Copyright © 2013 Will Pirkle

An Adaptive Filtering Plug-In using the FFT
Will Pirkle

This short addendum is a follow up to App Note 2 (Multithreading an FFT for Parallel Processing in a Plug-In) which used a FFT to display the audio spectrum on the LED meters in RackAFX. This App Note describes an adaptive system where a side-chained FFT is used to find the highest amplitude frequency component then sets the center frequency of a notch filter according to that. This is a very bare-bones Plug-In; the FFT is not windowed and there is no FFT overlap. This Plug-In is to get you started with adaptive FFT side-chaining.

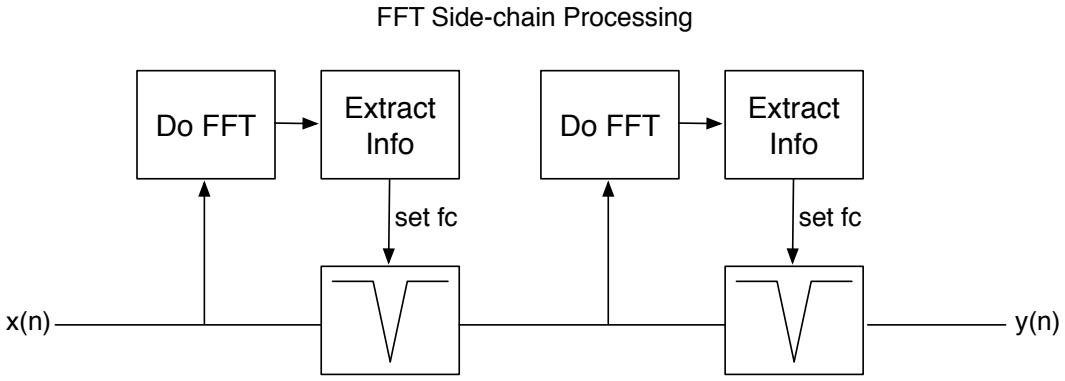

Figure 3.1: Using the FFT in a side-chain to analyze the audio and adaptively set the notch frequency

This Plug-In is built around the MTFFT Plug-In from App Note 2; make sure you have gone through that code and understand how it works. We won't need to modify the FFT thread callback function but we will modify the doFFT() and processAudioFrame() functions.

The Notch Filter

$$\theta_c = 2\pi f_c / f_s$$
$$\mu = 10^{Gain(dB)/20}$$
$$\zeta = \frac{4}{1+\mu}$$
$$\beta = 0.5\frac{1-\zeta\tan\left(\theta_c/2Q\right)}{1+\zeta\tan\left(\theta_c/2Q\right)}$$
$$\gamma = \left(0.5+\beta\right)\cos\theta_c$$
$$a_0 = 0.5-\beta$$
$$a_1 = 0.0$$
$$a_2 = -(0.5-\beta)$$
$$b_1 = -2\gamma$$
$$b_2 = 2\beta$$
$$c_0 = \mu - 1.0$$
$$d_0 = 1.0$$

The notch filter I am using here is the Parametric (Peaking) filter from my book Chapter 6. The user controls the depth of the notch (-dB or cut) and the Q. The FFT analysis chooses the center frequency. When the plug-in starts up, the filter is turned off. It is not enabled until the first FFT block is processed. The notch filter is based on the biquad structure so I use the built-in CBiquad object.

The coefficient equations for the filter are on the left; it uses the Audio Specific (Modified BiQuad) as described in my book.

- Two filters are needed, one for the left and the other for the right.
- The double-buffering system from the MTFFT project is employed
- a boolean flag is used to turn on the filters after the first FFT

Copyright © 2013 Will Pirkle

The following are added to the .h file:

```
// notch filter
CBiQuad m_LeftFilter;
CBiQuad m_RightFilter;

// start flag
bool m_bStartFilter;

// specific to the parametric EQ
float m_c0;
float m_d0;

// function for the FFT thread to call for setting the filter params
void calculateFilterCoeffs(float fc, float Q);

// returns -1 if failure
//             or Bin of max amplitude if success
inline int normalizeBufferEx(double* pInputBuffer, UINT uBufferSize)
{
      int nRet = -1;
      double fMax = 0;

      for(UINT j=0; j<uBufferSize; j++)
      {
            if((fabs(pInputBuffer[j])) > fMax)
            {
                  nRet = j;
                  fMax = fabs(pInputBuffer[j]);
            }
      }
      // normalize
      if(fMax > 0)
      {
            for(UINT j=0; j<uBufferSize; j++)
                  pInputBuffer[j] = pInputBuffer[j]/fMax;
      }

      return nRet;
}
```

The calculateFilterCoeffs() simply implements the design equations above; it is called from doFFT().

