

An Introduction to Building an audio effect plug-in in JUCE

Brief Intro to REAL-TIME AUDIO PROCESSING

Non-REAL-TIME

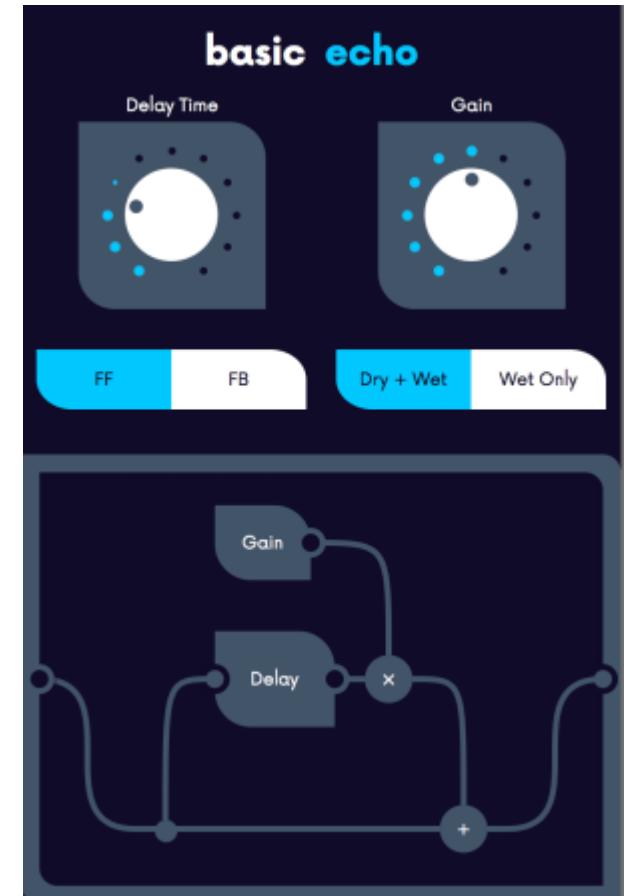
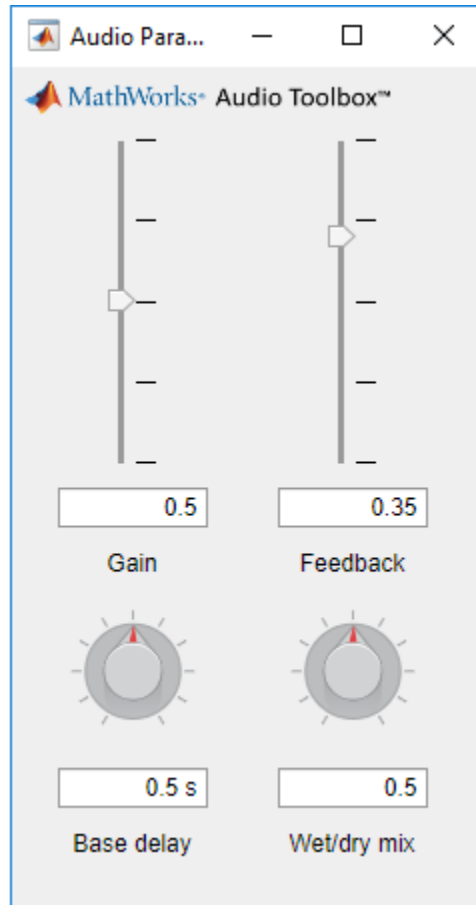
```
Fs = 48000;           % Sampling Frequency
T = 1 / Fs;           % Sampling Step Size
duration = 0.01;      % Signal duration
f0 = 100;             % Signal Frequency
t = T:T:duration;
u = sin(2*pi*f0*t);    % Signal
%% Signal processing stage
out = 2*u;

figure(1)
plot(t,out)
hold on
plot(t,u)
```

What if the whole signal is not available?

