A STREAMING AUDIO MOSAICING VOCODER IMPLEMENTATION

Edward Costello, Victor Lazzarini, Joseph Timoney

The Sound and Digital Music Technology Group
National University of Ireland
Maynooth, Ireland
Edward.Costello@nuim.ie
Victor.Lazzarini@nuim.ie
Joseph.Timoney@nuim.ie

ABSTRACT

This paper introduces a new extension to the concept of Audio Mosaicing, a process by which a set of unrelated sounds are blended together to form a new audio stream of shared sonic characteristics. The proposed approach is based on the algorithm that underlies the well-known channel vocoder, that is, it splits the input signals into frequency bands, which are then processed individually, and then recombined to form the output. In a similar manner, our mosaicing scheme first uses filterbanks to decompose the set of input audio segments. Then, it introduces the use of Dynamic Time Warping to perform the matching process across the filterbank outputs. Following this, the re-synthesis stage includes a bank of Phase Vocoders, one for each frequency band to facilitate targeted spectral and temporal musical effects prior to recombination. Using multiple filterbanks means that this algorithm lends itself well to parallelisation and it is also shown how computational efficiencies are achieved that permit a real-time implementation.

1. INTRODUCTION

From the works of James Tenney and John Oswald to popular music's embrace of sampling and remixing, the re-appropriation of existing acoustic material into musical works is still a popular composition tool for electronic musicians [1]. More recently, the concept of Audio Mosaicing was introduced. In essence this means combining segments from different audio samples in such a way as to create a new hybrid sonic texture. The rules governing the assembly are particular to the individual mosaicing algorithm. However, to ensure perceptual continuity some form of matching between the segments needs to be made. Originally this was user-driven, that is, the composer painstakingly worked through many possibilities to find the best combination. This could be intensive and very time consuming. Therefore, automation of this audio mosaicing process has become an active area of research in recent years and has incorporated a wide variety of audio analysis and re-synthesis techniques.

Figure 1 gives a general overview of how an automated audio mosaicing system works that is audio input driven. These systems consist of an audio pre-processing component and a performance component. At the pre-processing stage a database of audio samples is analysed and one or more audio feature vectors are derived from each of the samples. These feature vectors and the corresponding samples are stored in the system's memory, in some cases using a database component. At the performance stage, the system takes a streaming audio input signal and splits it into smaller segments. Each segment is then analysed using one or more analysis feature vectors. The feature vectors derived from the streaming input are then compared to the corresponding feature vectors for each sample in the database. When a suitable match from the database has been found according to the matching algorithm, this audio is output from the system.

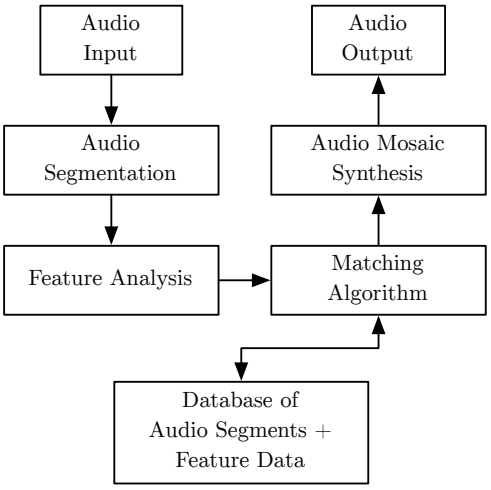


Figure 1: *Diagram showing the operation of an audio driven sound mosaicing system.*

Most audio mosaicing systems either use some form of spectral decomposition or a series of high level audio descriptors to describe the audio samples used for synthesis. The system described in [2] performs real-time concatenation synthesis using a database of audio units and pre-analysed descriptors. Segments are selected for output based on their calculated geometric proximity to a target position within a descriptor space. The system proposed by [3] incorporates models to predict the high-level descriptors of the transformed audio segments rather than using the many instances of the transformation itself. This produces computational savings. Other methods such as [4] use an 11-band spectral loudness measure to match a pre-analysed corpus and re-synthesise the data using a phase vocoder.

Similar to some previous work, the system proposed in this paper also uses spectral analysis to describe the audio. The audio input and pre-analysed audio are compared using the dynamic time warping algorithm. Matching is also done on a per band basis

rather than across the whole spectrum because this ensures that the output segments will have a more fine-grained spectral similarity. This work also augments the number of output re-synthesis possibilities by enabling the combination of segments from different spectral bands at different time points in an audio file. The system proposed here shares many properties with the channel vocoder [5] and can be thought of as a modification to this synthesis approach. It replaces the carrier with a pre-analysed audio file and phase vocoder re-synthesis, and the envelope follower with dynamic time warping segment comparison and selection. Using a phase vocoder for re-synthesis, this system in essence performs a limited descriptor driven transformation as per [3] that is in the form of a linear time stretch applied to the selected output segments so that they match the length of the analysed input segments. Although this is, compared to some corpus based systems, a computationally expensive method of audio mosaicing, some implementation optimisations can be introduced that will help make this approach feasible for real-time performance. It is the aim of this audio mosaicing implementation to produce an audio output which closely resembles the audio input in sonic character, yet has recognisable tonal characteristics from the pre-analysed audio used for the synthesis.

Section two of this paper will explain how the system presented here is designed and implemented. Starting initially with how it resembles the channel vocoder, the pre-analysis stage is described showing how the audio used for synthesis is processed into filtered spectral magnitude vectors used during the segment comparison process. This process uses dynamic time warping which is shown to achieve accurate segment matches. The re-synthesis process is covered next, and it is detailed how a bank of phase vocoders can be used here for additional processing flexibility. At the end of section two implementation optimisations are discussed. Since much of the processing is done using filterbanks, the algorithm should be amenable to parallelisation. If considered carefully, it should be sufficiently fast to achieve real-time performance, even when comparing large numbers of segments. In section three some test results are presented comparing spectral features from the systems input to the processed audio output. Finally in the concluding section a summary evaluation of the work is given followed by a number of possible improvements to the system that are currently being investigated.