The normalizeBufferEx() method is based on the built-in normalizeBuffer() function except that it returns the index of the maximum value in the array. This is used to calculate the new notch center frequency; this happens in the doFFT() function after the FFT has been taken. You can see the code below which calculates the bin frequency and sets the new center frequency of the filter. It also prints the frequency to the RackAFX status window.

Copyright © 2013 Will Pirkle

Inside doFFT()

```
// normalize and find peak bin
//
// normalizeBuffeEx is implemented in the .h file for this object
int nMaxBin = normalizeBufferEx(&m_dFFT_Magnitude[0], FFT_LEN/2);

// adaptive filter control
if(nMaxBin > 0)
{
	float fMaxFreq = nMaxBin*((float)m_nSampleRate/FFT_LEN);

	// update filter!
	this->calculateFilterCoeffs(fMaxFreq, m_fNotchQ);

	char *f = floatToString(fMaxFreq, 2);
	char* p = addStrings("fMaxFreq = ", f);
	this->sendStatusWndText(p);

	delete [] p;
	delete [] f;

	// NOW OK to start Filter up...
	// NOTE: this is sticky-on; reset in prepForPlay
	m_bStartFilter = true;
}
```

Inside processAudioFrame()

This is where the data is buffered for the FFT in the same way as for the MTFFT project except that I am summing the left and right inputs into mono and feeding that to the FFT; this way only one FFT needs to be taken - obvious room for improvement. The MTFFT project simply passed the input to the output; in this project is it filtered through the two Biquads.

```
// now accumulating STEREO MIX for FFT
if(bAccumulatePoints)
{
	float stereoMix = 0.5*pInputBuffer[0] + 0.5*pInputBuffer[1];

	// if processing A, load up B
	if(m_bProcessFFFTonBufferA) //starts at 1024, 1025, 1026...
		m_dAudioDoubleBuffer[m_nSampleCount + FFT_LEN] = stereoMix;
	else // load up bufferA starts at 0, 1, 2
		m_dAudioDoubleBuffer[m_nSampleCount] = stereoMix;

	// inc the counter
	m_nSampleCount++;
}

// left-mono
if(m_bStartFilter) // see book for info on parametric (Modified BiQuad)
	pOutputBuffer[0] = m_c0*m_LeftFilter.doBiQuad(pInputBuffer[0]) +
	                   m_d0*pInputBuffer[0];
else
	pOutputBuffer[0] = pInputBuffer[0];
```

Copyright © 2013 Will Pirkle

```
// Mono-In, Stereo-Out (AUX Effect)
if(uNumInputChannels == 1 && uNumOutputChannels == 2)
      pOutputBuffer[1] = pOutputBuffer[0];

// Stereo-In, Stereo-Out (INSERT Effect)
if(uNumInputChannels == 2 && uNumOutputChannels == 2)
{
      if(m_bStartFilter)
            pOutputBuffer[1] = m_c0*m_RightFilter.doBiQuad(pInputBuffer[1]) +
                               m_d0*pInputBuffer[1];
      else
            pOutputBuffer[1] = pInputBuffer[1];
}
```

Build and test the Plug-In; with the Q at 0.707 (pretty wide) you will easily hear the notch moving around with the input signal; as you make the Q more narrow the effect is less obvious for music signals. If you use the RackAFX oscillator you can easily hear the adaptive notch chasing your signal around as you move the frequency slider. You will notice a slight delay - why? Because the FFT analyzes the data to find the peak frequency, but by the time that is found, the audio event has already passed by - if the double-buffering is running smoothly without overruns then we know that the FFT lags by FFT_LEN samples. We can accommodate for this using the look-ahead technique described in my Chapter 13 of my book. This can be done by placing a FFT_LEN delay in front of the filter so that the signal lines up with its analysis block.

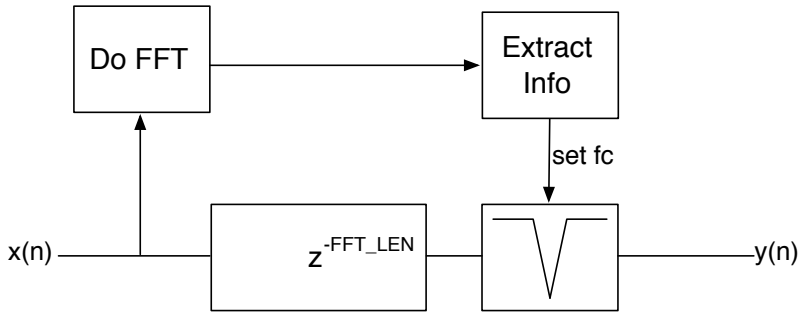

Figure 3.1: delaying the signal by FFT_LEN samples can be used as a look-ahead technique

References:

Pirkle, Will. 2012. *Designing Audio Effect Plug-Ins in C++*, Chap. 13. Focal Press.