Process data in chunks
using the buffer

1. Load data to the buffer.
2. Process the buffered data.
3. Output the buffer.



An Introduction to



JUCE

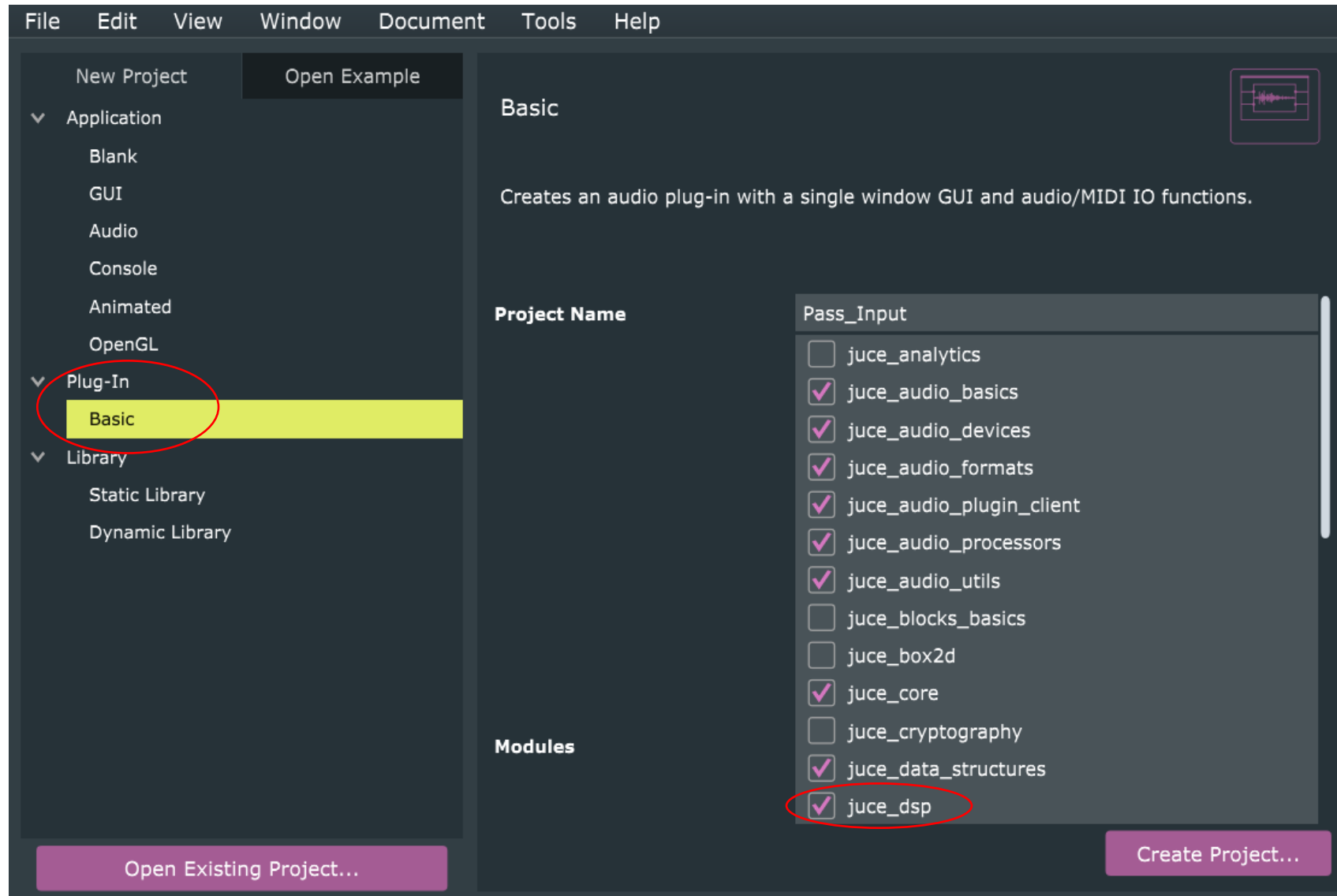
What is JUCE

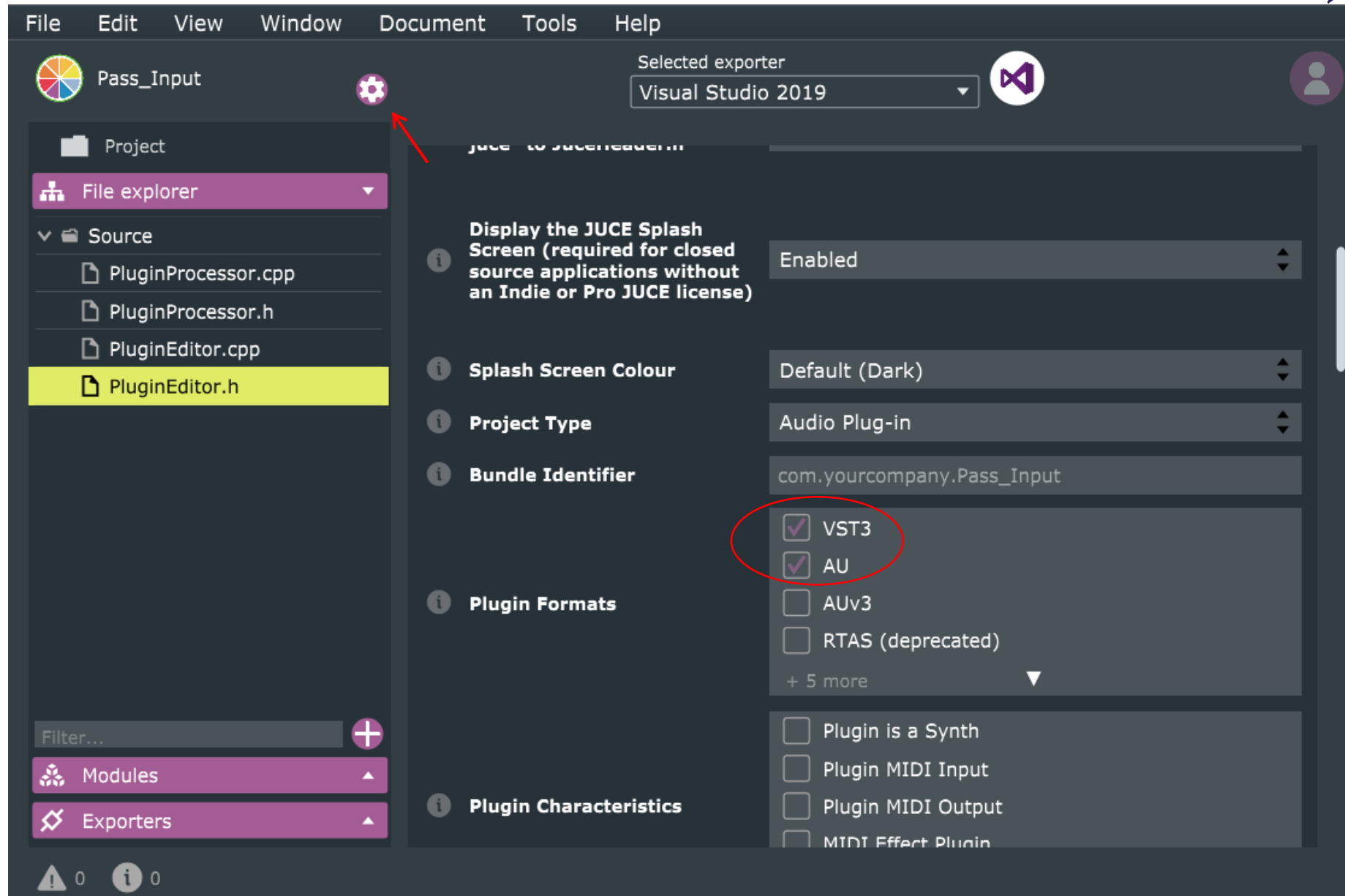
Partially open-source framework for C++ application and plugin development.

- It is mainly used for or its GUI and plug-in libraries.
- Has useful dsp libraries and the learning curve is gentle.

JUCE was used to build Max.

