Lab Report 2

A. D. Hanna 19 May 2013

In *Written Review 1* a brief historical outline and survey with regard to granular techniques as they relate to digital signal processing was undertaken. Granulation is just a specific subset of time-dependent algorithms that transform input signals according to the principle that "what something *is* depends on *when* it is more than anything else". [1] What characterises granular synthesis is the short - possibly 0.5 ms - sections of contiguous samples taken from the input signal and relayed to the output signal. Subsequent to the initial research some basic granulation techniques have been implemented within the Matlab environment and this writing seeks to furnish the reader with an understanding of the reasons for the decisions made with regard to the development of the algorithm as well as a step-by-step account of how the procedure achieves its purpose. Following this is a discussion, the focus of which is a number of specific soundfiles that the program has produced. The conclusion of this writing deals with any shortcomings and lack of robustness percieved in the implementation as it currently stands and with the scope of improvements that might both address these deficiencies and extend the utility and capability of the program in the future.

The approach taken regarding the implementation of the algorithm was not that of identifying and then solving a specific technical problem although many technical problems had to be addressed in order to produce a satisfactory outcome. That there are better ways to identify frequencies in a signal for the purposes of pitch-shifting and that the 'froginess' produced by down-pitching using conventional PSOLA techniques can be significantly reduced by refining pitch-marking algorithms is not in question. [2] But while such trajectories are undoubtedly worthy of consideration a need to think about the motives that the devised algorithm embodies is equally important. If a convincing simulation of reality is the 'prime directive' then the computer is only too able to provide the means to this end; legions of worthless Hollywood CGI-powered films are testament to this fact. In contrast the artifacts produced by the failure of an algorithm when pushed to the limit can on occasion provide aurally interesting material. Rather than labour to produce a tool only to leave concerns regarding its possible uses to others, this writer was motivated to direct the programming toward producing an output that would be aesthetically interesting whilst remaining detached from imperatives relating to verisimilitude. In other words I wanted to devise a tool that would make me want to use it.

GrainLn

The starting point for the coding process was the 'grainLn' function presented on page 230 of chapter 7 of Udo Zölzer's DAFX: Digital Audio Effects. [3] The first argument to the function takes an input signal 'x' this being a single column vector of amplitude values x_a where $-1 \leq a \leq +1$. An initial sample S_n where $1 \leq n \leq$ (N = the total number of samples in the input file) is chosen via the second argument 'init', and the length of the grain in samples is obtained via the third argument 'L'.

One problem with the original function is the case where the length of L exceeds N. This is dealt with simply via the conditional statement: if L > N,then L = N as represented by the Matlab code at line 11 in grainLn.m: if L > length(x), L = length(x); end. The second problem with the original function is the case where the length of the input file N ≤ init+L. As the contiguous index range of values for the length of each grain is established as being from 'init' to 'L-1' the condition where init ≥ N - init+L will terminate the function - and the program that uses it - printing: 'length(x) too short.'. With regard to 'bullet-proofing' the function so that the program as a whole is not 'murdered in its infancy', the conditional statement: if init+L+1 > N, then init = N - L+1 as represented by the Matlab code at line 13 in grainLn.m: if init+L+1 > length(x), init = length(x) - L+1; end will perform the necessary adjustment.

The final argument to grainLn is 'Lw', this serving, in the original code, as a value for the number of samples, double the length of which plus 1, equals the length of the window function applied to the amplitude values of each grain. Here the problem with the original code is the use of the transposition operator ' before the closing semicolon. Window functions in Matlab produce column vectors by default and removal of the ' enables the original code to function as intended.

