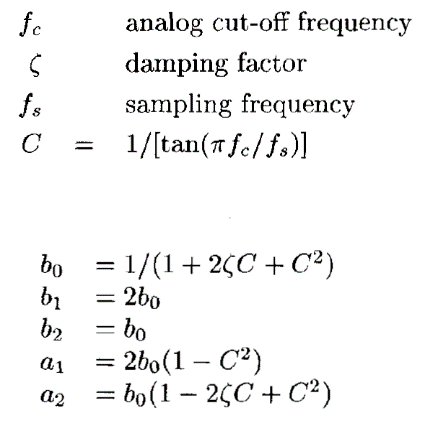
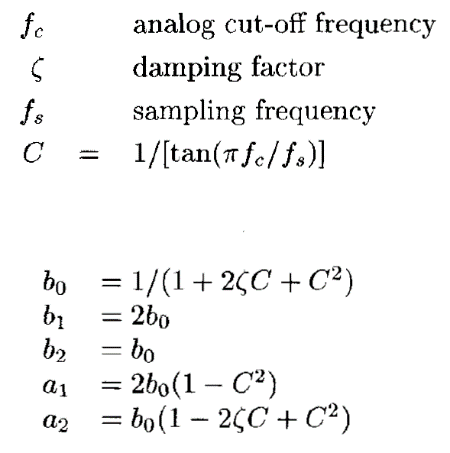
Low-Pass Filter VST Plugin

# The Algorithm

The filter algorithm I used for this plugin was taken from Digital Audio Effects (DAFx). The filter coefficients are calculated as follows





*Filter coefficients, DAFx, Chapter 2.2, page 34*

Where the damping factor is calculated as and the difference equation is:

*Second-order filter difference equation, DAFx, Chapter 2.2, page 34*

Where is the filtered output signal and is the input signal.

# Implementation in JUCE

The bulk of the code was generated by ROLI’s Projucer software. Projucer creates four C++ files two .cpp and two .h files. The application of each generated file is detailed below.

## PluginProcessor

The PluginProcessor.cpp file is where the bulk of the audio processing is done. Only some of the premade functions were edited in this file. The first of these is the LowPassFilterAudioProcessor() which sets the initial values for the Q and frequency.

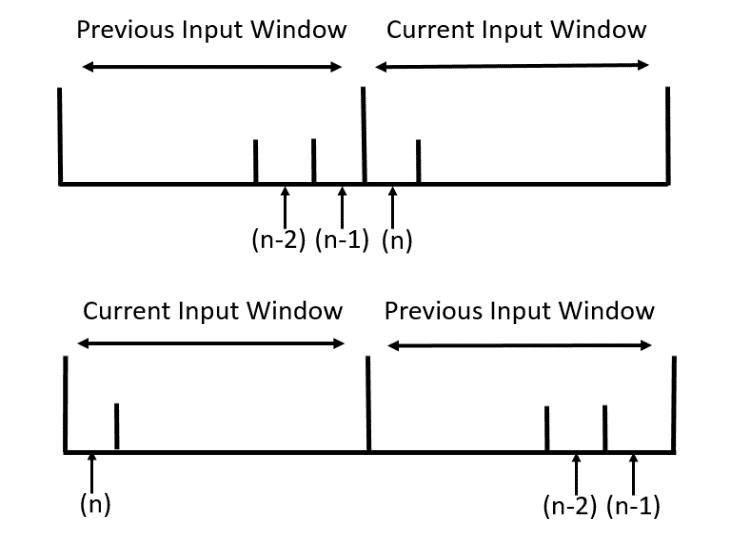
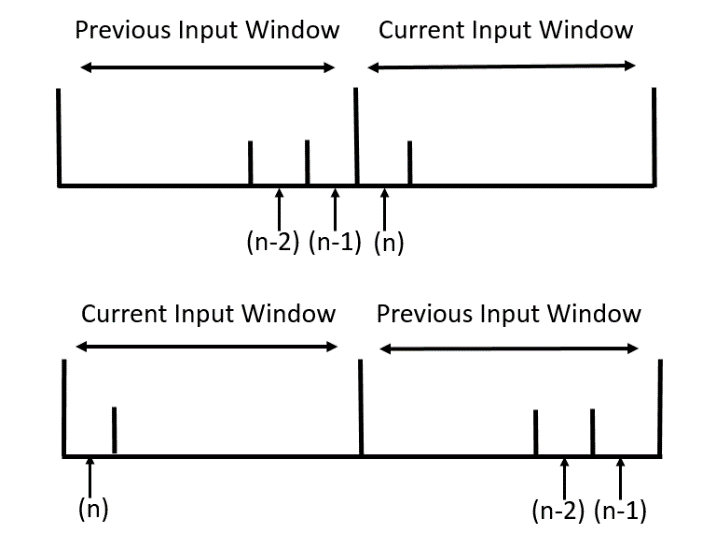
Two functions were added for the purpose of this project calculateCoefficients() and resizeBuffers(AudioSampleBuffer& buffer). The first of these functions calculates the filter coefficients and is called whenever the cut-off frequency, Q factor or sampling rate are changed and is always run the first time the plugin loads. The second changes the size of the circular buffer used in the processing loop if the size of the input window is changed.

