NEW SNDOBJ LIBRARY CLASSES FOR SINUSOIDAL MODELLING

Joseph Timoney, Victor Lazzarini† and Thomas Lysaght

Department of Computer Science,
†Music Technology Laboratory,
National University of Ireland, Maynooth.
{Jtimoney,Vlazzarini,Tlysaght}@may.ie

ABSTRACT

We present an object-oriented implementation for sinusoidal modelling based on the C++ Sound Object Library (SndObj). We outline the background to this analysis/synthesis technique and its inclusion in many well known methods of speech and music signal processing. Incorporation of such a well known technique into the SndObj library will enable the development of further audio processing techniques such as vocoding, time and pitch scaling and cross-synthesis on an object-oriented development platform.

1. SINUSOIDAL MODELLING

Sinusoidal models are based on a well known assumption that the audio signal can be represented as a sum of sine waves with time-varying amplitudes and frequencies. In the basic model, the audio signal $s(n)$ is modeled as the sum of a small number, *L*, of sinusoids:

$$s(n) = \sum_{l=1}^{L} A_l \cos(\omega_l n + \varphi_l) \qquad (1)$$

where $A_l(n)$ and $\varphi_l(n)$ are the time varying amplitude and phase of each sinusoidal component associated with the frequency track ω_l. The basic model is also known as the McAulay / Quatieri Model [1] (see Figure 1). This technique is incorporated in the SMS model [2] as well as for voice and instrumental morphing applications [3]. This basic model has also some modifications such as ABS/OLA (Analysis by Synthesis / Overlap Add) and Hybrid / Sinusoidal Noise models.

1.1. Analysis

The analysis stage estimates sets of amplitude, frequency, and phase parameters for each analysis frame. The short-time Fourier transform (STFT) is used to analyse the input signal into frames. The frequency window size and hop period are input to the STFT. An instantaneous frequency routine (**IFGram**) is also computed and uses a separate window for discrete differentiation to computing the instantaneous frequency as the time-derivative of the phase. The analysis stage is illustrated in the first part of Figure 1. Figure 2 shows the instantaneous frequency plot for the first 500msecs of a stopped violin sound, A4=440Hz (A4, MUMS Vol. 1, Track1, Index 16).

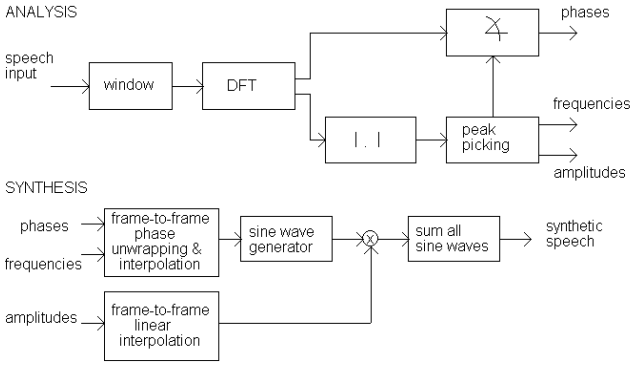

Figure 1. *McAulay/Quatieri Model.*

To account for rapid movements in spectral peaks due to such factors as voiced/unvoiced transitions and pitch changes present in speech signals or harmonic and inharmonic peaks and frequency modulation in music signals a peak-tracking the McAulay/Quatieri model uses a simple and effective peak-tracking. This tracks time-varying development marking the 'birth and 'death' along each constituent sinusoidal component. The algorithm matches peaks between STFT frames and interpolates the frequency position and magnitude along each track.. The output is sorted by start time (track 'birth') and then by frequency of each track's first value. In Figure 1. the absolute value of the SFTF is used in computing the amplitude and frequency for each track and the phase from the angle of the STFT. Figure 3 shows a total of 123 tracks for the first 500msecs of the violin note in Figure 2. Figure 4 shows the overlay of tracks on a speech signal STFT.

A magnitude threshold determines the number of tracks recovered and thus accounts for the timbral quality of the synthesized signal. In Figure 3 the threshold is set at 0.002 times the maximum magnitude present in the STFT. Frequency in Hz is calculated using the instantaneous frequency and bin position for each track.

