How to Shape Existing Sounds with LHC Data

by Archer Endrich — LHCsound
http://www.lhcsound.com

General Description

The sonification of data collected from various sources is becoming more fascinating by the day for engineers and composers. 'Sonification' means turning data into audible sound. All manner of data is being collected and analysed in many areas of study.[1] Lying within this data may be meaning, pattern, revelations about the world around us. In an effort to understand this data, which at first is an incomprehensible string of numbers, some people have taken to enlisting the help of sound: the language of pattern, the other face of mathematics, the medium of our supersensititive receptors, the ears.

This particular document discusses efforts to sonify data that comes from particle collisions in the Large Hadron Collider at CERN, or simulations of these collisions. It deals with one approach to the task: using the data to shape existing sounds. The pre-existing sound brings its own characteristics into the equation, while the data itself brings its own numerical qualities, whatever they are, and turns the sound into something more than it was before, something that opens a new sonic universe while providing easily perceptible information about the data itself. This approach, therefore, leads to many new possibilities both for musical composition and for scientific analysis.

We are not here setting out to write an academic paper, but rather to take the interested reader and potential researcher on a journey into the procedure of data sonification. It introduces a set of mathematical and software tools with which the reader can begin to carry out their own explorations. In this, it aims to be both practical and thorough, moving from general background to increasingly technical matters. In essence, it is a 'How-to' manual.

- Section I - **The Challenge:** sets the scene at the Large Hadron Collider.
- Section II - **Data Conversion:** a long and detailed discussion about mathematical procedures that can be used to convert raw data to sound. It focuses on the principle of proportion, dynamic relationships central to Nature and to Art. This material may be of use to mathematics and teachers of physics, especially when connected with the sounds that emerge from the data. Data conversion methods underly the creation of software to carry out these various tasks.
- Section III - **Musical Applications:** an overview of some of the musical objectives that form the end result, i.e., a melody, a harmony, a rhythm etc.

[1]One of the most fascinating examples I have heard is a series of soundtracks made from data collected by the Voyager Space Probe.

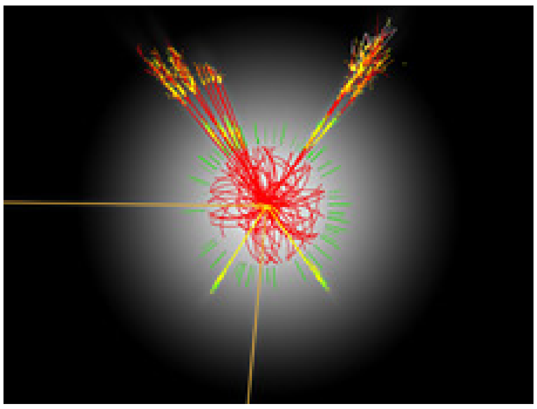
- Section IV **-Script Library:** a descriptive list of the contents of a ‘library’ of scripts written in Tcl/Tk to carry out data conversion and create files needed by specific sound-making programs. This script library is directed at the task of shaping existing sounds with the data. There is a Reference Manual for the Script Library.
- Section V - **Run sound-making programs with the files produced:** now one actually shapes sounds with the input data and produces audible output. This section lists each program and which musical task it handles. These programs are selected from the sound transformation software of the Composers’ Desktop Project (CDP), but the Script Library can be adapted to run other software that will accept command line input.
- Section VI - ***SoundingData*****, a Tcl/Tk GUI Application:** This section provides a general introduction to this graphic program. *SoundingData* forms the main software tool that puts the above together into a practical, working environment. It also has its own Reference Manual.

Please note that this approach of shaping existing sounds with data is complemented by another approach, namely deriving sound directly and only from the data itself. This side of things has been developed by Richard Dobson and has its own paper and graphic software application. Many of the procedures involved in the two approaches overlap.

I - THE CHALLENGE

Millions of Numbers

The Large Hadron Collider churns out numbers by the million. Two particles collide after whizzing around their 17 kilometer ring in opposite directions. Bang! and they break into smaller 'pieces' that fly out in all directions and speeds and energies and distances. It all happens in an instant.


Particle Collision simulating the Higgs Boson

What are they doing?

The scientists want to know what these ’pieces’, these ’particles’, are and how they behave. Which ones have more energy or speed or mass? Do the flying particles make any kind of pattern that helps us to answer these quetions? Finding the answers is not easy because it all happens so fast and there are so very, very many numbers. How can we find out which ones are important and tell us what we want to know?

Studying the Numbers

Much of the work at CERN is concerned with analysing the numbers, and every data-analysis and graphic tool imaginable is used or invented to help in the effort. There may be a completely different approach that can help. We know that our ears are super-sensitive and can pick up tiny, rapid differences – even better than our eyes can. Not only that, but in the audio realm, time can be slowed down. LHCcsound has been set up to explore what we might be able to learn about the data should we be able to listen to it.

Numbers into Sound

OK, but how do we do that? How do we turn LHC lists of numbers into something that we can hear? Fortunately, lists of numbers and music have something in common: they both go up and down, become higher and lower or larger and smaller. Melodies go up and down, loudness goes up and down, the time-lengths of notes are shorter or longer, there is the temporal augmentation and diminution of phrases, the number of notes in a chord could be few or many, the number of partials (frequency components) in a timbre (the ’colour’ of a sound) can be few or many, regularly or irregularly spaced. Therefore, to hear what a list of numbers sounds like, we just have to get the original numbers to fit into the value ranges that work for these various parameters of music. Then music software can use these numbers to make or manipulate sounds.