### processBlock

All of the DSP is done in the function processBlock(AudioSampleBuffer& buffer, MidiBuffer& midiMessages) this function allows the engineer to apply any processing to the input audio.

For the purposes of this project I used circular audio buffers to store both filtered and unfiltered audio data. JUCE has a class used to hold only audio data, the AudioSampleBuffer. This acts similarly to a C++ array but with some notable differences. One difference used in this project is the function setSize which allows the buffer to be dynamically resized.

Two circular buffers were used in this process block, one held the unfiltered data and the other the filtered data. Since the input audio data in JUCE is given as a square window of data the circular buffer held data from the current input window and the previous input window. This was necessary due to the filter equation requiring samples at time (n-1) and (n-2). If only the current input or filtered audio window is used then when n = 0 there would be no record of data for (n-1) and (n-2) to access.



*The circular buffer holding the input audio at 2 points in time*

The filter equation used for this plugin requires samples from the filtered audio as well as the input data. The same circular buffering technique was therefore applied to the filtered data.

The filter equation defined in the previous section is then applied and stored on a sample by sample basis through the use of a for loop that increments up to length of the input buffer.

The PluginProcessor.h file was edited to define all global audio buffers, pointers, variables and functions used in PluginProcessor.cpp.

## PluginEditor

The PluginEditor.cpp allows the engineer to define all aspects of the GUI. The first function LowPassFilterAudioProcessorEditor(LowPassFilterAudioProcessor& p) defines each aspect of the two sliders used to control the frequency and Q values for the filter. The range of value, the style and label are all set in this function. For the frequency slider the range of values are arranged logarithmically with 3000Hz as the mid-point with the function setSkewFactorFromMidPoint(3000.0).

The sliderValueChanged(Slider\* slider) connects the values outputted from the slider objects to the variables defined in PluginProcessor.cpp. resized() sets the position and size of each slider within the plugin window. Finally paint(Graphics& g) sets the colours and fonts used in GUI.

PluginEditor.h has been edited to define the QFactorSlider and FrequencySlider Slider objects.

# Test

The plugin was initially tested against a stream of live audio through the use of the JUCE Audio Plugin Host. It was then tested in Reaper against pre-recorded audio of varying sample and bit rates. The range of values for the Frequency and Q sliders were decided by comparison against commercially available EQ plugins.

Tests were performed to see that the resonance at high values of Q were suitable and that lower Q values effectively dampened the filtered audio. Some aliasing was found in lower sampling rates (such as 22.05kHz) and was remedied with an if statement in the processing loop that forced the filter frequency down below (a similar effect to this is seen in Cockos’ ReaEQ). Ideally the slider shown in the GUI would adjust its range of available frequencies whenever the sampling rate was changed, however this was not feasible in the available timeframe.