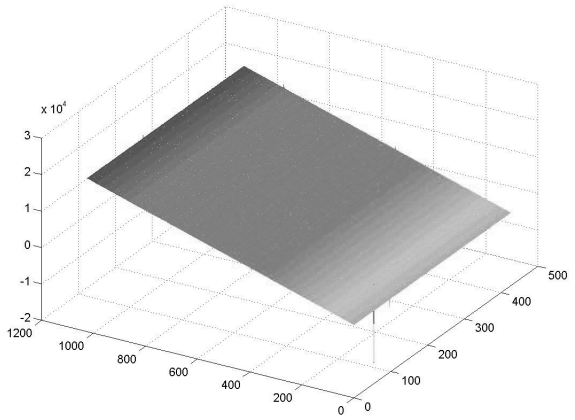

Figure 2. *IFGram plot for 500 msecs violin A4 sound.*

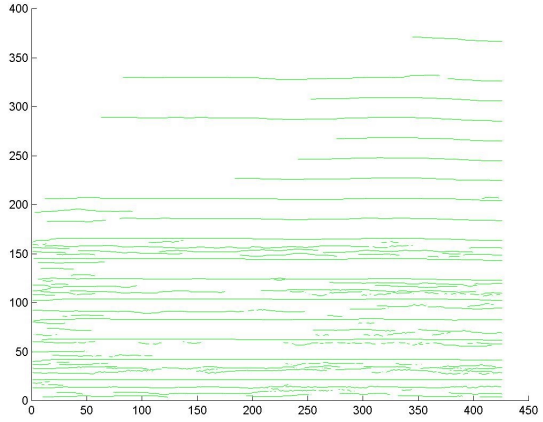

Figure 3. *123 tracks for the first 500msecs of the violin sound in Figure 1.*

1.2. Synthesis

In the synthesis stage a simple interpolation is used to compute amplitude values for each track between hop periods. To ensure that each frequency track is maximally smooth across frame boundaries, phase unwrapping is performed along each track of the STFT. For this, a cubic interpolation function is applied to smoothly interpolate the phase between STFT frames ensuring that the interpolated phase and frequency values equal those at the frame boundaries. The computed amplitudes, frequencies and phases for each track provide the parameters for the overlap-add resynthesis using a sine wave oscillator. The inputs to the synthesis routine are the amplitude, frequency and phase parameter sets for each track. A sine wave is computed for each track based on the interpolated frequency and phase values and a magnitude envelope is applied using the interpolated amplitude. This is illustrated in the second part of Figure 1. This analysis/synthesis method results in speech and music signals that are perceptually indistinguishable from the original signals and has been used for time-scale, pitch-scale, and frequency modification of speech.

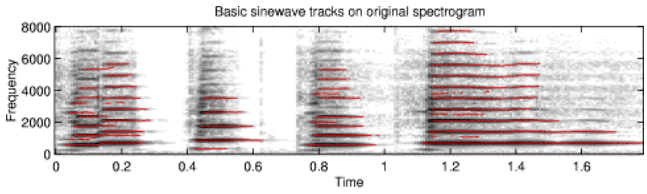

Figure 4. *Overlay of tracks on a speech signal STFT.*

2. IMPLEMENTATION

An object-oriented implementation of the technique is proposed, integrating the SndObj library [4]. It is based on two main classes: SinAnal and SinSynth. These classes are derived from a FFT base class, which provides the mechanisms for short-time Fourier analysis and Instantaneous Frequency Distribution. The FFT class itself is derived from SndObj, which is the base class for all processing classes in the library. The processing in this class is based on a `DoAnalysis()` method, which implements the STFT and a `DoIFAnalysis()` method which implements the IFGram. The former is an overridable method which can be modified to suit any analysis methods provided by derived classes. The analysis output is available to any FFT-derived object or to an FFT object itself (which can perform the inverse operation, by invoking the `DoProcess()` method). The outline of the main attributes and methods for this class is shown on fig.5.

```
class FFT : public SndObj
attributes
int      m_N          analysis size
int    m_hopsize      hopsize in samples
table* m_window     analysis window
float*  m_frame     STFFT frame data
float*  m_ifgram     IFGram data
methods
Output_FFT()    frame data access
Output_IFGram()  IFGram data access

DoAnalysis()    overridable main analysis method
DoIFGram()      IFGram analysis method
```

Figure 5. *The FFT class*

Since the Sinusoidal Analysis data is likely to be stored for further processing, a way of storing it on a proprietary file format is needed. A third class, providing these file IO services complements this initial set of tools. This will enable complex processing to be done in separate steps, if necessary.

2.1. SinAnal class

This class encapsulates the processes involved in extracting the harmonic peak tracks, with their associated amplitude, frequency and phase data. Figure 6 shows its most important attributes and

methods. It takes an input consisting of a number of STFT frames describing the analysed signal plus the IFGram. This input is generated by an FFT object. For each hop period, upon the invocation of its `DoAnalysis()` method, the SinAnal class generates an output consisting of a specified number of tracks, with amplitude, frequency in Hz and phase offset values on each track. This output is available through the `Output_Tracks()` method. The implementation outline of this method is shown on Fig. 7.