2. SYSTEM OVERVIEW

A high level overview of the audio mosaicing synthesis system presented here is shown in Figure 2. This system creates audio mosaics by combining segments of audio which are split in the time and frequency domain. The system consists of a pre-processing and performance stage. At the pre-processing stage, an audio file is read into the system and is split into smaller audio segments in the time domain. Each of these segments is analysed using a high level audio description algorithm, producing a feature vector. This feature vector is further split into a number of sub-vectors which correspond to a particular frequency band within the audio segment. During the performance process, audio is read into the system from a streaming input, such as a microphone, and is also split into smaller segments in the time domain. Each streaming input segment is analysed with the same high level audio description algorithm as the pre-analysed audio segments. The current input feature vector is also then split into sub-vectors that are representative of the audio at different frequency bands within the input signal. Each of the sub-vectors from the input signal are then compared to sub-vectors in every pre-analysed segment at the corresponding frequency band. When this comparison has been completed across every pre-analysis segment, the best matching sub-vector from each frequency band is selected for output. The corresponding audio of each selected sub-vector is combined at every frequency band into a new audio segment created from the pre-analysis audio. Finally, this audio segment is output from the system.

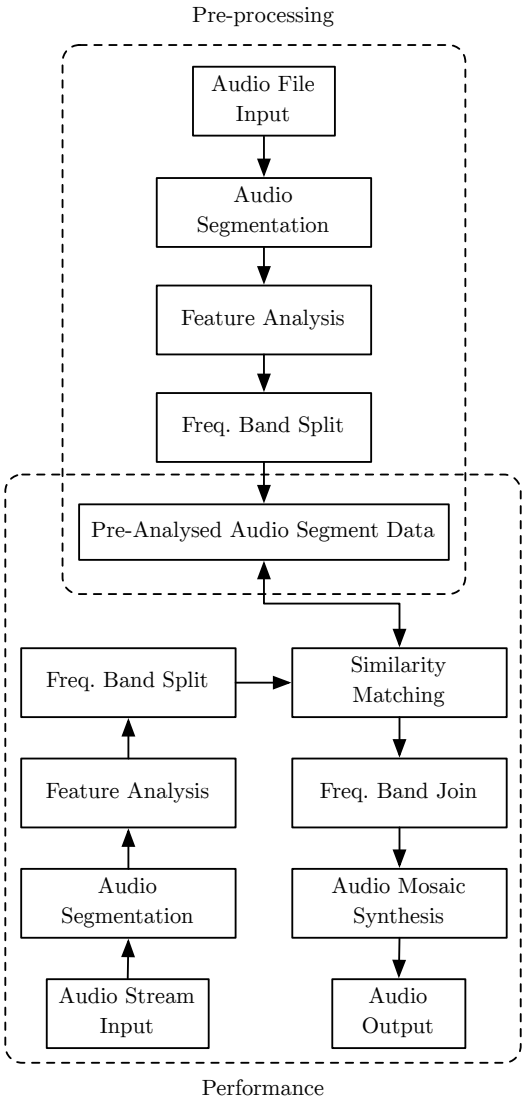


Figure 2: *Diagram showing an overview of the audio mosaicing system described here.*

2.1. A Modified Channel Vocoder

This system bears many similarities to the standard channel vocoder synthesiser. This section will detail the design of the channel vocoder followed by an explanation of how the audio mosaicing system presented here is similar in implementation.

The channel vocoder takes two input signals to perform synthesis, these are known as the carrier and modulator respectively. The carrier is normally a spectrally rich sound source and provides the frequency content of the synthesised sound. The modulator is optimally a signal with high sound energy fluctuation and it controls the amplitude of each frequency band in the carrier signal. As shown in Figure 3 both inputs are split into frequency bands using two identically configured filter banks. The filter banks use band pass filters with their centre frequencies and bandwidths logarithmically distributed across the audio frequency spectrum. The energy levels of each band produced by the modulators filter bank are tracked by an envelope follower. The output of the envelope follower is used to control each of the gains in the corresponding bands of the carrier's filter bank.

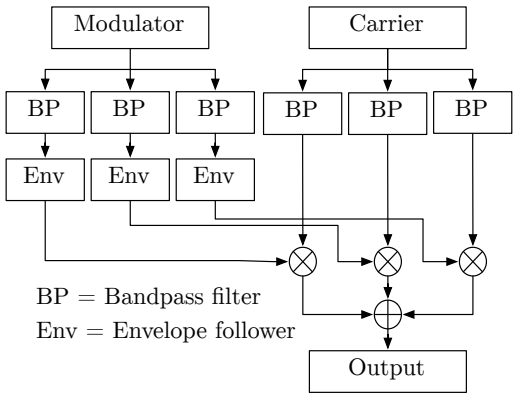


Figure 3: *The channel vocoder using a filter bank comprised of three bandpass filters and three envelope followers*

Figure 4 shows how the design of this system differs from the channel vocoder. Instead of using a carrier and modulator signal this audio mosaicing system uses an audio file and any arbitrary streaming sound source as input. The audio from the file and streaming input is segmented in the time domain prior to any processing. Similar to the channel vocoder, when audio has been input to the system and analysed using a high level feature vector, in this case using a filter bank, this feature vector is split into sub-vectors which represent different frequency bands. Whereas the channel vocoder uses the amplitude variations in the modulator to control the carrier channel in each frequency band, this system uses feature vectors derived from the audio input using a filter bank and dynamic time warping to select output segments from an audio file. When the best matching segments from each frequency band are selected they are synthesised using a phase vocoder and output from the system. This system can use mono or stereo audio for the audio input and pre-analysis files. If the audio is in a stereo format it is mixed to mono for the feature analysis and segment comparison components of this system.