II - DATA CONVERSION

Introduction: Scaling & Proportion

This process of ’fitting’ numbers into specific value ranges is called ‘scaling’ in mathematics: we convert the data from one value range to another. So we call this process ‘data conversion’. Our goal is to turn LHC data numbers intp musically useful values.

For example, suppose we have the data number list: 2, 3, 5, 7, 8. In music, one range of numbers that is used for pitch (melody or harmony or loudness) is 1 to 127. This is the MIDI range: ‘Musical Instrument Digital Interface’. When used for pitch, these values are called ‘MIDI Pitch Values’, abbreviated to ‘MPV’. To hear our data values as pitches, therefore, we need to scale them into MPVs. To use them as amplitudes (‘velocity’ in MIDI-speak), we also scale them into the MIDI range. Let’s see how this is done.

One way of doing this is to keep the same proportions between the numbers as in the original data (i.e., ‘direct proportion’ or ‘linear conversion’). We can describe a procedure in two steps. Before we outline these two steps, let us try something simpler to see what happens. This may show us why we need those two steps.

But what is ‘proportion’? It is a question worth asking and a topic worth studying in some detail, because proportion is something that is extremely important in artistic creations of all types: it refers to the (measurable) relationship of parts to one another, and of parts to the whole. For the artist, it has an directly intuitive ‘felt’ dimension as well as a numerical dimension. This is because so many other factors are also at work in an artistic creation, such as colour / timbre, intensity, pitch range, and amplitude (energy levels) — all of which influence the perception of proportions. But there is often a firm numerical foundation as well.

A proportion may look just like a fraction, but it is something more, something dynamic. The point is that when one value changes — and the emphasis is on change — the other part also changes so that the relationship between them, the ‘ratio’, stays the same. When we convert data values to musical value ranges, we need the relationship between the values, the proportions, to stay the same. Otherwise, the connection between the data and the music is lost and the result is meaningless.[2] So let’s look at this more closely.

Proportion[3] is the relationship between quantities, expressed mathematically as y : x. This means: ’y varies directly with x’, or ’y is proportional to x’, or ’what part of x is y?’, or ’$\frac{y}{x}$’). In other words, when x changes (source), y needs to change by a certain amount (target) to keep the same numerical relationship — proportion — between them.

Thus, if x (source) is 5 and y (target) is 10, $\frac{y}{x} = \frac{10}{5} = 2$ — 2 is the ‘certain amount’, which we can call k: here, y is twice x. We found the scaling factor k by determining what part of x, y is: $k = \frac{y}{x}$. In a ’direct proportion’, all number pairs are in the same relation. In this case, therefore, all y’s will be twice x. The proportional relationship here is 1 to 2, i.e., whatever x is, y will be twice that value, and 2 is the ’scaling factor’ (k in our formula). $10 = 2 * 5$ or generically, $y = kx$.

Let’s try this with different values. What if x is 30 and the scaling factor k is 2, then y is $2 * 30 = 60$. If x is 12, y is $2 * 12 = 24$, and so on. Note that each of these relationships resolve to 2: $\frac{10}{5}$, $\frac{30}{15}$, $\frac{24}{12}$. We find the scaling factor by determining the ratio between two known values. Then this scaling factor is used to find the ’other’ value when one of them changes: $k = \frac{y}{x}$ becomes $y = kx$ — [the algebra: $k * x = (\frac{y}{x}) * x$, and the x’s on the right cancel out]. Summarising, when x is changing:

$y = k * x$
$y = 2 * 5 = 10;\ \frac{10}{5} = 2$
$y = 2 * 30 = 60;\ \frac{30}{15} = 2$
$y = 2 * 12 = 24;\ \frac{24}{12} = 2\}$

[2]The value relationships can also be effectively skewed for a number of good reasons that brings out the perception of the data more, not less, accurately.

[3]See for example *Higher Mathematics for AQA GCSE* by Tony Banks and David Alcorn. Causeway Press. 2006, pp. 164-172: ’Direct and Inverse Proportion’.

Thus we can see that each numerical relationship has the same proportion: all y's are twice x. Now let us work through an example of data conversion.

Worked Example of Data Conversion to Musical Values

Illustration of an inadequte solution

Clearly, we need to make the data numbers 2, 3, 5, 7, 8 higher so that they are in a more useful part of the MIDI range, neither too quiet to be heard nor too loud. Let us try multiplying the data values by the arbitrary value 10 to make them higher. This gives us 20, 30, 50, 70, 80 – which seems OK. But what if a data value was 1 or 0.5 or 0.05. Multiplying by 10 gives us 10 or 5 or .5, all of which are too quiet to be heard. And what if a data value was 19? Multiplying by 10 gives us 190, which is a long way above the top value in the MIDI range – and could damage our hearing (if the speaker didn't blow out). So something else needs to happen.

This brings us to the first of two steps.

STEP 1 to find out what part (i.e., proportion) of the whole range of data values each data value is.