The next problem with the original code is that it seeks, via an over-complicated routine, to turn the bell-curve produced by the 'hanning' function into a tukey window. A simple solution is to use Matlab's 'tukeywin' function. The value of L can be passed directly to the first argument of tukeywin, this being the length of the grain in samples and Lw is passed to the second argument where Lw/2 is a percentage value of L used as the ascending part of a cosine for fading the signal in and as the descending part of a cosine for fading the signal out. Values close to zero for Lw produce a window that approaches a rectangle Figure 1, and values close to 1 for Lw produce the bell-curve Figure 3. While Figure 1. provides more of the signal, the almost-square shape will induce high fre

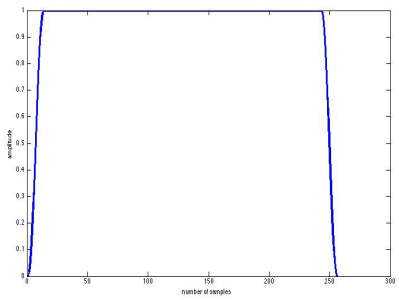

Figure 1.

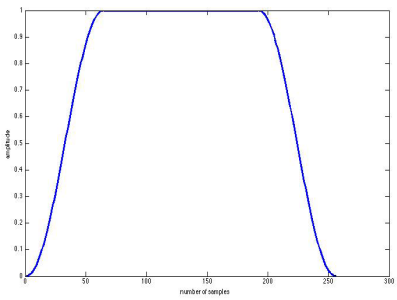

Figure2.

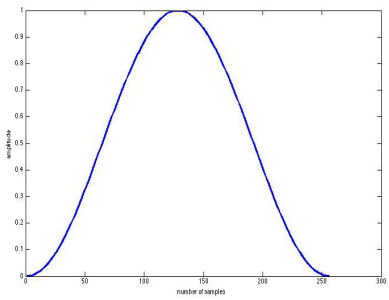

Figure 3.

quency energy as a result of convolution in the time domain. Similar effects may also be produced by the window depicted in Figure 3 albeit with less of the original signal present. Figure 2. represents a Tukey window with Lw = 0.5 and is obviously the best compromise. All y axis values in the three plots are between 0 and 1, and all x axis values are, in relation to the curve, the discrete sample numbers ranging, in this case, from 1 to 256. The raw amplitude values are multiplied piecewise with those of the window function and this is achieved in the code at line 24: y = y.* w;.

To summarise, by itself, the grainLn function does the basic work of extracting one short section of contiguous samples from an input signal and the arguments of the function determine the sample number in the input file that is the initial sample for the grain, how many samples long the grain is and what percentage of it is faded in and out by the window function used by grainLn.

Granulatormono

At the next level, the 'granulatormono' function determines the choice of mode with respect to how the grains are to be selected in the input file and how those grains are to be disposed in the output file; this function contains the program's synthesis engine.

Once again the function granulatormono is based on Udo Zölzer's code, this time 'granulation.m' on page 230 of chapter 7 of DAFX: Digital Audio Effects. In the original code, the number of samples in the input file is placed in a variable 'Ly' and Ly zeros are created, these providing a 'placemark' for the output sound file. The first step was to decorrelate the two functions and make L_{input} independent of L_{output}. The original code has a number of 'constants' these being set values for 'nEv' = the number of grains (or events), 'maxL' and 'minL' = the maximum and minimum length respectively of the grain, and 'Lw' the percent of (maxL - minL) samples used by the window function for fading the grain in and out. These variables are now arguments to the function.

The first argument to granulatormono is 'insig'. This can be a stereo file but for reasons discussed below the script 'granulator' will eliminate one of the two channels transforming insig into a single column vector. The length of insig is stored in 'numSamplesIN' and the 4th argument 'Lout' receives user-input, this being the length in seconds desired for the output file i.e. Lout*(fs = the sample rate = samples per second). This value is stored in 'numSamplesOUT', and a vector of numSamplesOUT 'place-holding' zeros is created and stored in 'outsound'.

The 3rd argument 'numEvents' determines the number of grain lengths taken from the input file pseudo-random between argument 6 and 7 i.e. 'maxL' and 'minL' respectively. The values produced are offset by adding minL, rounded, and stored in 'grainLength'.