2.2. Pre-processing

The audio used for synthesis is input to the system from a sound file which is converted into 32 bit normalised floating point sample values at a sampling rate of 44,100 Hz. The audio samples are converted to spectral magnitudes using a windowed short-time Fourier transform (STFT). The STFT is performed using a frame size of 1024 samples and a hop size of 256 samples. Each frame

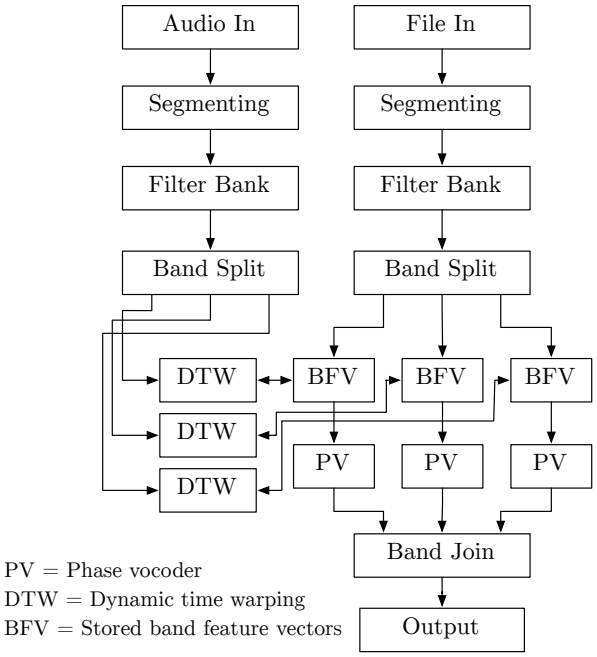


Figure 4: *Diagram showing the structure of an audio mosaicing vocoder using three frequency bands*

is windowed using a von Hann windowing function. The spectral magnitude is then derived from each of these frames. These magnitudes are filtered using a bank of logarithmically spaced triangle filters similar to the triangle filter bank used in Mel-frequency Ceptral Coefficient processing. In this case the filters' frequency ranges span from 0 Hz up to the Nyquist frequency and the centre frequencies are calculated using the following formula:

$$f_n = \frac{10^{\frac{n}{N+1}} - 1}{9} \cdot \frac{Fs}{2},\ n = \{1, 2, 3, ..., N\} \qquad (1)$$

Where f is the centre frequency n is the filter number, N is the amount of filters used and Fs is the sample rate.

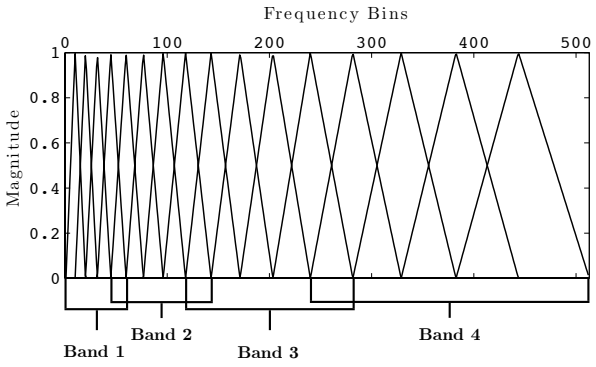


Figure 5: *Plot showing the filter weights of 16 triangle filters over 512 frequency bins and split into 4 frequency bands*

As shown in Figure 5, the frequency ranges of the triangle fil-

ters overlap. The 'lower and upper' cut-off frequencies of each filter are defined as the centre frequencies of the previous and following filters except for the first and last triangle filters where the cut-off is at 0 Hz and the Nyquist frequency respectively. Each STFT frame is filtered by taking the dot product of the spectral magnitude frame and the spectral weights of the triangle filter bank. The filtered magnitude vectors are then divided into sub-vectors which correspond to different frequency bands in the audio spectrum. This can also be seen in Figure 5 where the filter weights are split into four bands. The sub-vector split is done by dividing the filtered magnitude vector into evenly spaced sub-vectors. The sub-vector data for each frequency band of the pre-analysis audio is stored in memory for on-line comparison with sub-vectors from the corresponding input audio frequency band.

Pre-analysed audio is segmented in time either by using a constant based on the number of STFT frames contained within a segment, or using a beat detection algorithm. The beat detection algorithm is a slightly modified version of that proposed by [6]. This beat detection algorithm makes an estimate of the audio file's tempo. It then finds the STFT frame numbers that denote the beginning of a measure. These frame numbers are used to segment the audio file by measure. This implementation of the beat detection algorithm differs slightly from that proposed in [6] as instead of calculating the 'onset strength' vector by using the first order difference of a log-magnitude 40 channel Mel-frequency spectrogram, it uses the spectral flux function [7]. The spectral flux function was used due to both its simplicity of implementation and accuracy when used for onset detection.