The **range** is highest-value minus lowest-value, in this case $8 - 2 = 6$. If we divide each data value by 6, we find out what part of 6 each data value is:

value/range = what part of the range
$2/6 = 0.33$
$3/6 = 0.5$
$5/6 = 0.833$
$7/6 = 1.1616$
$8/6 = 1.333$

If the range is 6, we can see that this still isn't quite right, because we are not starting at the bottom of the range of 6 (i.e., 0) and ending at the top (i.e., 1). Therefore we subtract the minimum value from *all* the values just as we did to determine what the range was: 8 (max) - 2 (min) = 6 (range). Therefore we have:

(value-minval)/range = what part of 1
$(2 - 2)/6 = 0$
$(3 - 2)/6 = 0.166667$
$(5 - 2)/6 = 0.5$
$(7 - 2)/6 = 0.833333$
$(8 - 2)/6 = 1$

Normalisation / Reciprocals

We can do this in a simpler, more generic way by 'normalising' the data. This means finding out what part of 1 each data item is, which effectively transfers all the data to the same value range. We can do this by dividing 1 by the **range** of data values: in this case 1/6 = 0.1666... or 0.166667 if we round to six decimal places. This gives us the 'scale factor' by which we can move all our input data values into the 0 to 1 range (using the formula discussed above, $y = kx$). As before, as we want all the values to stay in this range (6), we subtract the minimum data value from all of the data values:

*(value-minval) * normalising scale factor = what part of 1*
$(2-2) = 0 * 0.166667 = 0.000000$
$(3-2) = 1 * 0.166667 = 0.166667$
$(5-2) = 3 * 0.166667 = 0.500001$
$(7-2) = 5 * 0.166667 = 0.833335$
$(8-2) = 6 * 0.166667 = 1.000002$

Notice that the numbers come out the same as before (but to more decimal places), but the calculation is simplified by using the scaling factor. Each data item has been transferred into the 0 to 1 value range, maintaining the same relationship. We can check this by comparing the 2nd and 3rd data items (as parts of the range): $\frac{1}{3} = 0.333333$ and $\frac{0.166667}{0.500001} = 0.333333$

This completes the first of the two steps. It has told us what part of 1 each data item is, 'rationalising' them into a single value range.

STEP 2 is to move these values into the MIDI 1 to 127 value range.

Let us select 30 to 90, i.e., neither too quiet nor too loud. 90 - 30 = 60, so the target range is 60. If we multiply each normalised input data by 60, we accomplish this task:

*normalised-range * target-range = value in 1 to 127 MIDI values*
$0.000000 * 60 = 0.0$
$0.166667 * 60 = 10.000002$
$0.500001 * 60 = 30.000006$
$0.833335 * 60 = 50.00001$
$1.000002 * 60 = 60.0012$

(The miniscule fractional parts are caused by rounding the scale factor to six decimal places. Without this rounding, some results end up fractionally below, which seems worse. Further rounding can make everything integers, should we wish to keep to standard MIDI tunings.)

We can see that these results are still not in the target MIDI range of 60 to 90. To achieve this, we simply need to add back the minimum value of the target range to each of the above results:

value within the range + mininum value of the range
$0.0 + 30 = 30$ (bottom of target range)
$10.0 + 30 = 40$
$30.0 + 30 = 60$

$50.0 + 30 = 80$
$60.0 + 30 = 90$ (top of target range)

We have now succeeded in converting the input source data 2 3 5 7 8 (source range is 6) to target MIDI values 30 40 60 80 90 (target range is 60).

This explanation and example have been very detailed. The reason for it is that proportion is such a vital topic, the very essence of data conversion techniques. More subtle conversions which skew the data by various means and for various (valid) reasons are built on this foundation. It should now make sense to see the whole operation in a single formula (after pre-calculating the input range scaling factor)[4]:

*Target-MIDI-value = (((datain - mindata) * input-range-scaling-factor) * target-range) + min-target-range*

In numbers:
$60 = (((5 - 2) * 0.166667) * 60) + 30$

This procedure will convert any data value to the (selected) MIDI range, as used for pitch or amplitude. It also works for frequency in Herz (wave cycles per second), say from 200Hz to 5000Hz (a range of 4800). Using the whole formula, let's see how the above input data values 2 3 5 7 8 come out as Herz within this range:

*Target-Hz-value = (((datain - mindata) * input-range-scaling-factor) * target-range) + min-target-range*

$y = (((2 - 2) * 0.166667) * 4800) + 200 = 200$
$y = (((3 - 2) * 0.166667) * 4800) + 200 = 1000$
$y = (((5 - 2) * 0.166667) * 4800) + 200 = 2600$
$y = (((7 - 2) * 0.166667) * 4800) + 200 = 4200$
$y = (((8 - 2) * 0.166667) * 4800) + 200 = 5000$

The calculations for amplitude or pan in the 0 to 1 range are also the same. But pan in the -1 range has to use a centre value and calculate it below it for Left of centre, and above it for Right of centre.

B. Discussion of mathematical principles

What is interesting here is the matter of ranges and proportions within ranges. In effect, we need to move from a data input *as a proportion of the data input range* to a musical value *as a proportion of the target range.* (AE: is this an accurate statement??)

The calculations do not work if the minimum data input value is not subtracted from all the input data values (so that one starts at 0). Why is this? What is happening here? And why does it help to normalise the input data (place it between 0 and 1)?

