation function we just wrote.

First it checks the current **rms**value is in the knee zone, between *thresh – knee / 2.0* and *thresh + knee / 2.0.*

Then we make two arrays of points,   
xPoints holds db values of bottom and top of knee region.  
yPoints hold slope values.

Once these arrays are set up we interpolate a new slope value and set the threshold of the compressor to the lower knee value.

Rebuild the plugin and you should see/hear a difference when you change the knee slider.