2.3. On-line processing

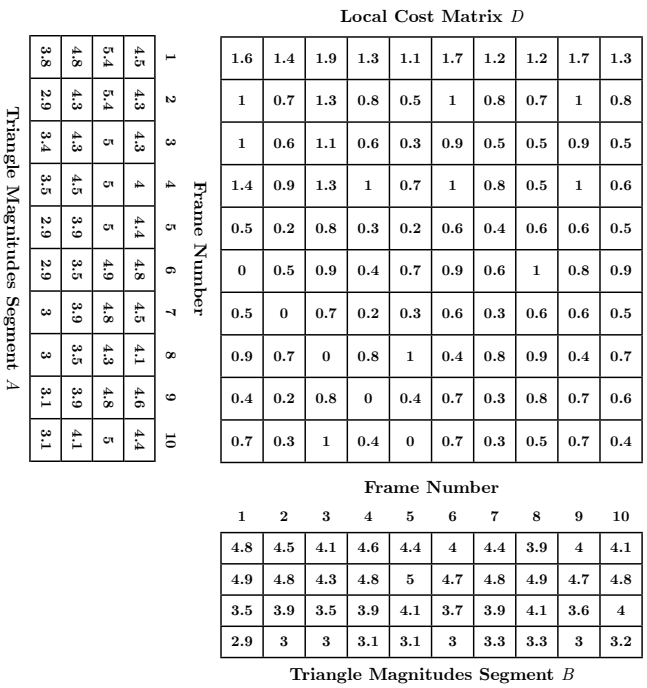
During performance audio samples are read into a buffer before analysis. The size of this buffer is specified by the amount of STFT analysis frames it can store, and dictates the size of each input segment that will be compared to the pre-analysed audio. A buffer size of four frames will yield a minimum latency of 23 milliseconds at a hop size of 256 samples and sample rate of 44,100 Hz.

Input audio is processed when the input sample buffer has been filled with the number of samples required. As with the pre-analysed audio file, each frame of sampled audio is processed using the STFT and filtered in the frequency domain using the triangle filter bank. These magnitudes are split into the previously specified frequency bands for comparison with the pre-analysed audio data. This comparison is performed sequentially over every pre-analysed audio segment.

2.4. Feature vector comparison

The feature vectors derived from input audio segments are compared for similarity with those stored in memory using the dynamic time warping algorithm. This comparison calculation is performed on each sub-band separately and compares the current input triangle filtered magnitudes to every segment stored from the pre-analysis stage of processing.

The dynamic time warping algorithm is used to find a similarity value of two discrete signals, in this case the triangle filtered magnitude segments of pre-analysed audio and input audio. In order to find the similarity value, a local cost matrix D of size $p \times q$ is created where p is the amount of frames in an input triangle filtered magnitude segment and q is the amount of frames in a pre-analysed triangle filtered magnitude segment. This matrix contains



Triangle Magnitudes Segment A	Frame Number
3.8	1
2.9	2
3.4	3
3.5	4
2.9	5
2.9	6
3	7
3	8
3.1	9
3.1	10

Local Cost Matrix D									
4.5	4.3	4.3	4	4.4	4.8	4.5	4.1	4.6	4.4
4.8	4.3	4.3	4.5	3.9	3.5	3.9	3.5	3.9	4.1
5.4	5.4	5	5	5	4.9	4.8	4.3	4.8	5
4.5	4.3	4.3	4	4.4	4.8	4.5	4.1	4.6	4.4
1.6	1.4	1.9	1.3	1.1	1.7	1.2	1.2	1.7	1.3
1	0.7	1.3	0.8	0.5	1	0.8	0.7	1	0.8
1	0.6	1.1	0.6	0.3	0.9	0.5	0.5	0.9	0.5
1.4	0.9	1.3	1	0.7	1	0.8	0.5	1	0.6
0.5	0.2	0.8	0.3	0.2	0.6	0.4	0.6	0.6	0.5
0	0.5	0.9	0.4	0.7	0.9	0.6	1	0.8	0.9
0.5	0	0.7	0.2	0.3	0.6	0.3	0.6	0.6	0.5
0.9	0.7	0	0.8	1	0.4	0.8	0.9	0.4	0.7
0.4	0.2	0.8	0	0.4	0.7	0.3	0.8	0.7	0.6
0.7	0.3	1	0.4	0	0.7	0.3	0.5	0.7	0.4

Frame Number									
1	2	3	4	5	6	7	8	9	10
4.8	4.5	4.1	4.6	4.4	4	4.4	3.9	4	4.1
4.9	4.8	4.3	4.8	5	4.7	4.8	4.9	4.7	4.8
3.5	3.9	3.5	3.9	4.1	3.7	3.9	4.1	3.6	4
2.9	3	3	3.1	3.1	3	3.3	3.3	3	3.2

Triangle Magnitudes Segment B

Figure 6: *Tables showing the calculated local cost matrix as part of a dynamic time warping comparison of two triangle filtered magnitude band segments A and B*

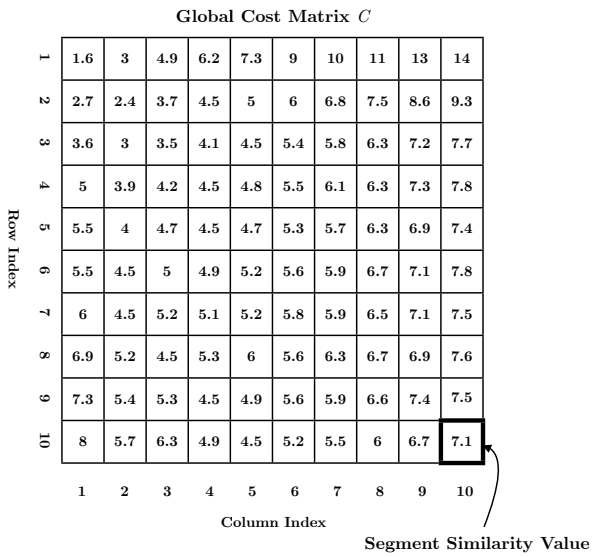
the euclidean distance of every frame in each of the segments. The euclidean distance of two vectors, a and b, is given by:

