POLITECNICO MILANO 1863

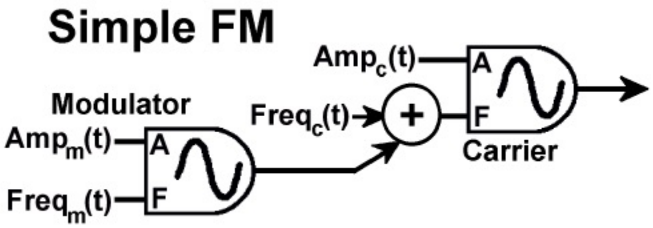
ISPL Image and Sound Processing Lab

JUCE

Second Part

Ex 4 : FM Synth

- You are going to implement a **very primitive** MIDI synth.
- It receives MIDI messages, and converts the noteOn and noteOff messages into audio signals.
- It implements a FM synthesis
- The user should controll modulation index and modulation frequency



$$y(t) = a(t) \cdot \sin(\phi(t))$$

$$\phi(t) = 2\pi[f_0 t + I(t)\sin(2\pi f_m t)]$$

$y(t)$ is the synthesized signal

$a(t)$ is the time varying amplitude of the signal

f_0 is the carrier frequency

$I(t)$ is the modulation index

f_m is the modulation frequency

FM Synth

Tips:

- use as an example the `Oscillator` plugin
- probably you will need 6 variables: phase, amplitude and frequency of the carrier + phase, index and frequency of the modulator
- from the MIDI input intercept the `noteOn` MIDI messages and transform the MIDI note in frequency (Hertz) using the method `juce::MidiMessage::getMidiNoteInHertz`
- when a `noteOff` MIDI message is received you can shut off the synthesis by controlling its amplitude

FM Synth

The GUI part is omitted here but the logic is the always same (not classic sliders but rotary sliders).

- in the `Processor` header define a struct type, called `FMData:` it contains the parameters that control

1 the sinusoidal FM synthesizer + some global variables

```
#define SAMPLE_RATE   (44100)
#ifndef M_PI
#define M_PI  (3.14159265)
#endif
```

FM Synth

- add as Processor private members the variables we will need for FM synthesis

2

```
float mod_phase;
float mod_freq;
int mod_index;

float phase;
float amp;
float car_freq;
```

- in the method `prepareToPlay()` the variables needed later:

3

```
amp = 1.0;
phase = 2.0;
car_freq = 0.0;
mod_freq = 0.0;
mod_phase = 1.0;
mod_index = 0.0;
```

FMSynth

4
- in the method `processBlock()` first we process the incoming MIDI messages and then we compute sample by sample the FM synthesis result using the parameters obtained from both that MIDI message (the note) and both the GUI (modulation frequency, modulation index and carrier frequency)
  - for each message in the MIDI buffer, if the MIDI message is a `NoteOn` update the value of the note, if the MIDI message is a `NoteOff` set the amplitude value to 0

FM Synth

```
float mod;
juce::MidiMessage m;
int time;
for (juce::MidiBuffer::Iterator i (midiMessages); i.getNextEvent (m, time);){
     if (m.isNoteOn()) {
          amp = 0.1;
          car_freq = m.getMidiNoteInHertz(m.getNoteNumber());
     }
     else if (m.isNoteOff()) {
          data.amp = 0;
     }
     else if (m.isAftertouch()) {
      }
     else if (m.isPitchWheel()) {
     }
}
```