Building a plug-in for audio effect in JUCE





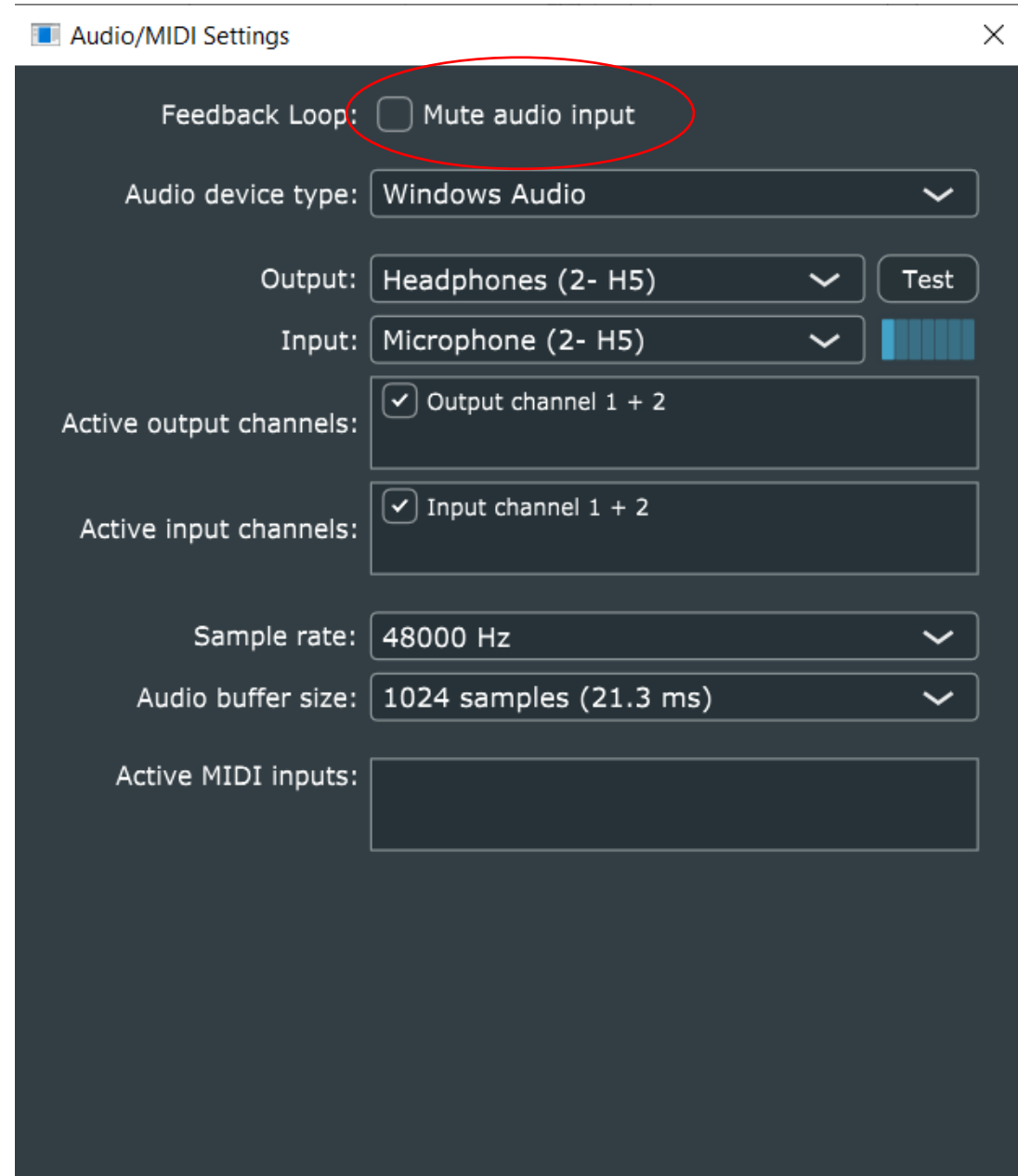
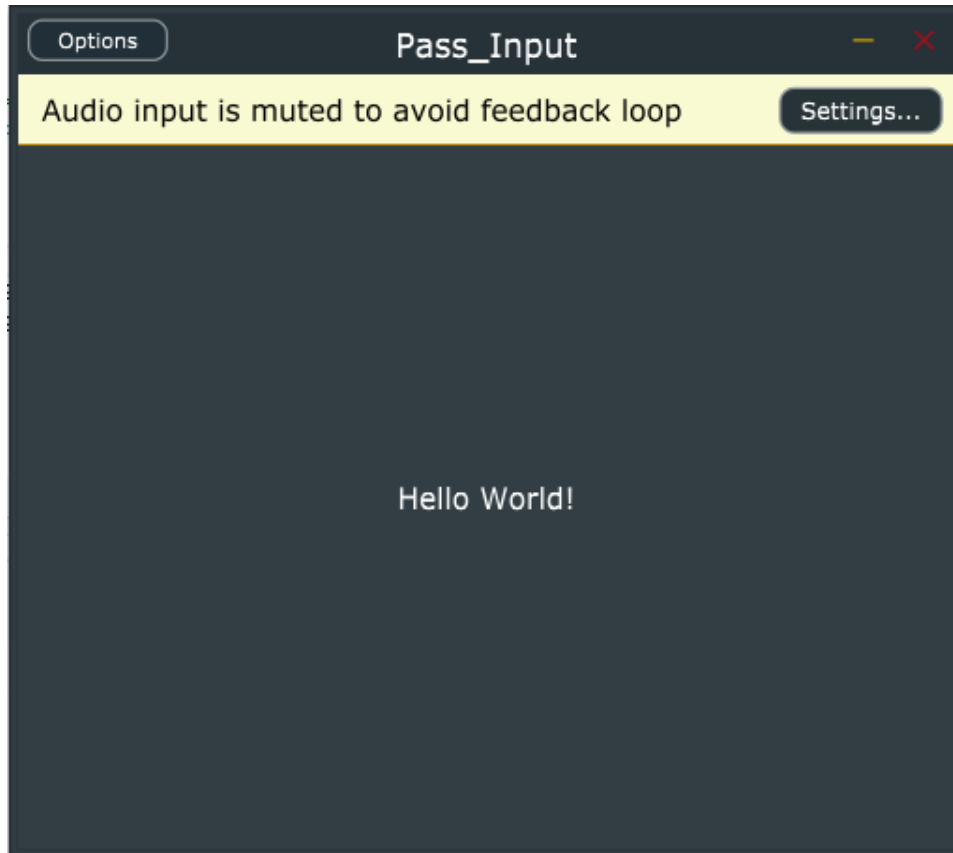
We don't need to change anything in these

```
Pass_InputAudioProcessor::Pass_InputAudioProcessor()
Pass_InputAudioProcessor::~~Pass_InputAudioProcessor()
const juce::String Pass_InputAudioProcessor::getName() const
bool Pass_InputAudioProcessor::acceptsMidi() const
bool Pass_InputAudioProcessor::producesMidi() const
bool Pass_InputAudioProcessor::isMidiEffect() const
double Pass_InputAudioProcessor::getTailLengthSeconds() const
int Pass_InputAudioProcessor::getNumPrograms()
int Pass_InputAudioProcessor::getCurrentProgram()
void Pass_InputAudioProcessor::setCurrentProgram (int index)
const juce::String Pass_InputAudioProcessor::getProgramName (int index)
void Pass_InputAudioProcessor::changeProgramName (int index, const juce::String& newName)
void Pass_InputAudioProcessor::releaseResources()
bool Pass_InputAudioProcessor::isBusesLayoutSupported (const BusesLayout& layouts) const
bool Pass_InputAudioProcessor::hasEditor() const
juce::AudioProcessorEditor* Pass_InputAudioProcessor::createEditor()
void Pass_InputAudioProcessor::updateFilter(bool realFreq)
void Pass_InputAudioProcessor::getStateInformation (juce::MemoryBlock& destData)
void Pass_InputAudioProcessor::setStateInformation (const void* data, int sizeInBytes)
juce::AudioProcessor* JUCE_CALLTYPE createPluginFilter()
```

processBlock

```
void Pass_InputAudioProcessor::processBlock (juce::AudioBuffer<float>& buffer, juce::MidiBuffer& midiMessages)
{
    auto totalNumInputChannels = getTotalNumInputChannels();
    auto totalNumOutputChannels = getTotalNumOutputChannels();
    /***/
    for (int i = totalNumInputChannels; i < totalNumOutputChannels; ++i) // clear the buffer
        buffer.clear(i, 0, buffer.getNumSamples());
    /***/
    for (int channel = 0; channel < buffer.getNumChannels(); ++channel){
        for (int sample = 0; sample < buffer.getNumSamples(); ++sample){
            vin = buffer.getSample(channel, sample);    // get the input sample
            vout = vin;
            buffer.setSample(channel, sample, vout);    // set the output sample value
        }
    }
}
```