It is clear from the worked example above that the input calculations must use values

[4]Remember, the normalisming scale factor is $\frac{1}{(MaxInputValue - MinInputValue)}$

that remain within the range of input values. This range was 6, not 8, because the first value was 2. By subtracting 2 from all values, we in effect start at 0 and end at the top of the range. Otherwise, all the values are offset by 2 and do not reflect the true range of values, causing the resulting MIDI values to be too high. If the input data values began at 0, we wouldn't have to perform the subtraction — or when we did (to make coding generic), it would have no effect.

Normalising the values is optional. We could have done this: (inputvalue - 1stinputvalue) / range for each input value. By normalising we only have to perform a division once, which is more efficient, and it also clarifies what is happening by carrying out this first scaling operation when each data input (less the 1stinputvalue) is multiplied by the scaling factor. This was the first step above. In $y = kx$, x is the input data, k is the scaling factor, and y is the resultant part of 1 that the input datum represents.

The resulting normalised input data becomes the multiplier into the target MIDI range: i.e., we are determining which part of the target MIDI range a given input datum will use. This was the second step above. Again in y = kx, x is the target MIDI-Hz range, k becomes the *variable* output of step 1, and y is the resultant part of the target MIDI-Hz range that will be used. The lowest value of the target MIDI range is then added back in to move the values based on 0 to 1 into the correct target MIDI-Hz range.

To summarise, we are finding out what proportion of the input data range each input data value is, scaling this between 0 and 1. These generic proportions become multipliers (scaling factors) into the target range to get a corresponding set of proportions. The minimum value of the target range is added to the result to place the numerical values correctly within the target range. The same 1 to 127 MIDI range is used for both pitch and amplitude ('velocity').

C. Musical considerations and warping the values

There are times when it may make musical sense to warp the values during data conversion. This may happen when a value range covers an extreme range, such as from 0.2 to 1250. A direct linear conversion may put too many lower values too low for practical purposes, and the higher values beyond a sensible upper end, with not enough inbetween. On the other hand, the value range of the data may be too small to allow the ear to hear differences effectively, such as from 0.02 to 0.2. One solution for these situations is to warp the output values. A root calculation occurs exponents are less than 1. Exponents larger than 1 give us an expansion of values that may already be too close together.

- The first point to make is that warping is applied to normalised data. This provides a uniform starting point for the warping and other scaling involved in moving the values into a target range.

- When the exponent is < 1, small values (between 0 and 1) become bigger and big numbers become smaller. For example, $.1^{0.5} = 0.316$ (considerably larger), and $0.9^{0.5} = 0.94868$

(very little change). We therefore will hear the smaller original data values more clearly without pushing the higher original data values excessively high.

- When the exponent is > 1, small values (between 0 and 1) become smaller and large values change very little. For example, $.1^2 = 0.01$, and $0.9^2 = 0.81$. This could reduce the smaller original data too much, but it will make the higher values more audible, should this be important.
- This can now be spread into the target MIDI or Hz range for more audible results.

This section may be expanded later.

D. Handling the time element

Important overall constraints are that times in breakpoint files must start at time 0 (there are exceptions, not relevant here), cannot include duplicate times, and must ascend (time cannot go backwards). All times must be different, although just a miniscule difference (perceptibly simultaneous) is acceptable.

The timing of events can be handled in two different ways:

1. Ensure that the data being used for time is in ascending order, and sort it in this way if it is not. Each time becomes an onset for an event. Converting the data to a set of (ascending) start times provides one very audible way to hear the numerical distance between successive values, i.e., a rhythm. If the data is not already in ascending order and has to be sorted, this will rearrange the original order and may disconnect the time from the data in corresponding columns being used for other musical parameters. This disconnection will not occur if the data being used for time is already in ascending order. It also has the disadvantage of making the gaps between events become longer and longer, which may also not reflect the data accurately.
2. The second method involves treating the input data as event durations. The times start at 0 and each duration is added to the total value of all the events so far. Thus the order of the original data is maintained. We do this by calculating the distance between each successive pair of data items, turning that into duration in (parts of) seconds, and adding the duration to the previous start time.

There is a certain amount of arbitrary decision-making that goes into creating a set of times from data which is not in itself temporal. For example, the data from a collision in the LHC is for all practical purposes simultaneous — it all happens in a tiny fraction of a second. We want to spread this out in order to be able to hear the separate particle events, i.e., scale the values into a larger temporal range. Another reason for scaling the times is to produce a file of times to a specific duration so that it matches the length of a soundfiile that is going to be shaped by the data.

While any data can be used for times, it is worth asking whether some types of data are more approriate than others for this purpose. So far, the LHCsound Project has chosen to use distance in millimeters for times, because there seems to be some correspondence between spatial and temporal distance. This data comes already in ascending order.

Removing duplicate times also involves making a decision or two. First of all, duplicate times can be removed altogether. This implies that simultaneous events are of little importance and can be regarded as background noise, so that one event can justifiably represent all simultaneous events. If one wants to retain all the events, eliminating none, if only as an indication of how many there are, then a small amount can be added to subsequent simultaneous events so that each will have its own valid start time. This 'small amount' is another arbitrary decision: enough to make each event perceptible, but not so much that it separates events to a misleading extent.

Sometimes, while not simultaneous, some time-gaps between events are so small to be usable, either for software or hardware reasons. In these situations, an arbitrary amount needs to be added to the time — if it is to be retained — so that it becomes both valid and sufficiently audible.