FM Synth

- get two pointers for writing in the output channels

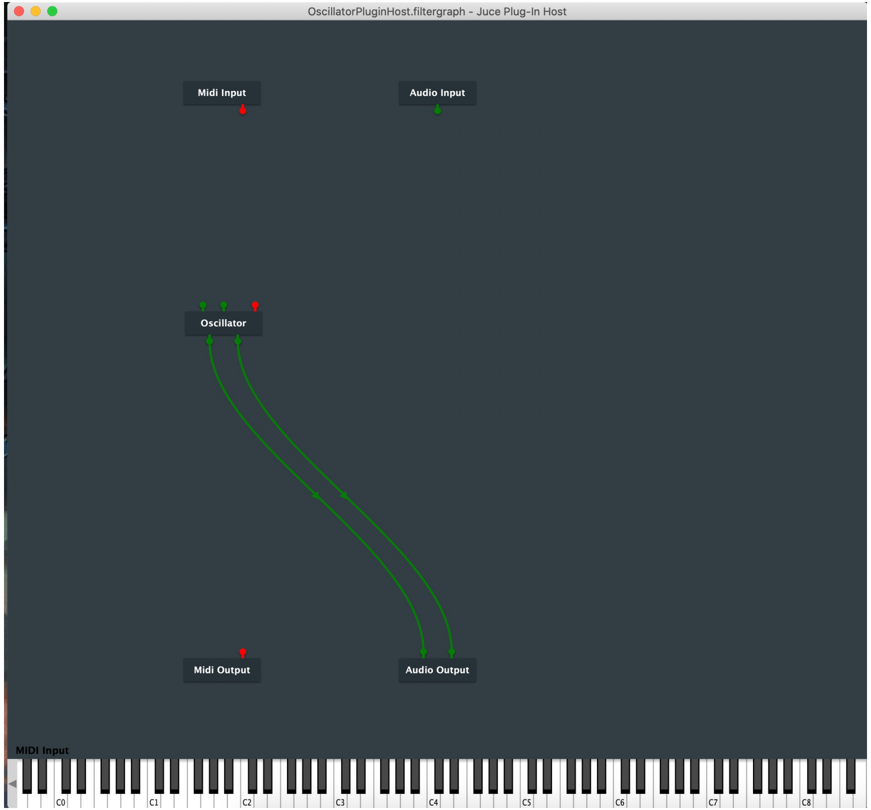
```
float* channelDataL = buffer.getWritePointer(0);
float* channelDataR = buffer.getWritePointer(1);

int numSamples = buffer.getNumSamples();
```

FMSynth

- sample by sample (`for` cycle) we update the values for the argument of both the sinusoids, the modulator and the carrier. Then we compute the output sample.

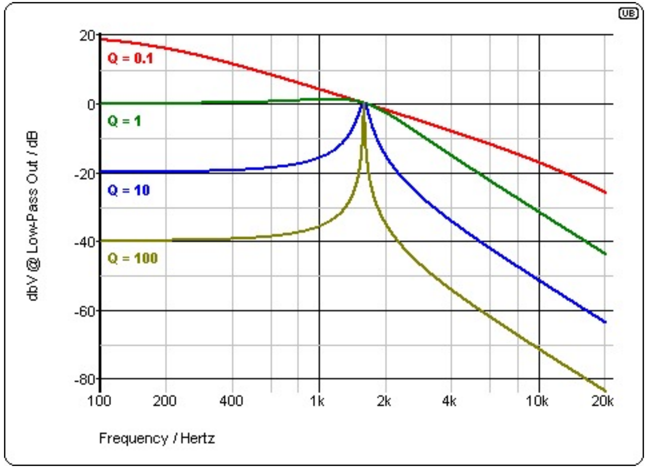
```
for (int i = 0; i < numSamples; ++i){
    mod = mod_index * (float) sin((double) mod_phase);

    channelDataL[i] = amp * (float) sin ((double) phase + mod);
    channelDataR[i] = channelDataL[i];

    phase +=  (float) ( M_PI * 2. *( ((double) car_freq  / (double)
SAMPLE_RATE)));
    if( phase >= M_PI * 2. ) phase -= M_PI * 2.;

    mod_phase += (float) ( M_PI * 2. * ((double) mod_freq / (double)
SAMPLE_RATE) );
    if( mod_phase >= M_PI * 2. ) mod_phase -= M_PI * 2.;
}
```

Audio Plugin Host configuration



Low Pass Filter

We are going to implement a simple low pass filter, controlling the cut-off frequency and the Q factor.