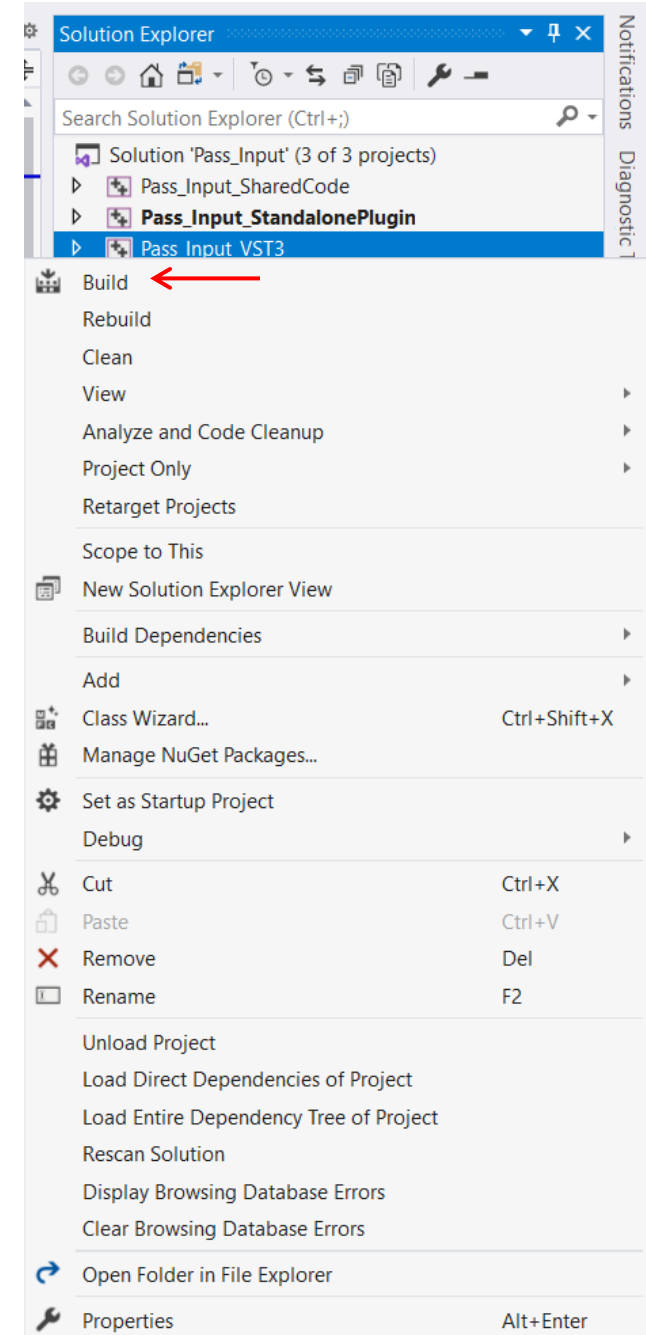
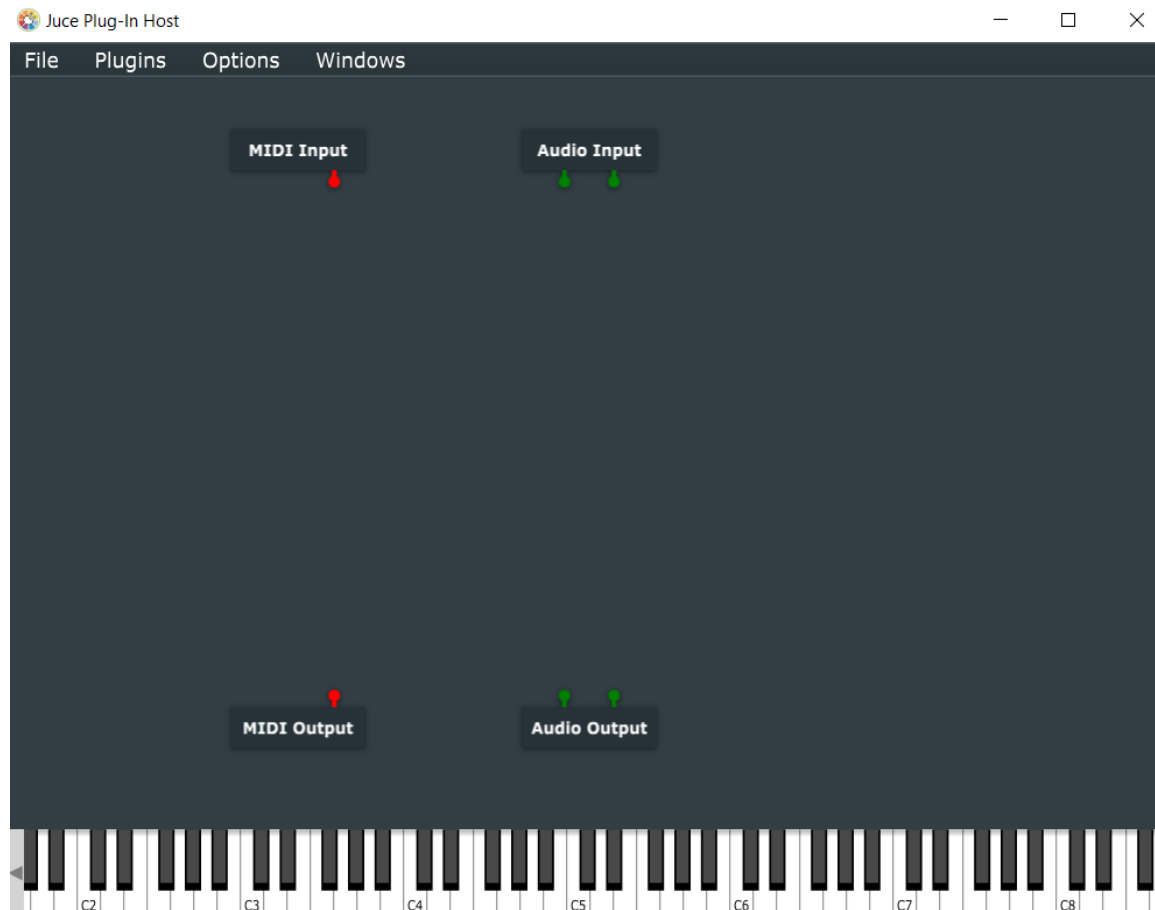
That's it!!