The final decision regards the degree to which the resultant set of times should be compressed or augmented. If the numerical time-range is impractically long, it can be compressed into a reasonable length. If the resultant set of times bunches a lot of events very closely together, one may want to augment the times in order to hear each event separately.

E. Handling Pan

Musical Pan values have two different value ranges: one is 0 to 1, and the other is -1 to +1. It is possible for the values to be less and 0 or -1 or more than +1. Such pan values create the illusion that the sound is coming from a position further left than the left speaker, or from further right than the right speaker. For the time being, we are keeping within the 0 to 1 and -1 to +1 ranges.

The 0 to 1 presents no problems, because we are in effect 'normalising' the data: placing it between 0 (far left) and 1 (far right) – 0.5 is centre. We sort the data into ascending order so that we can easily fetch the highest data value, and then derive a pan-scaling-factor by 1 / highest-data-value. Finally we multiply each (unsorted) data value by this pan-scaling-factor:

1/highest-data-value = pan-scaling-factor e.g., 1 / 1437 = 0.000696 input-data-value * pan-scaling-factor = pan-value e.g., 718.5 * 0.000696 = 0.500076 (Centre position for the middle-value)

With the -1 to +1 range. the left side extends from -1 (far left) to 0 (centre) and the right side extends from 0 (centre) to +1 (far right). Our calculations need to place the pan on the correct side.

To do this we sort the data into ascending order and divide by 2 to find the data's middle-value. If a datum is less than the data's middle-value, we need to place the event on the Left. We also need to know how much less than the highest value in the original data it is in order to determine how far Left of Centre it needs to be positioned. Thus the formula becomes:

*left-pan-val ((highest-value - datum) * -1) * pan-scaling-factor*
$l - p - v = ((1437 - 10) * -1) * 0.000696 = -0.993192$(almost full Left)

If a datum is greater than the data's middle-value, we need to place the event on the Right. We also need to know how much less than the highest value in the original data it is in order to determine how far Right of Centre it needs to be positioned. In effect, we just leave out the -1 multiplier:

*right-pan-val (highest-value - datum) * pan-scaling-factor*
$r - p - v = (1437 - 10) * 0.000696 = (+)0.993192(almost full Right)$

F. Salient issues when coding sonifications in a programming language

You will have noticed the use of parentheses when laying out the longer formulas above. This is to ensure that the correct precedence is followed when performing the calculations, and are an essential part of the programming. As the calculations are applied to a list of numbers, the software must work through each line in the list, requiring one of the looping mechanisms.

The generic formula:

*target-MIDI-value = (((datain - mindata) * input-range-scaling-factor) * target-range) + min-target-value*

will therefore get coded something like this:

```
for(i = 0; i < listlength; i++)
target-MIDI-value[i] = (((datain[i] - mindata) * input-range-scaling-factor) * target-range) + min-target-value
```

The various essential tasks to master include:

- opening files *from which* to read the input data
- sorting lists to find the minimum and maximum values in the input data
- determining the range of the input data
- creating appropriate scaling factors
- creating event start time sequences for various purposes

- writing file formats that use both alphabetic and numeric characters, as might be needed by a particular sound-making program. For example, the file needed by CDP's **Texture Postornate**[5] (used to make melodies) has a particularly complex alpha-numeric format. *Csound* score files are also alpha-numeric.
- calling executable programs if using a shell script
- opening files *to which* to write the musical values, times, or other required text
- creating new procedures and switching mechanisms for optional ways to handle the data that you might want to create, such as additional ways to search the data or skew the output values for musical reasons
- remember to comment your code copiously, so that you can decipher what it is doing at a later date

You are invited to study the source code provided. These are written in Tcl/Tk[6], which use lists in a powerful but at first obscure way if you are familiar with C or *Tabula Vigilans*[7]

Our overall package is designed to provide every opportunity for entering as deeply as one wishes into the sonification process.

III - MUSICAL APPLICATIONS

Our overall topic is using data to shape existing sounds. Other approaches will create sound directly from the input data.

First we need to think about our musical objectives: towards what musical shaping goals can be apply the data? Here are some ideas that we have developed so far:

- create event start times
- turn the data into a melody of successive pitches by internally transposition-warping an existing soundfile
- turn the data into a melody in which each note forms a new iteration of the source sound
- turn the data into a harmony comprising simultaneous pitches by filtering an existing (timbrally complex) sound

[5]Written by Trevor Wishart

[6]Programmers new to TCL/Tk may wish to consult the excellent *Tcl and Tk Programming for the absolute beginner* by Kurt Wall. Thomson. Course Technology Series. 2008

[7]*Tabula Vigilans* is a programming language written by Richard Orton and designed especially for music. It is used for creating algorithmic score files and also as a real-time MIDI instrument.