There are currently 3 methods for extracting grains from the input file and these are managed by first asking for user-input and then, via a conditional control structure, determining the choice accordingly. Method '1' selects 'numEvents' grains, the onset points of which are pseudo-random between (numSamplesIN - maxL). The values are rounded up and stored in initIN. If the input signal contains speech, then this method will render the output signal unintelligable for grains of short duration. The intelligibility is proportional to the ratio of the length of the grains to the number of events multiplied by the length of the output file or $F_i = (L_{grains} / Ev_n) \ L_{output}$. 'Frankenstein26.wav' 'Frankenstein32.wav'

Method '2' also selects 'numEvents' grains, the onset points of which are pseudo-random between (numSamplesIN - maxL) but the values are sorted in ascending order of magnitude. In this mode the program functions as a serviceable time-compressor-expander. 'Frankenstein28.wav' 'Frankenstein34.wav'

In addition to this capability, for an output file of around 2 seconds in length, if the number of grains approaches 7000 and each grain is around 20000 samples long then an effect occurs that this writer was most unprepared to hear i.e. reverberation. To this writer's ears the quality of the reverb effect easily surpasses the Schroeder Reverb and takes vastly less time to calculate for roughly the same 'room size'. 'Frankenstein7.wav' 'Frankenstein9.wav' 'Frankenstein17.wav'

Method '3', by using Matlab's 'linspace' function accesses the input file according to numEvents equidistant between 1 and (numSamplesIN - maxL). For numEvents > 20 per second an audible frequency will result and if this is coupled with the same method for the output file, the frequency (F = numEvents / Lout) will increase in amplitude and where numEvents > 4000 per second, produce what sounds like feedback. 'Frankenstein13.wav'

There are currently 4 methods for disposing the grains selected from the input file with respect to the output array. Method '1' uses Matlab's 'logspace' function to distribute numEvents onset sample numbers between $10^{0.1}$ and $\log_{10}$ (numSamplesOUT - maxL). When the results are rounded down via the 'floor' function, $10^{0.1} = 1$. For demonstration purposes, we simplify the input:

```
numSamplesOUT = 1000;
maxL = 200;
numEvents = 30;
initOUT = floor(logspace(0.1,log10(numSamplesOUT - maxL),numEvents));
initOUT =  1    1    1    2    3    3    4    5    7    9   11   14   18   22   28   35   44   55  69   86
107   134   168   210   262   328   410   512   640   800
```

The function attempts to apply negative acceleration in the time domain but is less than ideal when one considers how the values of initOUT will effect the output in relation to an input file containing speech. The first 20% the output is 'verticalised' by way of the grains having the same onset sample number during 5/6 of this initial period. 'Frankenstein22.wav'. Leaving aside issues of intelligibility, the effect can be quite interesting 'Frankenstein30.wav'.

Method '2' attempts to implement positive acceleration in the time domain via a slightly different route. Using the same method as method '1' the maximum value of initOUT = 'biggest' is determined and a set of 'numEvents' differences are calculated by subtracting the values of initOUT from biggest. These values are then stored in 'difference'. A set of 'numEvents' ratios are then calculated by dividing the values in difference by biggest. A new array of values 'newTime' contains 'numEvents' values and is calculated by multiplying 'biggest' by the values in 'ratio' with an offset of 1. The values are sorted and stored in 'initOUT'.

initOUT = 1 161 289 391 473 539 591 633 667 694 715 732 746 757 766 773 779 783 787 790 792 794 796 797 798 798 799 800 800 800

In this case the differences between the onset sample points gets progressively smaller and the 'verticalisation' occurs at the end of the array. 'Frankenstein31.wav'

Method '3' and Method '4' at the output stage are the same as method 3 and method 1 respectively, at the input stage.

Next, a set of 'numEvents' amplitudes, pseudo random between 0 and 1 are created and stored in 'amplitudes'. The sample numbers for the points at which the individual grains are terminated are acquired by adding 'grainLength'-1 to the onset sample values in 'initOUT'. These values are stored in 'endOUT'.

The final stage of granulatormono, is the looped routine that is the function's synthesis engine. The first line of code addresses the input stage: for an index from 1 to 'numEvents', the 'grainLn' function is called with the input file as the first argument, the indexed values of 'initIN' for argument 2, the indexed values of 'grainLength' in argument 3, and the fade value used by the window function at argument 4. The result is stored in 'grain'.