$$D(i,j) = \sqrt{\sum_{n=1}^{N}(a_n - b_n)^2}. \tag{2}$$

Where i is the row index of D, j is the column index of D, N is the number of elements in each vector and n is the vector's element index. Figure 6 shows an example of a calculated local cost matrix for the filtered magnitude segments A and B. These segments are 10 frames in length with each frame containing a triangle filtered magnitude sub-vector with four filtered values, equivalent to one of the frequency bands shown in Figure 5. A global cost matrix C of size $p \times q$ is then calculated by moving sequentially through every element in the local cost matrix in row major order, and adding the minimum value from cells within a local search grid to the current matrix cell:

$$C\left(i,j\right) = D\left(i,j\right) + \min \begin{cases} C\left(i,j-1\right), \\ C\left(i-1,j\right), \\ C\left(i-1,j-1\right) \end{cases} \tag{3}$$

Figure 7 shows the global cost matrix C calculated from the local cost matrix D. The similarity value of segments A and B is given as the value at $C(p,q)$. Typically the dynamic time warping algorithm also computes an optimal warp path to align the two series, but this step has not been added to this implementation. The dynamic time warping comparison process is complete when it has compared the current input segment to every segment in the



	1	2	3	4	5	6	7	8	9	10
1	1.6	3	4.9	6.2	7.3	9	10	11	13	14
2	2.7	2.4	3.7	4.5	5	6	6.8	7.5	8.6	9.3
3	3.6	3	3.5	4.1	4.5	5.4	5.8	6.3	7.2	7.7
4	5	3.9	4.2	4.5	4.8	5.5	6.1	6.3	7.3	7.8
5	5.5	4	4.7	4.5	4.7	5.3	5.7	6.3	6.9	7.4
6	5.5	4.5	5	4.9	5.2	5.6	5.9	6.7	7.1	7.8
7	6	4.5	5.2	5.1	5.2	5.8	5.9	6.5	7.1	7.5
8	6.9	5.2	4.5	5.3	6	5.6	6.3	6.7	6.9	7.6
9	7.3	5.4	5.3	4.5	4.9	5.6	5.9	6.6	7.4	7.5
10	8	5.7	6.3	4.9	4.5	5.2	5.5	6	6.7	7.1

Figure 7: *Figure showing the calculated global cost matrix C*

pre-analysed audio at every frequency band. Once this is done it will have identified which frequency bands of which segments in the pre-analysed audio will need to be synthesised.

2.5. Synthesis

The audio synthesis for this project is performed using the Csound audio programming language and the underlying Csound API. The segments for each frequency band are synthesised using separate instances of a Csound instrument shown in Figure 8 that uses the streaming phase vocoder opcodes. Using the phase vocoder opcodes greatly simplifies doing any necessary time-dilation of the pre-analysed audio. This is needed when the segments of pre-analysed audio are selected using the beat detection method as they will differ in length with the input audio segments and possibly other segments in the pre-analysed audio file. The audio segments are also split into frequency bands in the frequency domain by scaling the magnitude part of the phase vocoder data.

When the pre-analysis of the audio file input has completed, Csound's *pvanal* program is invoked to convert the input audio files samples to a phase vocoder (PVOCEX) data file that can be read by the phase vocoder opcodes. The *pvsfread* opcode reads the phase vocoder data file and converts it into streaming phase vocoder data. This opcode takes a control rate time pointer variable which is used to read each section of the audio file. Each phase vocoder's time position is read from a function table allocated using the Csound API before performance. When the segment comparisons for each band of the current input segment have been completed, the program writes the time positions of the selected segments for every frame to the table. The phase vocoder time position tables are read using a single always-on instrument that outputs a global phasor value at control rate, oscillating one cycle for every input buffer segment read. Although linear time dilation can be achieved by simply changing the oscillation rate of the phasor opcode and using the phasor's output value directly with the *pvsfread* opcode, the strategy used here makes the possible future implementation of non-linear time stretching a more

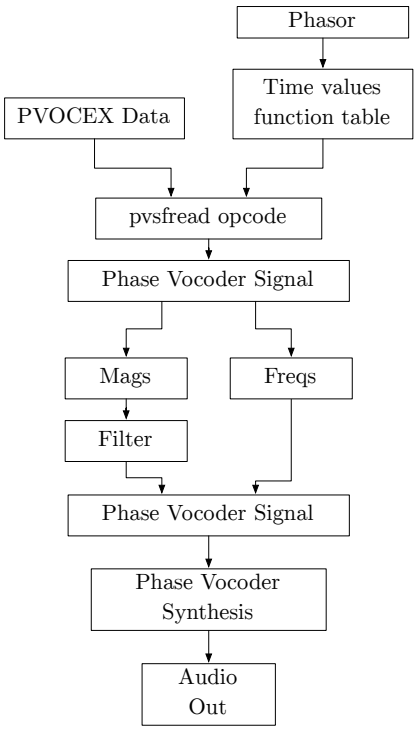


Figure 8: *Diagram showing the design of the phase vocoder instrument used in Csound*

straightforward process.

The band pass filtering of the phase vocoder data is performed in the frequency domain using the *pvsftw* and *pvsftr* opcodes. During performance the magnitudes of every current frame of phase vocoder data are written to a function table. This magnitude vector is then multiplied by a vector containing the sum of the triangle filters magnitudes for the corresponding frequency band.