The number of tracks is variable and depends on the nature of the analysed signal and the threshold value, which is set at construction time. Likewise, the size of the output data on each hop period is dependent on the number of tracks present. The current number of tracks is updated every hop period. The current number of tracks is retrieved with `GetNumberTracks()` method.

```
class SinAnal : public FFT
attributes
int m_tracknos   number of tracks
int m_threshold  analysis threshold
float* m_tracks   track data

methods
GetNumberTracks() retrieves number of tracks
Output_Tracks()  track data access

DoAnalysis()   specialised analysis method
                      for track extraction
```

Figure 6. *The SinAnal class*

The SinAnal class tracks output format is understood by two basic classes: SinSynth, designed to resynthesize the analysed signal; and SndSinIO, which implements file input/output to a proprietary file format. Other classes which take this analysis data format are envisaged to be implemented. With these, techniques such as timestretching, filtering, pitch-tracking and cross-synthesis would be available to library users.

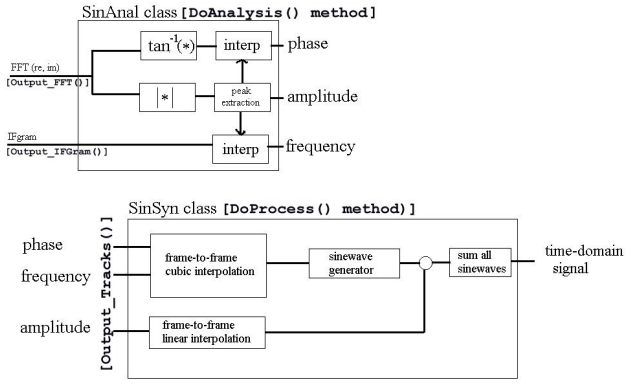

Figure 7. *SinAnal and SinSynth processing methods*

2.2. SinSynth class

This class, derived from FFT, takes an input from a SinAnal or SndSinIO object (plus any other object which can output track data) and implements a sinusoidal resynthesis of the analysis data. It implements a specialised version of the `DoProcess()` method inherited from the base class SndObj. This method implements the resynthesis from the magnitude, frequency and phase data present in the harmonic tracks, as shown in fig.?. Its output consists of time-domain signal with the resynthesized waveform. As with any SndObj-derived class, its output is available for further processing by other SndObj classes or to be output by a SndIO class. Figure 8 shows an example of a simple analysis-synthesis chain using the three classes discussed above. A soundfile input is put through FFT and sinusoidal analysis objects and is resynthesised using a sinusoidal resynthesis object. The resulting audio is written to a soundfile. It is expected that the usefulness of this processing will be complemented by spectral transformation classes, which will act on the sinusoidal analysis data.

```
SndWave input("infile.wav", READ); // input file
SndObj audio; // SndObj object to handle audio
FFT fft(&audio); // FFT
SinAnal sinus(&fft); // sinusoidal analysis
SinSyn  synth(&sinus); // resynmthesis
SndWave  output("outfile.wav", OVERWRITE); // output file

for(int i; i < end; i++){
  audio << input;
  fft.DoAnalysis();
  fft.DoIFGram();
  sinus.DoAnalysis();
  synth.DoProcess();
  synth >> output;
}
```

Figure 8. *Sinusoidal analysis example*

2.3. SndSinIO class

This class, derived from the SndObj library class SndIO, implements file IO to a WAVE FORMAT EXTENSIBLE-derived format[5]. Originally designed with multi-channel applications in mind, this is a new format specification from Microsoft which provides extensibility and customization of features, based on the original RIFF-Wave format. Within this format, any number of fields can be defined, as well as the data encoding and format itself. An identifier called the Globally Unique Identifier (GUID) is used to define new formats, so that that the original format can be freely extended by third parties without the need for previous registration. This specification has been successfully used for Phase Vocoder analysis files in the PVOC_EX format [6].

The file format is a chunked format, as shown in fig. 9. The format chunk is extended to incorporate proprietary information. It is defined by the C-language structure shown on fig. 10. In this format fields, the information stored is mostly related to the original waveform data (sampling rate, sample format, channels etc.). As with PVOC_EX, the full scope of the multichannel capabilities of the format can be used. The format proposed for the sinusoidal analysis data will include information such as hopsize, analysis threshold, maximum number of tracks etc. The

C++ structures defining the format are shown on fig. 11. The total size of the format chunk is 64 bytes (the WAVE_EXTENSIBLE requirement is that it should be aligned to 8-byte boundaries).