Build a VST3 or (AU) plugin and test it

Audio Plugin Host

JUCE\extras\AudioPluginHost\Builds\VisualStudio2019\x64\Release\App



Let's control the gain



```
Pass_InputAudioProcessorEditor:: Pass_InputAudioProcessorEditor(Pass_InputAudioProcessor& p,  
juce::AudioProcessorValueTreeState& vts) ←
```

```
: AudioProcessorEditor(&p), audioProcessor(p), audioTree(vts)
```

```
{
```

```
setSize(450, 250);
```

```
controlGain.setColour(0x1001400, juce::Colour::fromRGBA(0x00, 0x40, 0x00, 0x80)); } *
```

```
controlGain.setColour(0x1001700, juce::Colour::fromRGBA(0x00, 0x00, 0x00, 0x00)); }
```

```
controlGain.setSliderStyle(juce::Slider::LinearHorizontal);
```

```
controlGain.setTextBoxStyle(juce::Slider::TextBoxLeft, false, 40, 20);
```

```
addAndMakeVisible(controlGain);
```

```
sliderAttachGain.reset(new juce::AudioProcessorValueTreeState::SliderAttachment(audioTree, "controlGain_ID", ←  
controlGain));
```

```
labelGain.setText(("Input Gain"), juce::dontSendNotification);
```

```
labelGain.setFont(juce::Font("Slope Opera", 16, 0));
```

```
labelGain.setColour(juce::Label::textColourId, juce::Colour::fromRGBA(0x40, 0x40, 0x80, 0xff));
```

```
addAndMakeVisible(labelGain);
```

```
}
```

- Add juce::AudioProcessorValueTreeState audioTree; to the header file under public class definitions

```
Pass_InputAudioProcessorEditor::~Pass_InputAudioProcessorEditor()
{
    → sliderAttachR.reset();
}
void Pass_InputAudioProcessorEditor::paint(juce::Graphics& g)
{
    // (Our component is opaque, so we must completely fill the background with a solid colour)
    g.fillAll(juce::Colours::white);
    // set the current drawing colour to black
    g.setColour(juce::Colours::black);

    // set the font size and draw text to the screen
    g.setFont(15.0f);
    g.setFont(juce::Font("Slope Opera", 35.0f, 1));
    g.drawFittedText("Pass Input", getLocalBounds(), juce::Justification::centred, 1);
} void Pass_InputAudioProcessorEditor::resized()
{
    [ controlGain.setBounds(140, getHeight() - 20, getWidth() - 140, 20);
      labelGain.setBounds(0, getHeight() - 20, 140, 20);
    ]
}
```

```
#include <JuceHeader.h>
#include "PluginProcessor.h"
class Pass_InputAudioProcessorEditor : public juce::AudioProcessorEditor
{
public:
    Pass_InputAudioProcessorEditor (Pass_InputAudioProcessor&, juce::AudioProcessorValueTreeState&);
    ~Pass_InputAudioProcessorEditor() override;

    //=====
    void paint (juce::Graphics&) override;
    void resized() override;

private:
    // This reference is provided as a quick way for your editor to
    // access the processor object that created it.
    Pass_InputAudioProcessor& audioProcessor;
    juce::AudioProcessorValueTreeState& audioTree;
    juce::Slider controlGain;
    juce::Label labelGain;
    std::unique_ptr <juce::AudioProcessorValueTreeState::SliderAttachment> sliderAttachGain;
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (Pass_InputAudioProcessorEditor)
};
```

```

Pass_InputAudioProcessor::Pass_InputAudioProcessor()
#ifdef JucePlugin_PreferredChannelConfigurations
    : AudioProcessor (BusesProperties()
        #if ! JucePlugin_IsMidiEffect
        #if ! JucePlugin_IsSynth
            .withInput  ("Input",  juce::AudioChannelSet::stereo(), true)
        #endif
            .withOutput ("Output", juce::AudioChannelSet::stereo(), true)
        #endif
    ),
    audioTree(*this, nullptr, juce::Identifier("PARAMETERS"),
        {
            std::make_unique<juce::AudioParameterFloat>("controlGain_ID", "ControlGain", juce::NormalisableRange<float>
                (<del>0.01, 1.0</del>, 0.01), 0.5)
        })
    #endif
    {
        audioTree.addParameterListener("controlGain_ID", this);
        controlledGain = 0.5;
    }
    void Pass_InputAudioProcessor::prepareToPlay (double sampleRate, int samplesPerBlock)
    { controlledGain = 0.5;}

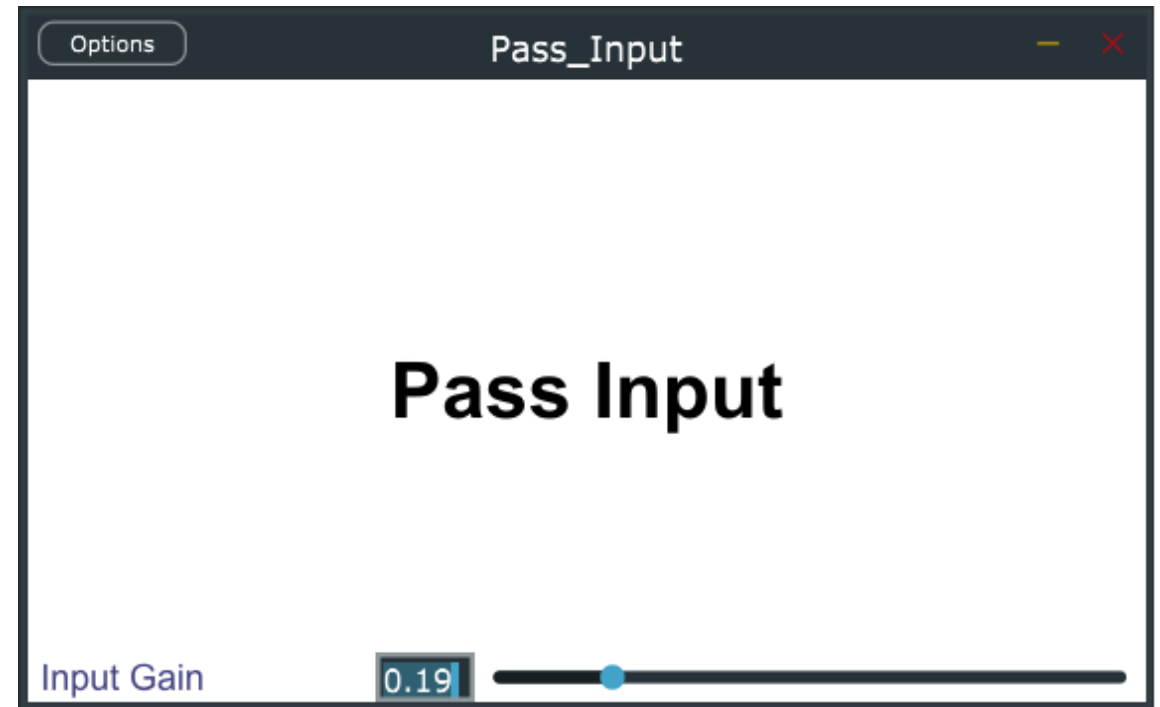
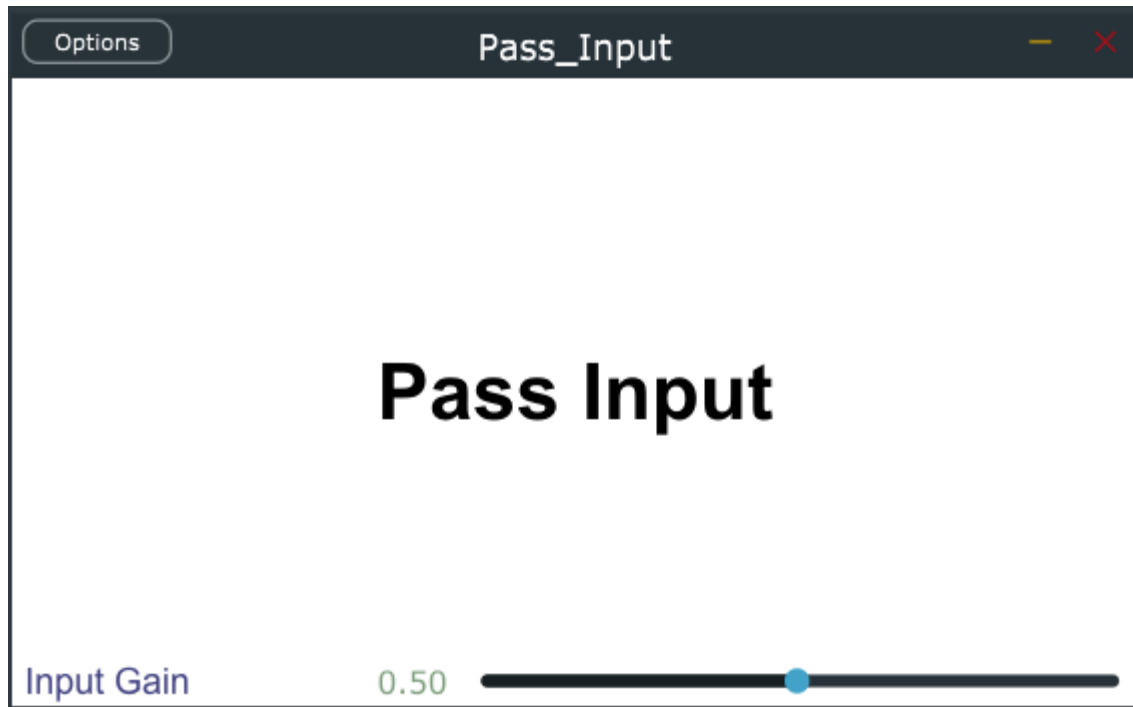
```



```
void Pass_InputAudioProcessor::parameterChanged(const juce::String& parameterID, float newValue)
{
    if (parameterID == "controlGain_ID") {
        controlledGain = newValue; // controlledGain is defined in the header file as a private variable
    }
}

void Pass_InputAudioProcessor::processBlock (juce::AudioBuffer<float>& buffer, juce::MidiBuffer& midiMessages)
{
    auto totalNumInputChannels = getTotalNumInputChannels();
    auto totalNumOutputChannels = getTotalNumOutputChannels();
    /*****
    for (int i = totalNumInputChannels; i < totalNumOutputChannels; ++i) // clear the buffer
        buffer.clear(i, 0, buffer.getNumSamples());
    *****/
    for (int channel = 0; channel < buffer.getNumChannels(); ++channel){
        for (int sample = 0; sample < buffer.getNumSamples(); ++sample){
            vin = buffer.getSample(channel, sample);    // get the input sample
            vout = vin * controlledGain;
            buffer.setSample(channel, sample, vout);    // set the output sample value
        }
    }
}
```