We are going to use two advanced features of JUCE:

- `AudioProcessorValueTreeState` class for sharing parameters between Editor and Processor
- `DSP` module for implementing a Low Pass filter

Create a plug-in project named `LowPassFilter` and include `juce_dsp` in the modules

Audio Processor Value Tree State

This class contains a `ValueTree` that is used to manage an `AudioProcessor`'s entire state.

It has its own internal class of parameter object that is linked to values within its `ValueTree`, and which are each identified by a `string ID`.

You can get access to the underlying `ValueTree` object via the state member variable, so you can add extra properties to it as necessary.

It also provides some utility child classes for connecting parameters directly to GUI controls like sliders.

This class allows the automation of plugin parameters in DAW framework and the saving/loading of the plugin state.

JUCE DSP module

DSP module allows to define and use DSP processes quickly.

A number of examples are present in `JUCE/examples/DSP` subfolder.

In the modules we can find:

- elements that can be used to create complex processing (`DelayLine`, `Interpolator`, `Panner`, ...)
- self-contained DSP blocks (`Chorus`, `Compressor`, `Limiter`, ...)

Low Pass Filter: Value Tree State

Let's start by adding as member of the Audio Processor an instance of `AudioProcessorValueTreeState` class and a function that create and associate all parameters returning a `ParameterLayout` object

```
juce::AudioProcessorValueTreeState apvts;
juce::AudioProcessorValueTreeState::ParameterLayout
createParameters();
```

Low Pass Filter: Value Tree State

Now we define in the cpp file the function `createParameters` : we create a vector of unique pointers to `RangeAudioParameter` , an abstract class for different types for `AudioParameter`; then we fill the vector by instantiating two `AudioParameterFloat`, one for the cut-off frequency and one for the quality factot parameter, indicating the range and the default value. Finally we return two iterators pointing to the first and one to the last element of the vector.

2

```
juce::AudioProcessorValueTreeState::ParameterLayout LowPassFilterAudioProcessor::createParameters()
{
      std::vector<std::unique_ptr<juce::RangedAudioParameter>> parameters;
      parameters.push_back (std::make_unique<juce::AudioParameterFloat> ("FREQ", "CutOff Frequency", 50.0f,
20000.0f, 500.0f));
      parameters.push_back (std::make_unique<juce::AudioParameterFloat> (”Q", ”Q Factor", 0.1f, 1.0f, 0.5f));
      return { parameters.begin(), parameters.end() };
}
```

Low Pass Filter: Value Tree State

Now we can instantiate the `AudioProcessorValueTreeState` object in the initialization list of the `AudioProcessor` constructor: we call the constructor indicating as `ParameterLayout` the result of the `createParameters()` function:

3

```
LowPassFilterAudioProcessor::LowPassFilterAudioProcessor()
#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
                       ), apvts(*this, nullptr, "Parameters", createParameters()),
```

Low Pass Filter: Value Tree State

Now we move on to the `Editor`. As usual we declare our `Sliders` and `Labels` in the header

4
```
juce::Slider qualitySlider;
juce::Slider frequencySlider;
juce::Label frequencyLabel;
juce::Label qualityLabel;
```

In the `Editor` constructor we specify the properties of the `Sliders` and `Labels`:

5
```
qualitySlider.setSliderStyle (juce::Slider::SliderStyle::LinearVertical);
qualitySlider.setTextBoxStyle (juce::Slider::TextEntryBoxPosition::TextBoxBelow, true, 100, 20);
qualitySlider.setRange (0.0, 1.0, 0.1);
addAndMakeVisible (qualitySlider);
qualityLabel.setText(”Q",juce::dontSendNotification);
addAndMakeVisible (resonanceLabel);
frequencySlider.setSliderStyle (juce::Slider::SliderStyle::LinearVertical);
frequencySlider.setTextBoxStyle (juce::Slider::TextEntryBoxPosition::TextBoxBelow, true, 100, 20);
frequencySlider.setRange (50.0, 20000.0, 100.0);
addAndMakeVisible (frequencySlider);
frequencyLabel.setText("Frequency",juce::dontSendNotification);
addAndMakeVisible (frequencyLabel);
```