The Csound code for the instrument which synthesises the system's output is discussed in the following paragraphs. Each part of the Csound code is preceded by a discussion of the code's function within the instrument. At the start of the instrument code, variables are declared to denote the number of the table that stores the time values for reading through phase vocoder buffer and a gain table to filter the phase vocoder magnitude data to a specific frequency band. The table to store the magnitudes from the phase vocoder signal is also generated using the *ftgen* opcode.

```
i_TimeTableNumber = p5 + $WarpPathBase
i_GainTableNumber = p5 + $BandGainBase
gi_PVSMagnitudes ftgen 0, 0, gi_FFTSize /
    2, -2, 0
```

The time pointer which reads through the phase vocoder stream is read from the table which stores the time values. This time pointer is used with the *pvsfread* opcode to read from the phase vocoder data stored in the file "PVSInput.pvoc". This is done for two channels as the output is in stereo.

```
gk_TimePointer table gk_PhasorIndex ,
    i_TimeTableNumber, 1
```

```
f_ChannelOne pvsfread gk_TimePointer,
    "PVSInput.pvoc", 0
f_ChannelTwo pvsfread gk_TimePointer,
    "PVSInput.pvoc", 1
```

The magnitude part of the phase vocoder signal is written to the magnitudes table using the *pvsftw* opcode. The values stored in the table are multiplied by the values in the table which stores the gain values for filtering the signal to a particular frequency band. This multiplication is done using the vectorial opcode *vmultv*. The magnitudes are read from the table using the *pvsftr* opcode and copied back into the phase vocoder signal.

```
k_flag pvsftw f_ChannelOne,
    gi_PVSMagnitudes
vmultv gi_PVSMagnitudes,
    i_GainTableNumber, gi_FFTSize / 2
pvsftr f_ChannelOne, gi_PVSMagnitudes
k_flag pvsftw f_ChannelTwo,
    gi_PVSMagnitudes
vmultv gi_PVSMagnitudes,
    i_GainTableNumber, gi_FFTSize / 2
pvsftr f_ChannelTwo, gi_PVSMagnitudes
```

Finally the phase vocoder signal is converted to audio samples using the *pvsynth* opcode and output from the instrument.

```
a_ChannelOne pvsynth f_ChannelOne
a_ChannelTwo pvsynth f_ChannelTwo
outs a_ChannelOne, a_ChannelTwo
```

2.6. System Optimisation

As the data processing required to synthesise audio with this system is quite processor intensive, some optimisations were introduced to this implementation in order to achieve an efficient operation. These optimisations were achieved by using Single instruction, multiple data (SIMD) programming libraries and the *OpenCL* programming language.

Wherever possible the DSP components of this system were programmed using Apple Inc.'s *Accelerate* framework. This framework contains a number of libraries such as *vDSP* and *cblas* that use SIMD in order to speed up common DSP tasks such as FFT, convolution, and vector / matrix arithmetic.

It was also necessary to optimise the dynamic time warping comparison calculations in order for the system to process streams of audio in real-time due to the number of comparisons required using large pre-analysis data sets. A pre-analysed audio file of three minutes length and a sample rate of 44,100 Hz will contain approximately 31,000 STFT frames using a frame size of 1024 samples and a hop size of 256. If this audio is segmented using a frame size of eight and a frequency band count of four this will require approximately 15,500 feature vector comparisons for each segment input to the system from the audio in stream.

The feature vector comparison process lends itself well to parallelisation and this was exploited in order to achieve good performance over large input audio files and numbers of frequency bands. The feature vector comparison is implemented using the OpenCL programming language which allows for a large number of lightweight threads known as *kernels* to execute concurrently. This greatly optimises the vector comparison operation on multi-threaded processors.

Every segment comparison for each band is calculated using separate OpenCL kernels, which are executed simultaneously on every processor core, or queued and executed on subsequent free processor cores as they become idle. When the comparison has completed, the segment with the lowest total cost value in each frequency band is selected for re-synthesis.

3. RESULTS

The aim of creating this audio mosaicing system is to produce an audio output that closely resembles the sonic character of the audio input but yet is synthesised using audio segments taken from another sound source. In order to quantify this, some test examples have been created with the audio mosaicing system that use several audio input and analysis files.

These tests use a number of different system settings which are shown as the column headers in Table 1. The first column however indicates the test number. The 'Mode' setting determines how the pre-analysed audio is segmented, whether it is segmented based on a constant frame size are using beat detection. In 'framed' mode the 'Seg. Length' setting specifies the number of frames in each segment taken from audio input stream and the size of each segment in the pre-analysis audio. In 'beat detection' mode this setting only effects the audio input stream. The 'Tri. Filters' setting indicates the number of triangle filters present in the filter bank. The 'Bands' setting designates how many frequency bands each audio segment is split into.

The spectrograms showing the audio input and resulting audio output were made using a MATLAB script entitled *logfsgram.m* available from [8] which creates a log-frequency spectrogram. The spectrograms were created using an FFT frame size of 1024 samples and a hop size of 256 samples using a Hamming window. The resulting amplitudes were also normalised to 0 dB and any amplitudes below -80 dB were reduced to $-\infty$ dB. This was done in order make the colours on each spectrogram representing any given amplitude values identical, further clarifying the similarities and differences between each spectrogram. It should be noted also that due to the way this system is implemented there will always be a slight latency introduced between the input and output audio. This can be seen on each of the spectrogram pairs presented in these results.