- turn the data into a harmony in which each note of a simultaneity is a transposition of the source sound
- turn the data into a harmony as staggered multi-events mapped to a pitch field
- turn the data into a harmony by tuning the (timbrally complex) partials of an existing sound
- turn the data into a rhythm of timed events on one pitch
- turn the data into changing levels of amplitude on one pitch (or in step with the pitches of a melody)
- turn the data into a sonic entity made from its N loudest components
- turn the data into changing timbral densities
- combine one of the above with movement through stereo or multi-channel space

The above musical goals are achieved by means of a two-step procedure:

1. Convert the data to values required by the sound-making program and place in a file constructed appropriately. There are many things to consider: the value requirements of different musical parameters, whether integer or floating point numbers are to be used, the mathematical nature of the data-conversion routine, and a variety of file formats to implement. (See Section IV)
2. Run the sound-making program with the script of musical values. These may be simple operations or require a number of different files designed specifically for certain parameters. (See Section V)

Ideally, a Graphic User Interface will put all this together for the user in a reasonably intuitive manner. For the time being, we are summarising the whole environment in the following chart(currently in HTML format).

IV - SCRIPTS LIBRARY

The task of data sonification involves a number of components and steps. We would have formed some idea of what is involved by the above discussion of data conversion. The overall process can be summarised like this: Raw Data $\Rightarrow$ Data Conversion, producing equivalent files with musical value ranges, some as single columns and some as *time value* breakpoint files $\Rightarrow$ Files using the converted data in formats (sometimes complex) as needed by a given sound-shaping program $\Rightarrow$ Run the sound-shaping program to produce the audible output.

The Scripts Library is written in Tcl/Tk, a reasonably widespread and powerful scripting language, well-supported by the programming community. They can of course be rewritten in another scripting language, such as Java, Python or Perl, and adapted for use with sound-shaping or sound-making software other than the CDP sound transformation software. These scripts are therefore generic and designed for command line use. They can within the context of a graphic user interface, with minor modifications.

We are going to list the Scripts Library on the basis of inputs and outcomes in order to describe it in a task-orientated way. At the moment the LHCsound Project is using 'distance' for time, 'angular momentum' (= 'circularity') for pitch, and 'energy' for amplitude. However, there is no restriction in using the data in a different way. For example, one may want to hear the pitch trace or harmonic signature of the energy data, or the rhythmic signature of the distance or pitch data. The 'energy' data is also useful for musical pan effects because the occasional high-energy datum can be heard to swing rapidly across the aural field. Further scripting to filter the data in some way is also an important task for future work. Flexibility and experimentation is the name of the game.

All of the scripts provide output range controls.

1. CREATE A LIST OF EVENT START TIMES
   (a) **data2stimes.tcl** — TIME — This script will convert raw data into a set of times usable in a musical context. It has two modes: 1 - data reordered into an ascending list, and 2 - data treated as durations and accumulated to form the (ascending) times. Sometimes the raw data is already in ascending order and the Mode 1 reordering is just a fail-safe. The difference between Modes 1 and 2 is mainly in the reordering process, which can disconnect a time from other data relating to that event. Mode 2 ensure that this does not happen. A 'duration' parameter makes it possible to define how long in time the output will be, either to scale input data into a practical range or to make it match the duration of an input soundfile that is to be shaped.
2. WORKING WITH SINGLE COLUMNS OF DATA
   (a) **data2ampcol.tcl** — AMPLITUDE — A single column of raw data (such as 'energy') is converted to a single column of amplitudes. Both the MIDI 1 – 127 range and the 0 – 1 range can be used.
   (b) **data2durcol.tcl** — DURATION — The numerical distance between successive input data items becomes durations.
   (c) **data2fbank.tcl** — HARMONY — The data is turned into a file of *frequency amplitude-in-dB* values, which the sound-shaping program uses as a simultaneity to create a harmony by passing the input soundfile through the 'sieve' of the filter bank. Frequency can be in MIDI values or in Hz, and either form can be expressed as floating point numbers (Mode 1) or as integers (Mode 2). Duplicate values are removed. The result might aptly be referred to as the 'harmonic signature' of the data.
   (d) **data2hf.tcl** — HARMONY — Similar to data2fbank.tcl, this script just produces a column of pitches without duplicates as a harmonic field, a version required by some sound-shaping programs.

   (e) **data2pancol.tcl** — PAN — The raw data becomes musical 'pan' information, i.e., instructions on where to move a sound in space. Some programs require pan information along a -1 – +1 scale (Mode 1), while others require it along a 0 – 1 scale (Mode 2).

   (f) **data2pchcol.tcl** — MELODY — 'Melody' is used in a loose sense here. What is meant is that a pitch trace is produced from the data. As a horizontal series of pitches, it is like a melody, but may very well not have all the musical characteristics we have come to expect by the term. The results are, however, often surprising and provide an easy way to hear what is going on in the data. Mode 1 of this script produces a series of pitches, and Mode 2 produces a series of transpositions. The pitch trace produced is like a 'melodic signature' of the data.

3. WORKING WITH *TIME VALUE* — 'BREAKPOINT FILES'. This means double column inputs, one for time and one for the other parameter. These scripts do not use raw data for time, but rather use the file produced by **data2stimes.tcl**.

   (a) **data2ampbrk.tcl** — AMPLITUDE — The amplitude conversion functions just like data2ampcol.tcl. Amplitude breakpoint files are used for enveloping.

   (b) **data2durbrk.tcl** — DURATION — The duration conversion functions just like data2durcol.tcl. At the moment, there doesn't seem to be very much call for this script, but it is there as a resource.

   (c) **data2panbrk.tcl** — PAN — The breakpoint version is needed by programs that focus on implementing pan. As mentioned above, it can provide a very noticeable aural event when a sudden jump in value causes the sound to swing sharply to the right or the left.