Low Pass Filter: Value Tree State

In the `Editor` resized() we specify the location of `Sliders` and `Labels`:

6
```
frequencySlider.setBounds(10,80,100,100);
frequencyLabel.setBounds(10,50,130,20);

qualitySlider.setBounds(200,80,100,100);
qualityLabel.setBounds(200,50,130,20);
```

Now we have to link somehow the GUI components to the parameters of the `AudioProcessorValueTree.` We are going to use the class `AudioProcessorValueTreeState::SliderAttachment` using again a unique pointer template. We declare in the `Editor` header two members:

7
```
std::unique_ptr<juce::AudioProcessorValueTreeState::SliderAttachment> qualitySliderAttachment;

std::unique_ptr<juce::AudioProcessorValueTreeState::SliderAttachment> frequencySliderAttachment;
```

Low Pass Filter: Value Tree State

In the `Editor` constructor we actually allocate memory for these two pointers and we link the GUI components to `AudioProcessorValueTreeState` parameters, accessible through `audioProcessor` member of the `Editor`

8

```
frequencySliderAttachment =
std::make_unique<juce::AudioProcessorValueTreeState::SliderAttachment>(audioProcessor.apvts,
"FREQ", frequencySlider);

qualitySliderAttachment =
std::make_unique<juce::AudioProcessorValueTreeState::SliderAttachment>(audioProcessor.apvts, ”Q",
qualitySlider);
```

Low Pass Filter: DSP module

Now let's focus on the DSP module provided by JUCE: remember to include the module in the Projucer project!

What we are going to do is to use the class `dsp::IIR:Filter`. The user will be able to control the cut-off frequency and the quality factor using special methods of the filter using methods of the class `dsp::IIR:Coefficients`. To allow multichannel processing we are going to use a `ProcessorDuplicator` to "wrap" our filter

9

```
juce::dsp::ProcessorDuplicator <juce::dsp::IIR::Filter<float>, juce::dsp::IIR::Coefficients
<float>> lowPassFilter;

float lastSampleRate;
```

As we did before in the constructor initialization list we add the instantiation for IIR filter. We call the `makeLowPass` method of the `Coefficients` template using default values.

10

```
lowPassFilter(juce::dsp::IIR::Coefficients<float>::makeLowPass(44100, 20000.0f, 0.1))
```

Low Pass Filter: DSP module

If we look at the documentation for the `dsp::IIR:Filter` class we notice that there are three main methods: `reset()`, `prepare()` and `process()`; in the `prepareToPlay` method of the `AudioProcessor` we ”prepare” also the IIR filter. This is done by exploiting the `ProcessSpec` structure to store some basic specifications for the IIR Filter. By using the `reset()` function we simply reset the processing pipeline

```
lastSampleRate = sampleRate;
juce::dsp::ProcessSpec spec;
spec.maximumBlockSize = samplesPerBlock;
spec.sampleRate = sampleRate;
spec.numChannels = getTotalNumOutputChannels();
lowPassFilter.prepare(spec);
lowPassFilter.reset();
```

Low Pass Filter: DSP module

Now we are ready to actually apply the filtering algorithm. In the `processBlock` method we will update the filter coefficients and we will apply the filtering to the audio buffer. We will use the `dsp::AudioBlock` class template, which simply “wraps” the audio buffer to be input for the filter.

We will update the frequency and quality factosr taken from the `AudioProcessorValueTreeState`.

and the `dsp::ProcessContextReplacing` template. In practice the process method will read from the input, process it and replace the content in the input buffer (then streamed to the output).

11

```
juce::dsp::AudioBlock <float> block (buffer);

float freq = *apvts.getRawParameterValue("FREQ");
float quality = *apvts.getRawParameterValue("Q");
*lowPassFilter.state = *juce::dsp::IIR::Coefficients<float>::makeLowPass(lastSampleRate, freq,
quality);

lowPassFilter.process(juce::dsp::ProcessContextReplacing<float> (block));
```