Wow



Oversampling simplified

1. Up-sample the input
2. Apply steep low pass at Nyquist of original signal
3. Process the signal
4. Apply low pass again
5. Down-sample the output

```

#include "PluginProcessor.h"
#include "PluginEditor.h"

//=====
Pass_InputAudioProcessor::Pass_InputAudioProcessor()
#ifndef JUCEPLUGIN_PREFERREDCHANNELCONFIGURATIONS
    : AudioProcessor (BusesProperties()
        #if ! JUCEPLUGIN_IsMidiEffect
        #if ! JUCEPLUGIN_IsSynth
            .withInput  ("Input",  juce::AudioChannelSet::stereo(), true)
        #endif
            .withOutput ("Output", juce::AudioChannelSet::stereo(), true)
        #endif
    ),
        audioTree(*this, nullptr, juce::Identifier("PARAMETERS"),
{std::make_unique<juce::AudioParameterFloat>("controlGain_ID", "ControlGain", juce::NormalisableRange<float>(0.01, 1.0, 0.01), 0.5)
        }),
        → lowPassFilter(juce::dsp::IIR::Coefficients< float >::makeLowPass((48000.0 * 4.0), 20000.0))

#endif
{
→ oversampling.reset(new juce::dsp::Oversampling<float>(2, 2, juce::dsp::Oversampling<float>::filterHalfBandPolyphaseIIR,
true));
}
Pass_InputAudioProcessor::~~Pass_InputAudioProcessor()
{
→ oversampling.reset();
}

```

prepareToPlay

```
void Pass_InputAudioProcessor::prepareToPlay (double sampleRate, int samplesPerBlock)
{
    oversampling->reset();
    oversampling->initProcessing(static_cast<size_t> (samplesPerBlock));

    juce::dsp::ProcessSpec spec;
    spec.sampleRate = sampleRate * 4;
    spec.maximumBlockSize = samplesPerBlock * 3;
    spec.numChannels = getTotalNumOutputChannels();

    lowPassFilter.prepare(spec);
    lowPassFilter.reset();

    controlledGain = 0.5;
}
```

```

void Pass_InputAudioProcessor::processBlock (juce::AudioBuffer<float>& buffer, juce::MidiBuffer& midiMessages)
{
    auto totalNumInputChannels = getTotalNumInputChannels();
    auto totalNumOutputChannels = getTotalNumOutputChannels();

    /*****/
    for (int i = totalNumInputChannels; i < totalNumOutputChannels; ++i) // clear the buffer
        buffer.clear(i, 0, buffer.getNumSamples());

    {
        juce::dsp::AudioBlock<float> blockInput(buffer);
        juce::dsp::AudioBlock<float> blockOutput = oversampling->processSamplesUp(blockInput);
        updateFilter(0);
        lowPassFilter.process(juce::dsp::ProcessContextReplacing<float>(blockOutput));
    }

    /*****/
    for (int channel = 0; channel < blockOutput.getNumChannels(); ++channel){
        for (int sample = 0; sample < blockOutput.getNumSamples(); ++sample){
            vin = blockOutput.getSample(channel, sample);    // get the input sample
            vout = vin * controlledGain;
            blockOutput.setSample(channel, sample, vout);    // set the output sample value
        }
    }

    {
        updateFilter(0);
        lowPassFilter.process(juce::dsp::ProcessContextReplacing<float>(blockOutput));
        oversampling->processSamplesDown(blockInput);
    }

    {
        updateFilter(1);
        lowPassFilter.process(juce::dsp::ProcessContextReplacing<float>(blockInput));
    }
}