Test	Mode	Seg. Length	Tri. Filters	Bands
1	Framed	2	30	10
2	Beat Detect	20	30	10

Table 1: *Audio mosaicing test settings*

3.1. Test 1

The first test was performed in framed mode with a short input segment size and using a high number of triangle filters and frequency bands as shown in Table 1. Figure 9 shows spectrograms of the first second of audio from the input and mosaicing output. These spectrograms share a number of similarities but there are also some notable differences. For instance, at the beginning of the mosaicing output spectrogram there is an area of silence spanning

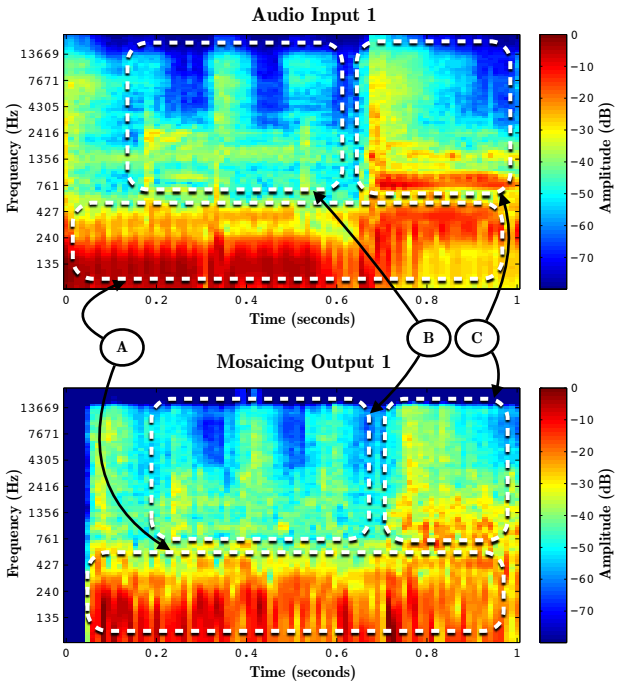


Figure 9: *Spectrograms from Test 1 comparing audio input and synthesised output from the system*

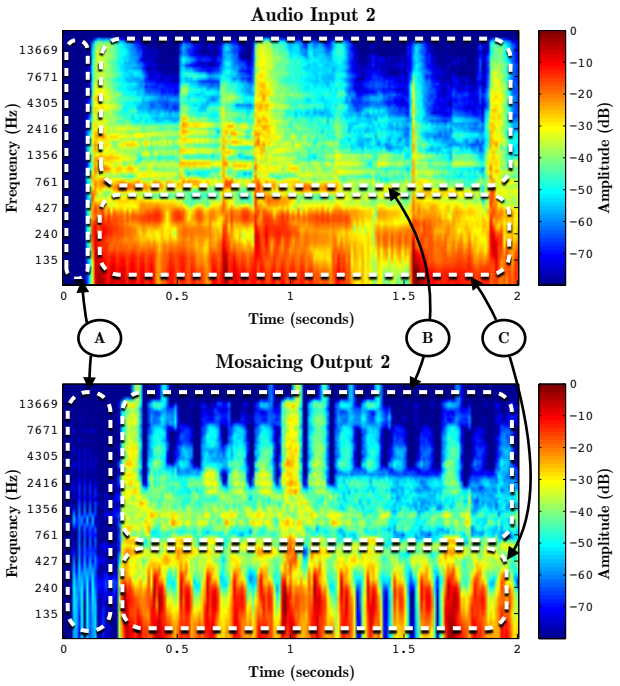


Figure 10: *Spectrograms from Test 2 comparing audio input and synthesised output from the system*

the entire frequency spectrum for the first ~0.04 seconds. This silence is introduced in the output due to the inherent latency within the audio-mosaicing system caused by input buffering and the synthesis component.

In the areas marked by label A in both spectrograms, a strip of high amplitude audio can be seen spanning the whole of the audio input, and starting at ~0.04 seconds and spanning the rest of the length of the audio output. This area has a frequency range of from 0 Hz to ~427 Hz in the audio input and from 0 Hz to ~700 Hz in the audio output. The area spanning from ~0.72 seconds to the end of the input spectrum with a frequency range of from 0 Hz to ~240 Hz and an amplitude ~-30 dB however does not seem to have been reproduced in the audio output.

In the areas marked by label B, three columns of higher amplitude audio can be seen in both spectrograms. These columns start at ~0.16,~0.32 and ~0.46 seconds in the audio input, and ~0.22,~0.34 and ~0.55 seconds in the audio output. The amplitude range in these sections spans from ~-65 dB to ~-25 dB in both the audio input and audio output.

In the areas marked by label C, the spectrogram of the audio input does not seem to have been synthesised as accurately in the audio output. A defined strip of high amplitude content ranging from ~0.7 to 1.0 seconds from ~740 Hz to ~900 Hz shown in the input spectrogram, cannot be clearly seen in the output spectrogram.

3.2. Test 2

This test was performed using the same triangle filter and frequency band settings as the first test, but uses beat detection mode and a larger input segment size. As can be seen from the spectrograms in Figure 10 a larger number of artefacts are introduced to the output signal than when using lower segment sizes in framed mode.

As can be seen at label A, the low amplitude section at the beginning of the audio input is reproduced in the audio output. In the output spectrogram however, a number of narrow columns of audio at ~-55 dB are present.

In the sections labelled by B, there are some notable features of the input spectrogram that can be seen in the output spectrogram. For example, similar columns of high amplitude audio between ~-30 dB and ~-20 dB can be seen in both spectrograms. These begin at ~0.1 and ~0.85 seconds in the input spectrogram and at ~0.25 and ~0.9 seconds in the output spectrogram. Large areas of low amplitude audio can also be seen in the audio input spanning from ~1.25 to ~1.6 seconds and ~1.65 to ~1.8 seconds ranging from ~1000 Hz to 22,050 Hz. Similar areas can also be found in the output from ~1.4 to ~1.65 seconds and ~1.7 to 2 seconds. Due to this test being performed in beat detection mode, there is a noticeable recurring spectral pattern present every ~0.1 seconds in the output spectrogram that is not shown in the input spectrum. This is due to the longer segment length of the pre-analysed audio.