   (d) **data2pchbrk.tcl** — 'MELODY' — Mode 2 (transpositions) in a breakpoint format provides a way to shape an existing sound in pitch over the course of its duration. It is important that the time file duration match that of the input soundfile. At the moment, there does not appear to be an application for Mode 1 (a breakpoint file with a series of pitches).

4. MULTIPLE FILE INPUTS ⇒ SINGLE COMPLEX OUTFILE — These scripts create multi-field files by which a number of parameters can be handled at the same time, thus utilising the multi-dimensionality of aural perception.

   (a) **data2csscore.tcl** — SCOREFILE — This script handles time (from data2stimes.tcl), duration, amplitude, pitch, rise and decay, producing a *Csound* score file. A suitable *Csound* orchestra file is required that will use an existing soundfile a input.

   (b) **data2mix.tcl** — MIXFILE — Mode 1 produces a Mono mixfile and Mode 2 produces a Stereo mixfile. Other modes can be added to produce multi-channel mixfiles. The same source soundfile can be used, or different input soundfiles. The main parameters handled are amplitude and pan. The mixfile enables timed entries and controlled overlaps.

(c) **data2txpoA/B.tcl** — NOTE DATA FILE — Mode 1 is without a harmonic field, and Mode 2 is with a harmonic field. The former enables a clear statement of the pitch trace, whereas the latter enables the introduction of a certain degree of (randomised) harmonic colouration. Times come from **data2stimes.tcl** and the pitches for the harmonic field, if used, come from **data2hf.tcl**. Pitch and amplitude use raw data and converted into a breakpoint format, and there is even the option to add pan movement as well. In the 'B' form (*data2txpoB*) the pitch and amplitude information is pre-formed, pitch from **data2pchbrk** and amplitude from **data2ampbrk** and can therefore be easier to use if the data is already prepared — but take care that it all comes from the same source: i.e., has the same number of lines of data. Because of its multi-dimensionality, this is a very powerful tool.

(d) **data2txtimed.tcl** — NOTE DATA FILE — The emphasis with this script is on the rhythm produced by the data: its 'rhythmic signature'. A note data file is produced that contains only time information, and the input soundfile will be initiated (from its beginning) at each new time. When the sound-shaping program is made to keep everything on one pitch, and when the sound used has a reasonably sharp attack, the rhythm of the data is easily perceived.

5. OTHER UTILITIES

(a) **datasort.tcl** — SORT DATA — There are three modes: 1 - ascending, 2 - descending, and 3 - retain duplicates. As any sorting required is handled inside the other scripts, this is probably useful only for diagnostic purposes.

(b) **listlenth.tcl** — NUMBER OF LINES — It may sometimes be useful to check the number of lines in a file, e.g., when it is important that different files to be used in the same application have the same number of lines.

(c) **randnums.tcl** — LIST OF RANDOM NUMBERS — This in effect produces a raw data file of random numbers, using TCL's **rand()** function. This random data can then be converted to musical values using one or other of the above scripts, thereby providing a point of comparison between the purely random and the other data being used for sonification. It could be useful to expand it to include a variety of types of 'random': white noise, pink noise, brownian movement etc.

V - RUN SOUND-SHAPING PROGRAMS WITH THE FILES PRODUCED BY THE SCRIPT OUTPUTS TO SHAPE EXISTING SOUNDS

We now turn our attention to exploring some software that is able to make use of these 'musical' numbers to shape pre-existing sound material (whether that sound material comprises sampled sounds or sounds that are computer-generated). As mentioned in the Introduction, the use of pre-existing sound material makes it possible to use different sounds to bring out different aspects of the data. It also provides plenty of scope for composers to work with the data in a freer and perhaps more emotive way.

The scripts that focus on data conversion are of a more generic nature. The scripts that write complex files need to be in the specific format required by the application that will use them. The present author uses the CDP Sound Transformation and the *Csound* software to accomplish the actual sonifications. The scripts can be adapted to use other suitable software.

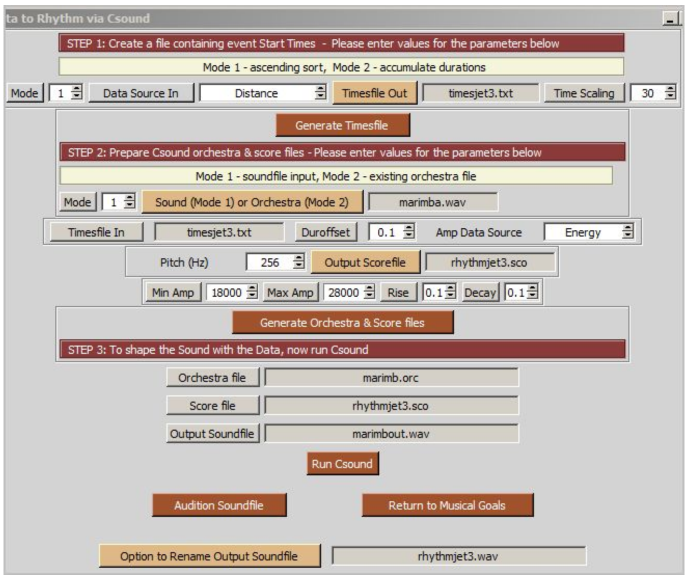
The programs listed below can be used directly in a command line environment or via one of CDP's graphic user interfaces (*Sound Loom* (MAC or PC) or *Soundshaper* (PC only). They are also used by our own sonification GUI *SoundingData*. *Csound* is public domain and readily available via **http://www.csound.com** (downloaded from Source Forge). The CDP software is not public domain and is available from **http://www.composersdesktop.com**.