```

```
void Pass_InputAudioProcessor::updateFilter(bool realFreq)
{
    double frequency;

    if (realFreq) frequency = 48e3;
    else frequency = 48e3 * 4;

    *lowPassFilter.state =
    *juce::dsp::IIR::Coefficients<float>::makeLowPass(frequency, 20000.0);
}
```

Newton Raphson Diode Clipper

Fixed Point

```
v(n) = R*T/(R*C+T)*(gdExp(-vTemp) - gdExp(vTemp))  
+ T/(R*C+T)*newU(n) + R*C/(R*C+T)*v(n-1);
```

Newton Raphson

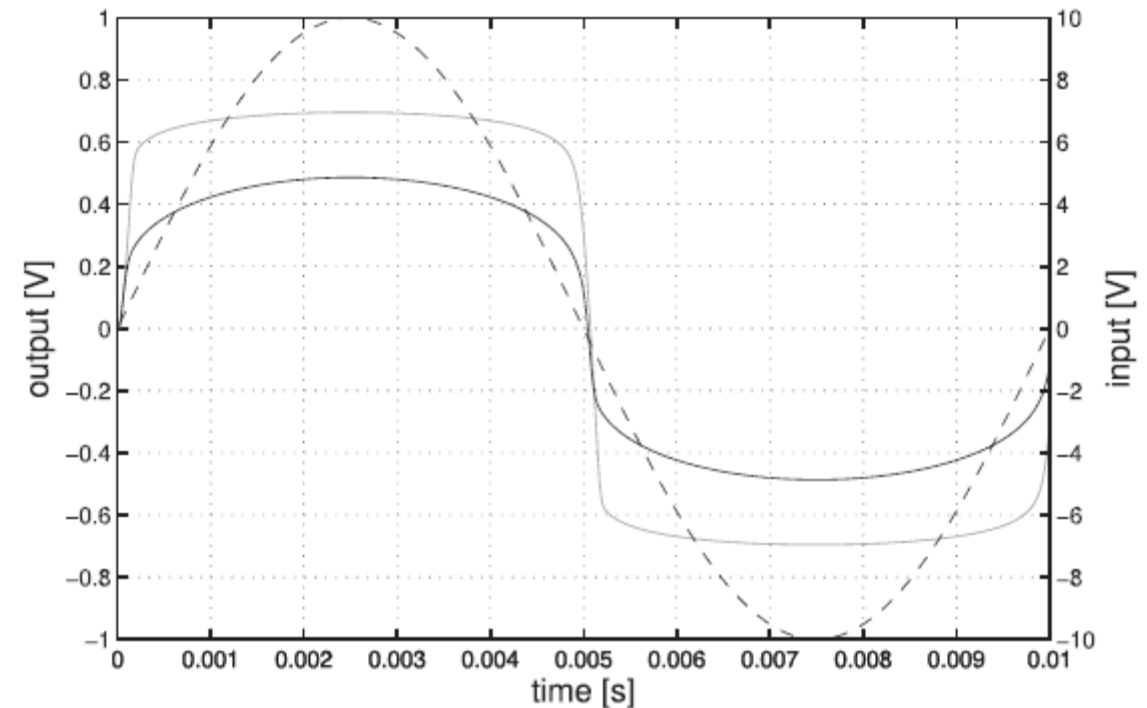
```
vNom = T*vTemp*R*gdExpDiff(-vTemp) + T*R*gdExp(-vTemp) +  
v(n - 1)*R*C + T*(gdExpDiff(vTemp)*R*vTemp -  
R*gdExp(vTemp) + newU(n));
```

```
vDenom = T*R*gdExpDiff(vTemp) + T*R*gdExpDiff(-vTemp) +  
R*C + T;
```

```
v(n) = vNom/vDenom;
```

Using exponential diode characteristic

$$g_D(v) = I_D(e^{\frac{v}{2V_E}} - 1)$$



Pass_InputAudioProcessor → DiodeClipperNRAudioProcessor

PluginProcessor.h

public:

...

double gdExp(double vc);

double gdExpDiff(double vc);

double limiter(double val);

private:

double controlledGain;

double Id, C, Ve, Vp, R, err;

double Fs, T;

double vNom, vDenom;

float vin;

float vout, voutTemp, voutOld;

...

PluginProcessor.cpp



```
...
double DiodeClipperNRAudioProcessor::gdExp(double vc)
{
    return Id * (std::exp(vc / (2 * Ve)) - 1);
}
double DiodeClipperNRAudioProcessor::gdExpDiff(double vc)
{
    return (Id * std::exp(vc / (2 * Ve))) / (2 * Ve);
}
double DiodeClipperNRAudioProcessor::limiter(double val)
{
    if (val < -1){
        val = -1;
        return val;
    } else if (val > 1){
        val = 1;
        return val;
    } else return val;
}
...
```

```
DiodeClipperNRAudioProcessor::DiodeClipperNRAudioProcessor()
#ifdef JUCE_PLUGIN_PREFERRED_CHANNEL_CONFIGURATIONS
    : AudioProcessor(BusesProperties()
#ifdef ! JUCE_PLUGIN_IS_MIDI_EFFECT
#ifdef ! JUCE_PLUGIN_IS_SYNTH
        .withInput("Input", juce::AudioChannelSet::stereo(), true)
#endif
#endif
        .withOutput("Output", juce::AudioChannelSet::stereo(), true)
#ifdef ! JUCE_PLUGIN_IS_MIDI_EFFECT
#ifdef ! JUCE_PLUGIN_IS_SYNTH
        .withInput("Input", juce::AudioChannelSet::stereo(), true)
#endif
#endif
    ),
    audioTree(*this, nullptr, juce::Identifier("PARAMETERS"),
        {
            std::make_unique<juce::AudioParameterFloat>("controlGain_ID", "ControlGain", juce::NormalisableRange<float>(0.0, 130.0, 0.1), 1.0)
        }
    ),
    lowPassFilter(juce::dsp::IIR::Coefficients<float>::makeLowPass((48000.0 * 16.0), 20000.0))
#ifdef ! JUCE_PLUGIN_IS_MIDI_EFFECT
#ifdef ! JUCE_PLUGIN_IS_SYNTH
    {
        oversampling.reset(new juce::dsp::Oversampling<float>(2, 4,
juce::dsp::Oversampling<float>::filterHalfBandPolyphaseIIR, false));
        audioTree.addParameterListener("controlGain_ID", this);
        controlledGain = 1.0;
    }
#endif
#endif
}
```