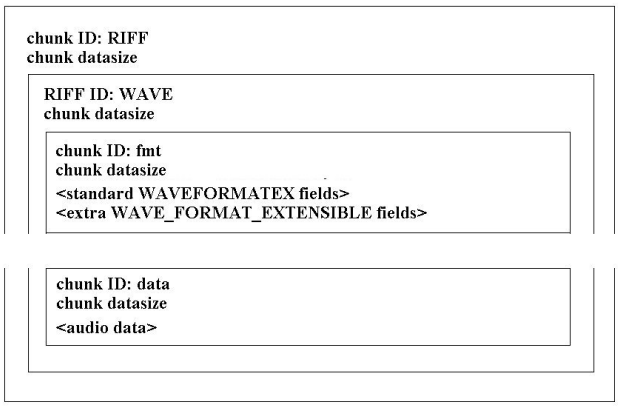

Figure 9. *WaveFormatExtensible outline*

The data chunk is composed of the track data, in the format output by the SinAnal class: for each hop period (frame) the data is composed of the number of tracks (integer) followed by the magnitude, frequency and phase for each track (single or double precision floating- point). Multiple channels are interleaved on a frame-by-frame basis (in a manner similar to the PVOC_EX format). The advantage of using a customised version of a well-known format is that it provides the possibility of sharing analysis data with other applications. Any number manipulating program that understand this format can be used to read, process or generate sinusoidal analysis data.

```
typedef struct {
    WAVEFORMATEX  Format; /* these are the first 18 bytes of
                     usual format data (channels, sr, etc)
        as applied to original waveform data */
    union {
        WORD wValidBitsPerSample; /* bits of precision */
        WORD wSamplesPerBlock;   /* valid if wBitsPerSample==0 */
        WORD wReserved;         /* If neither applies, set to zero. */
    } Samples;
    DWORD    dwChannelMask;
    GUID            SubFormat; /* the unique identifier */
} WAVEFORMATEXTENSIBLE;
```

Figure 10. *WAVE_FORMAT_EXTENSIBLE structure*

As mentioned above, the SndSinIO can provide input data for resysnthesis or further transformation well as storing the data generated by the SinAnal class. An object of this class can be constructed for input or for output. In the former case, as with all SndIO-derived classes, the `Read()` method is used iteratively to read data from the file. This data can be accessed by the `Output_Tracks()` method (in a similar way to SinAnal). Other information about that data can be retrieved by a number of methods (`GetTrackNumbers()`, `GetThreshold()` etc.). In the case of output, the data is written using the `Write()` method. A description of the main aspects of this class is shown in fig.12.

3. CONCLUSION AND FURTHER RESEARCH

The Sinusoidal analysis technique can be very useful for sound and music processing applications. The addition of this facility to the SndObj library provides a very powerful way of manipulating sound spectra. So far, only the analysis and the resynthesis have been proposed, but it is expected that many more classes will be developed to deal with the data generated by this type of processing.

```
struct WAVEFORMATSINUSEX {
    WAVEFORMATEXTENSIBLE wavexfmt;
    DWORD dwVersion;              // version control
    SINUSFORMAT sinusfmt;         // the format structure
}

struct SINUSFORMAT {
  WORD  wDataFormat;       // IEEE_FLOAT or IEEE_DOUBLE
  WORD  wHopsize;          // hopsize in samples
  WORD  dwWindowType;      // window type
  WORD  dwMaxTracks;       // max number of tracks
  DWORD  dwWindowSize;     // FFT window size
  float fThreshold;        // analysis threshold
  float fAnalysisRate;     // FFT analysis rate
} SINUSFORMAT
```

Figure 11. *WAVEFORMATSINUSEX structure*

```
class SndSinIO : public SndIO
attributes
WAVEFORMATSINUSEX m_format       data format
int m_mode      input or output
char* m filename  spectral data file

methods
GetThreshold(),
      GetFFTSize()  etc    analysis properties
GetTrackNumbers()    the number of tracks
                       relative to  the current frame
Write()      writes data to file
Read()       reads data from file
```

Figure 12. *SndSinIO class*

4. BIBLIOGRAPHY

[1] McCaulay, Robert, J., Quatieri, Thomas F., "Speech *Analysis/Synthesis Based on a Sinusoidal Representation*", IEEE Trans. On Acoustics, Speech, and Signal Processing, Vol. ASSP-34, No. 4, August 1986.

[2] Serra, X. "Musical Sound Modelling with Sinusoids plus Noise". G.D. Poli an others (eds.), *Musical Signal Processing*, Swets & Zeitlinger Publishers, 1997.

[3] Tellman, E., Haken L., and Holloway B., "Timbre *Morphing of Sounds with Unequal Numbers of Features*", J. Audio Eng. Soc., 43:9 1995.

[4] Lazzarini, V,"The Sound Object Library", Organised Sound 5 (1), Cambridge: Cambridge Univ. Press., 2000, pp. 35-49

[5] http://www.microsoft.com/hwdev/audio/multichaud.htm

[6] Dobson, R, "PVOC-EX: File format for Phase Vocoder data, based on WAVE_FORMAT_EXTENSIBLE". http://www.bath.ac.uk/~masrwd/pvocex/pvocex.html