Resampling using a FIR filter

by

Jens Hee
https://jenshee.dk

October 2016

Change log

29. September 2016

1. Document started.

23. October 2016

1. Document extended

13. November 2018

1. The example is modified to handle a varying number of input and output samples.

5. January 2019

1. The example modified to handle a varying number of input and output samples has been corrected.

13. March 2020

1. Meta data added.

Resampling

The resampling method described in the following is using a dedicated FIR filter, i.e the FIR filter is designed for a specific resampling ratio e.g. 51200/48000. First the resampling ratio must be reduced to an irreducible fraction: 51200/48000 = 16/15. The numerator is called the upsampling factor and the denominator is called the downsampling factor. The resampling can now be carried by first inserting zeroes, (upsampling factor minus one) between each sample and then reducing the sampling rate by the downsampling factor.
Without a low pass filter after upsampling aliasing is in general introduced. The exact filter specification depends on the application. For frequency analysis it is only necessary to avoid aliasing within the analysis band, for other applications it may be necessary to atenuate aliasing components up to half the resulting sampling frequency, If the frequency band of interest in the example above is 20 kHz, then the FIR filter must have a pass band of 20 kHz and give the required attenuation of aliasing components above 31.2 kHz, since no frequency components in the 20 - 31.2 kHz range will end up in the 0 - 20 kHz range after downsampling.
Since zeroes are inserted for upsampling these samples do not contribute to the summation in the FIR filter. Moreover the filter need only be applied at intervals corresponding to the downsampling factor. This leads to a considerable performance improvement. The algorithm for avoiding the unnecessary calculations is presented in the next chapters.

Filter design

The filter can be designed using Remez algorithm or by windowing. Windowing has the advantage that the stop band attenuation may be increasing with frequency giving better suppression of aliasing components, but the filter is longer than the Remez filter. Windowing may also give a filter without ripple in the pass band, which can be important when cascading filters. E.g. the Hanning window has no ripple in the passband and 60 dB/decade attenuation in the stop band.

Filter specification example upsampling:

Up sampling factor: 16
Down sampling factor: 15
Original sampling frequency: 48000
Resulting sampling frequency: 51200
Virtual intermediate sampling frequency; 768000 kHz
Pass band: 0 - 20000 Hz
Stop band: 31200 - 384000 Hz
Pass band ripple: 0.05 dB (-45 dB)
Stop band attenuation: 85 dB
A candidate using the Remez algorithm has 256 samples.

Filter specification example downsampling:

Up sampling factor: 15
Down sampling factor: 16
Original sampling frequency: 51200
Resulting sampling frequency: 48000
Pass band: 0 - 20000 Hz

Stop band: 28000 - 384000 Hz
Pass band ripple: 0.05 dB (-45 dB)
Stop band attenuation: 85 dB
A candidate using the Remez algorithm has 360 samples.

Filter algorithm

When looking at the input sequence after insertion of zeros it is clear that if the filter length is p times the upsampling factor, then the filtering requires exactly p multiplications for each output sample all other multiplicatios gives zero result. If the filter design does not give a filter length divisible by the upsampling factor, then the filter can be padded with zeroes. The filtering process can now be viewed as a FIR filtering with a filter length equal to p where the set of filter coefficients are changed for each output sample, the number of filters being equal to the upsampling factor. The order in which the subfilters are used and the use of input samples depends on the upsampling- and downsampling factors.
The algorithm process blocks of samples, where the number of samples may vary from block to block. In order to produce the same number of output samples for each input block the input buffer must have a length divisible by the downsampling factor and the output buffer must have a length divisible by the upsampling factor.
The algorithm can be used for upsampling (upsampling factor > downsampling factor) as well as for downsampling, the difference being the choice of dk and dn.

```
    public class Resampling
    {
        int upFactor;
        int downFactor;
        int subFilterLength;
        double[] coefficients;
        double[] delayLine;
        int n;
        int k;
        int dk;
        int dn;

        public Resampling(int upFactor, int downFactor)
        {
            this.upFactor = upFactor;
            this.downFactor = downFactor;
            coefficients = Coefficients();
            subFilterLength = coefficients.Length / upFactor;
            delayLine = new double[2 * subFilterLength];
            n = 0;
k = 0;
            // upsampling
            dk = 1;
            dn = upFactor - downFactor;

            // downsampling
//          int dk = downFactor / upFactor + 1;
//          int dn = dk * upFactor - downFactor;
```

```
}

public int Resample(double[] input, int nInputs, double[] output)
{
    for (int i = 0; i < subFilterLength; i++)
        delayLine[i + subFilterLength] = input[i];

    double[] data = delayLine;

    bool first = true;

    int nOutputs = 0;

    do
    {
        if (first && k >= subFilterLength)
        {
            data = input;
            k -= subFilterLength;
            first = false;
        }

        double sum = 0;
        int offset = n;
        for (int j = 0; j < subFilterLength; j++)
        {
            sum += coefficients[offset] * data[k + j];
            offset += upFactor;
        }

        output[nOutputs] = sum;

        nOutputs++;
        k += dk;
        n += dn;

        if (n >= upFactor)
        {
            n -= upFactor;
            k--;
        }

    } while (k < nInputs - subFilterLength);

    k -= nInputs - subFilterLength

    for (int i = 0; i < subFilterLength; i++)
        delayLine[i] = input[i + nInputs - subFilterLength];

    return nOutputs;
}
```

```
}
```