In the areas shown by label C, there is an area of high amplitude in the input spectrogram spanning from the ~0.1 seconds to the 2.00 seconds mark, with a notable area of weaker amplitude spanning from ~1.3 seconds to ~1.6 seconds with a frequency range of from 0 Hz to ~240 Hz. In the corresponding area of the output spectrum, areas of high amplitude can also be seen in this section with the notable difference that there is a sequence of lower

amplitude areas occurring at ~0.1 second intervals beginning at the ~0.5 second mark. These lower amplitude sections become weaker in amplitude in the area corresponding to the lower amplitude sections in the input spectrogram. Similar to the area labelled B, the recurring pattern introduced into the output which is not present in the input spectrum is an artefact of using the beat detection mode.

3.3. Discussion

As can be seen in the first test, using small segment lengths with a large number of triangle filters and frequency bands can produce an output spectrum that closely resembles the audio input, albeit slightly offset in time due to latency introduced to the system by buffering and synthesis. There are a number of differences between the spectrograms also but this should be expected as the pre-analysed audio may not contain the necessary audio content for a truly accurate reconstruction.

Beat detection mode was used during the second test, and as can be seen in the output spectrogram, this introduces a repeating pattern in the resulting audio spectrum caused by the large pre-analysis segment sizes created when using this mode. Although the spectral differences produced using this mode are clear between the input and output, a number of the more prominent features from the input are reproduced in the output.

As can be seen from the results this audio mosaicing system can produce an output that retains many of the sonic characteristics of the input used for processing yet introduces a number of unique spectral features to the produced audio. A more accurate reproduction of the input can be achieved when applying a high resolution of analysis to the audio by using a small segment size and high number of frequency bands. Lower resolutions will also produce output with similar audio characteristics to the input but will have introduced a number of differences within the audio spectrum. The audio for these tests is included with the source code for this system which is available at https://github.com/eddyc/Streaming-Audio-Mosaicing-Vocoder.

4. CONCLUSION

This paper has described the implementation of an audio mosaicing vocoder system. The system uses dynamic time warping of filtered magnitudes to identify similar audio segments over a number of separate frequency bands. By combining frequency bands from different sections of the pre-analysed audio file, the system can produce audio that more closely matches the input. Programming the dynamic time warping system in OpenCL has enabled the system to perform a larger amount of segment comparisons over a constrained time frame by taking advantage of multi-threaded processing.

5. FUTURE WORK

This system is under active development and the implementation of a number of improvements to the synthesis system is currently being investigated.

A possible method to improve performance of the segment comparison component could be accomplished by implementing dynamic time warping by calculating each row of the local and global cost matrix on a per-frame basis rather than all at once when a full segment has been buffered by the input. This method would spread the calculation over more than one audio callback perhaps enabling drop-out free performance with an increased amount of segment comparisons.

The resemblance of synthesised output to the input may be improved by performing the warp path calculation when the dynamic time warping comparison has been completed and the best matching segments from each band are identified. The warp path can then be used to alter the speed at which the segment is read by the phase vocoder bank. Another possibility of improvement could be to perform dynamic frequency warping producing a warp path to alter the magnitude spectrum of each phase vocoder to more closely match the spectrum of the target input audio segment.

6. ACKNOWLEDGEMENTS

This work is supported by the Program of Research in Third Level Institutions (PRTLI5) of the Higher Education Authority (HEA) of Ireland, through the Digital Arts and Humanities programme.

7. REFERENCES

[1] Kevin Holm-Hudson, "Quotation and context: Sampling and john oswald's plunderphonics," *Leonardo Music Journal*, pp. 17–25, 1997.

[2] Diemo Schwarz, Grégory Beller, Bruno Verbrugghe, Sam Britton, et al., "Real-time corpus-based concatenative synthesis with catart," in *Proc. of the 9th Int. Conference on Digital Audio Effects (DAFx-06)*, 2006.

[3] Graham Coleman, Esteban Maestre, and Jordi Bonada, "Augmenting sound mosaicing with descriptor-driven transformation," in *Proc. of the 13th Int. Conference on Digital Audio Effects (DAFx-10)*, 2010.

[4] Miller Puckette, "Low-dimensional parameter mapping using spectral envelopes," in *Proceedings, International Computer Music Conference, Miami*, 2004.

[5] Sadaoki Furui, *Digital Speech Processing: Synthesis, and Recognition*, vol. 7, CRC PressI Llc, 2001.

[6] Daniel PW Ellis and Graham E Poliner, "Identifying 'cover songs' with chroma features and dynamic programming beat tracking," in *Acoustics, Speech and Signal Processing, 2007. ICASSP 2007. IEEE International Conference on*. IEEE, 2007, vol. 4, pp. IV–1429.

[7] Simon Dixon, "Onset detection revisited," in *Proc. of the 9th Int. Conference on Digital Audio Effects (DAFx-06)*, 2006.

[8] Dan Ellis, "Spectrograms: Constant-q (log-frequency) and conventional (linear)," `http://www.ee.columbia.edu/ln/rosa/matlab/sgram/`, accessed <20/05/2013>.

[9] Aymeric Zils and François Pachet, "Musical mosaicing," in *Proceedings of the COST G-6 Conference on Digital Audio Effects (DAFX-01)*, 2001.

[10] Victor Lazzarini, Joseph Timoney, and Thomas Lysaght, "Streaming frequency-domain dafx in csound 5," in *Proc. of the 9th Int. Conference on Digital Audio Effects (DAFx-06)*, 2006.

[11] Simon Dixon, "Live tracking of musical performances using on-line time warping," in *Proc. of the 8th Int. Conference on Digital Audio Effects (DAFx-05)*, 2005.