- **Csound** - This is the application that uses the scorefile produced by **data2csscore.tcl**. The command line is quite simple: `csound -s -osoundout.wav -W crotale.orc csj1m1.sco`. The '-W' specifies wav file output. The orchestra needs to be able to handle using an existing soundfile as input. This requires the 'flooper2' op available in Version 512. More information about this is in the Manual for *SoundingData*.
- **ENVEL USERBANK** - This is a CDP program that takes the filter bank file produced by **data2fbank.tcl** is ENVEL USERBANK. Mode 1 handles floating-point values, Mode 2 integer values. There are also parameters for 'Q' (sharpness of the filter, i.e., more focused on the specified pitches), 'Gain' (usually raised to compensate for the filtering), 'Tail' (to handle the way the sound ends) and a switch for double-filtering, currently hard-wired to be on because it produces a cleaner harmony.
- **TEXTURE TIMED** - This CDP program uses the note data file produced by **data2txtimed.tcl**. It has many other parameters, most of which are filled in as defaults in *SoundingData*, and the user is recommended to study the Reference Manual provided either in the CDP documentation or in the Manual for *SoundingData*, though this GUI is reasonably self-explanatory.

VI - *SoundingData* - A Tcl/Tk GUI APPLICATION

Our own GUI for shaping existing sounds from input data streams is called 'SoundingData'. It is written in Tcl/Tk and provides a front end for the whole sonification process, from data conversion scripts drawn from the Scripts Library to the sound-shaping programs that realise the audio output. Although written for handling data from the Large Hadron Collider, the present sonification environment should be sufficiently generic to handle other data sources as well. The current version of the GUI exists on its own as public domain software, and it is also included as an extra module in Wellspring Music's **ProcessPack**, for the convenience of its users, and where its scope will continue to be expanded. Because CDP programs are involved, both CDP users and **ProcessPack** users will find *SoundingData* to be a powerful addition to their toolset.

This is the current Dialogue Window for creating a rhythmic signature:



ta to Rhythm via Csound

STEP 1: Create a file containing event Start Times - Please enter values for the parameters below

Mode 1 - ascending sort, Mode 2 - accumulate durations

Mode: 1 | Data Source In: Distance | Timesfile Out: timesjet3.txt | Time Scaling: 30

Generate Timesfile

STEP 2: Prepare Csound orchestra & score files - Please enter values for the parameters below

Mode 1 - soundfile input, Mode 2 - existing orchestra file

Mode: 1 | Sound (Mode 1) or Orchestra (Mode 2): marimba.wav

Timesfile In: timesjet3.txt | Duroffset: 0.1 | Amp Data Source: Energy

Pitch (Hz): 256 | Output Scorefile: rhythmjet3.sco

Min Amp: 18000 | Max Amp: 28000 | Rise: 0.1 | Decay: 0.1

Generate Orchestra & Score files

STEP 3: To shape the Sound with the Data, now run Csound

Orchestra file: marimb.orc

Score file: rhythmjet3.sco

Output Soundfile: marimbout.wav

Option to Rename Output Soundfile: rhythmjet3.wav

From time and data files to sound-shaping application

SoundingData enables you to:

- select a soundfile to be shaped with the data (one long soundfile can be shaped, or a shorter soundfile can be used repeatedly to create a separate sound for each data item)
- select the input data file
- name the output data and sound files
- select a sonification process and enter the required parameters
- run the sonification process, which will write the appropriate output data file and call the corresponding sound-making program
- audition the result

Please see the HTML Reference Manual for *SoundingData*.

Glossary

- **Amplitude** - a measure of loudness (volume)

- **CERN** - The European Organisation for Nuclear Research located near Geneva, Switzerland. The instruments used at CERN are particle accelerators and detectors forming a ring 17 kilometers long. About 3000 people work at CERN.
- ***Csound*** - comprehensive sound synthesis and processing packaged developed initially by Barry Vercoe at MIT, and subsequently much expanded in the public domain by many distinguished contributors. It now includes real-time performance facilities.
- **Data Conversion** - the process of converting numerical data from one range of values to another
- **Linear** - there is a 1-to-1 relationship between different value ranges
- **LHC** - the Large Hadron Collider, one of the three particle detectors at CERN
- **MIDI** - 'Musical Instrument Digital Interface': an industry standard for expressing musical data for synthesisers and other electronic instruments
- **Normalisation** - the process of converting any value range into a range between 0 and 1. It is a simple calculation: divide 1 by the range of values and muliply each value by this factor.
- **Orchestra File** - a text file (.orc) containing sound handling information for *Csound*
- **Score File** - a text file (.sco) containing note event information for *Csound*
- **Sonification** - the process of turning numerical data into audio format
- **Warping** - normalised values are adjusted in some way before moving them into the target numerical range. We have been concentrating on raising numbers to powers $>$ or < 1 as a strategy for bringing into greater audible clarity certain parts of the data. Most useful with the LHC data appears to be compression of differences (values < 1), which makes small numbers bigger in relation to large numbers.

Last updated: 31 January 2011