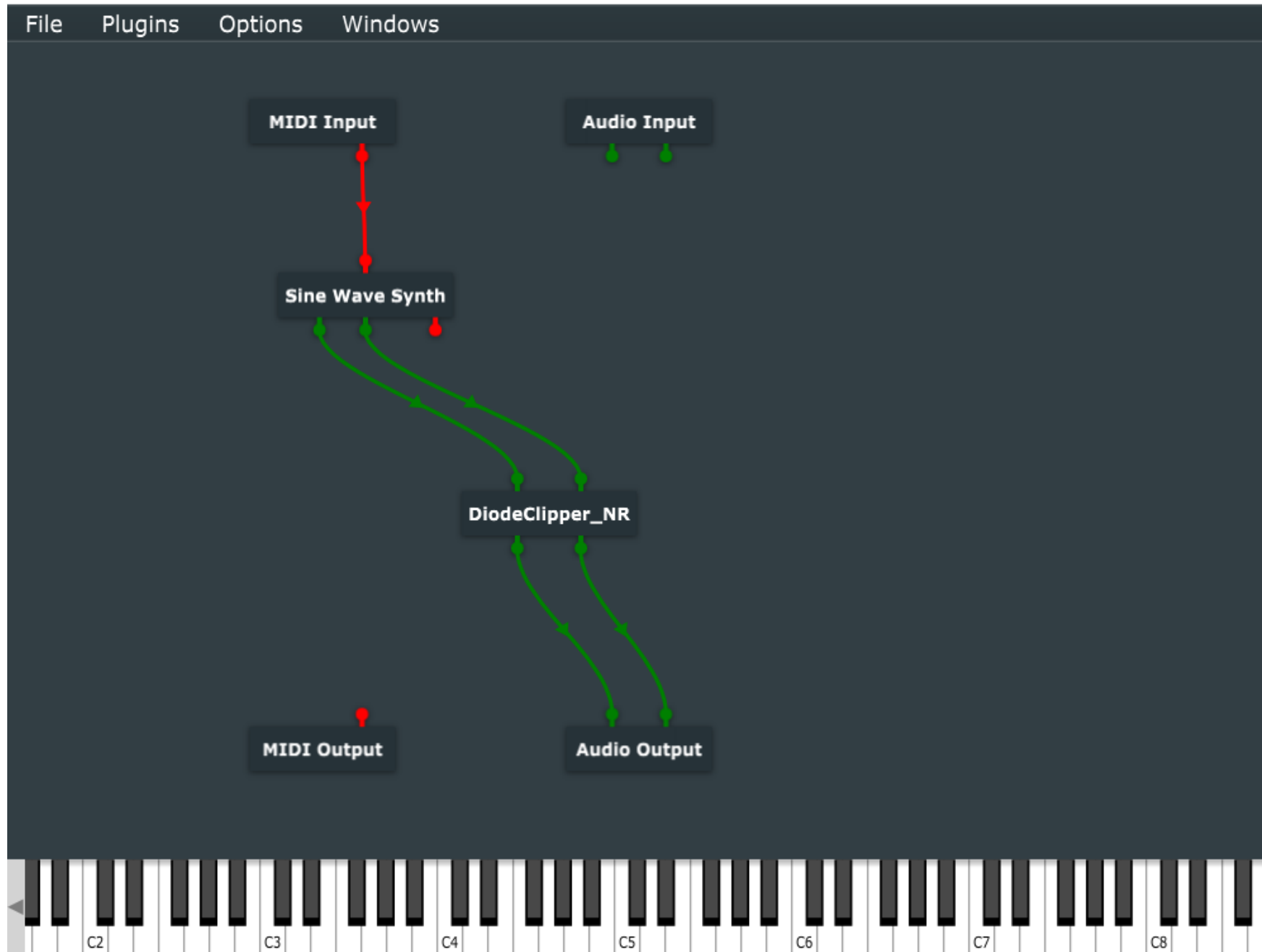
```
void DiodeClipperNRAudioProcessor::prepareToPlay(double sampleRate, int samplesPerBlock)
{
    ...
    spec.sampleRate = sampleRate * 16;
    spec.maximumBlockSize = samplesPerBlock * 15;
    spec.numChannels = getTotalNumOutputChannels();
    lowPassFilter.prepare(spec);
    lowPassFilter.reset();
    Fs = sampleRate;
    err = 0.1e-3; // err for stopping iterations
    T = 1 / Fs;
    C = 100e-9;
    R = 1e3;
    Vp = 0.17;
    Ve = 0.023;
    Id = 2.52e-9;
    controlledGain = 1.0;
    voutOld = 0;
}
```

```
void DiodeClipperNRAudioProcessor::updateFilter()
{
    double frequency;
    if (realFreq) frequency = 48e3;
    else frequency = 48e3 * 16;
    *lowPassFilter.state =
    *juce::dsp::IIR::Coefficients<float>::makeLowPass(frequency, 20000.0);
}
```

PluginProcessor.cpp processBlock



```
...
for (int channel = 0; channel < blockOutput.getNumChannels(); ++channel){
    for (int sample = 0; sample < blockOutput.getNumSamples(); ++sample){
        voutTemp = 1;
        vout = 0;
        vin = controlledGain * blockOutput.getSample(channel, sample);
        while (std::abs(voutTemp - vout) > err) {
            voutTemp = vout;
            vNom = T * voutTemp * R * gdExpDiff(-voutTemp) + T * R * gdExp(-voutTemp) + voutOld * R * C + T * (gdExpDiff(voutTemp) * R *
voutTemp - R * gdExp(voutTemp) + vin);
            vDenom = T * R * gdExpDiff(voutTemp) + T * R * gdExpDiff(-voutTemp) + R * C + T;
            vout = vNom / vDenom;
        }
        voutOld = vout;
        vout = limiter(vout);
        blockOutput.setSample(channel, sample, vout);
    }
}
...
```



Useful links

- Audio Processor Tree
<https://docs.juce.com/master/classAudioProcessorValueTreeState.html>
- Oversampling class
https://docs.juce.com/master/classdsp_1_1Oversampling.html
- Sliders (Colour IDs) <https://docs.juce.com/master/classSlider.html>
- Filters https://docs.juce.com/master/structdsp_1_1FilterDesign.html
- Github link with Pass_Input JUCE files
https://github.com/titas2001/JUCE_Audio_Effect_Core