The second line produces the output: the indexed value for initOUT (the start of the grain) and endOUT (the end of the grain) are located within the place-holder array of zeros 'outsound' and 'grain' multiplied by 'amplitudes' is added to those zeros contiguous from initOUT to endOUT.

The Synthesis formula is given by

$$y(n) = \sum_{k} a_k g_k (n - n_k)$$

where ak is an eventual amplitude coefficient and nk is the time instant where the grain is placed in the output signal. [4]

Granulator

All the procedures so far described are, with the exception of both the ± acceleration output methods, roughly 'steady state' with regard to the ratio of the grain density to the length of the output file. Rather than output one file produced with a single set of parameters, the 'granulator' script loops the entire algorithm according to the number of 'segments' input by the user and these segments are concatenated before the final output stage. By this method a sound file can be rendered that exhibits significant discontinuity in relation to all the parameters that receive user-defined input. 'Frankenstein1.wav'

The first stage of the 'granulator' script asks the user for the name of the input sound file. This file needs to be a .WAV file and needs to be in the same directory as the 'grainLn.m', 'granulatormono.m' and 'granulator.m' files. Once the file is read in via Matlab's 'wavread' function, information regarding the sample rate and the number of channels in the file is ascertained. If the input file is a stereo file

the user is asked which channel is to be used and the other channel is discarded. The reason for this is to preserve simplicity at the input stage so that the often complex results of the output are more easily analysed. 'Frankenstein36.wav'

Next the number of segments desired for the final output is evaluated from user-input. For each segment the following information is required: the number of events for each channel (currently 2) these being the grains taken from the input file first for channel 1 and then channel 2 of the output file, the maximum and minimum length of each grain for each channel, and a single fade-in/out value used for both channels. The 'granulatormono' function is called twice, first for channel 1 and then for channel 2. Each time the function is called, user-input is requested in relation to the 3 modes of selection from the input file and the 4 modes of disposition intended for the output file. The output from both instances of granulatormono is then normalised and then stored in cell array 'bothchannels'. When the above process has completed ('numseg' = number of segments repetitions), the loop is terminated, the arrays are extracted from 'bothchannels', converted back from cell arrays to matrices and concatenated. The result is played via Matlab's 'soundsc' function and a final decision regarding whether or not to write out to file is requested.

Conclusion

The variety of output that the program produces was unexpected and this writer is of the opinion that such time-dependent processes are capable of producing *many* more different types of effects.

From a listening perspective, the sine wave represents an auditory nightmare, an eternally uniform dreariness that, nevertheless, traditionally forms the basis of digital signal processing. A dramatic improvement is attained, however, if one granulates a sine wave signal using such techniques as those implemented in the program. 'Frankenstein35.wav' What makes a sound listenable is surely a degree of 'roughness', a kind of irregularity and imperfection that often distinguishes the sounds made by musical instruments from those made by digital means. Granularity can provide this quality 'in spades'.

With regard to further development, a whole series of concatenation techniques that use the procedure defined at the last stage of the 'granulator' script as a starting point will, when deployed at the beginning of the program, deliver an initial sound file to the input that is the product of many sound files.

Certainly, refinements with regard to the ± acceleration methods for disposing the grains in the output file are necessary. An obvious remedy is to test 'initOUT' for identical values, remove them, and inform the user of the amended value associated with 'numEvents'.

A graphic user interface is probably desirable but this writer is happy not to deal with yet another program where endless amounts of windows have to be dragged around, at least for the time being.

References

1. Frank Zappa.

2. Filling the gaps between the grains of down pitching PSOLA or "Getting the frog out of PSOLA", Adrian von dem Knesebeck, Udo Zölzer, Helmut Schmidt, Audio Engineering Society Convention Paper 8504, Presented at the 131st Convention 2011 October 20–23, New York,USA.

3. DAFX: Digital Audio Effects, Udo Zölzer, Copyright © 2002 John Wiley & Sons, Ltd ISBNs: 0-471-49078-4 (Hardback); 0-470-84604-6 (Electronic) Chapter 7, p. 230.

4. Ibid.