A tuning strategy for VOLTAGES

Paul Darlington
m0xpd
paul@appledynamics.com
https://github.com/m0xpd/TuningStrategyForVoltages

Overview

This document presents a tuning strategy for the "VOLTAGES" expander for Music Machine Modular's "Turing Machine" random looping sequencer. The strategy allows the user to find settings in which VOLTAGES can generate 12-tone equal-temperament ('12TET') sequences which otherwise do not arise naturally from this system without the use of a quantiser. The methods presented are applicable to other "Klee-type" sequencers.

In addition to describing the tuning strategy, this document presents methods for calculating not only all possible pitches of the notes within the sequences produced by the VOLTAGES expander, but also the probability of their occurrence. This probability is important because, in the absence of other cues (such as other harmonising voices), the distribution of pitches plays a part in establishing tonality.

The methods work both in the semitone tunings advocated in the tuning strategy and in more general 'microtonal' tunings. These methods include an interesting approach based on convolution, which appears to offer advantages in computation and in a clearer understanding of the operation of "Klee" sequencers. There is computer code to support all the mathematical models. There is also a catalogue of representative tunings, demonstrating VOLTAGES' ability to make sequences based on diatonic and symmetric scales, modes, chords, and other musically useful patterns.

The document also includes analysis of significant non-linearity in the response of VOLTAGES' sliders, which does not help in setting up tunings. This is addressed by a simple circuit modification which can remove the non-linear response.

The document presents a large volume of material which some readers will find unnecessary or overwhelming. If you are just interested in the tuning method, go straight to section 5, dive into the practical 'first steps' and then try examples from the catalogue of tunings in section 13.

If this is enough to whet your appetite, you could come back and look at the rest later. Otherwise, just enjoy VOLTAGES' ability to work as an independent source of 12TET sequences!

Contents

1 Introduction

'VOLTAGES', an expander for Tom Whitwell's popular 'Turing Machine' random looping sequencer, exists to generate voltage sequences. Whilst these sequences can sound interesting and pleasant as pitch profiles, they are usually applied to a quantizer to recover the familiar structure associated with melodic patterns. VOLTAGES' sliders offer the user opportunity to control the weighting applied to the Turing Machine's gate sequence, such that it is possible to arrange for musically useful 12TET sequences to be created directly from VOLTAGES, including those based on diatonic, modal, and symmetric scales and arpeggiated chord sequences. This document describes a tuning strategy which gives access to these sequences, gives details of the tunings which produce the sequences, and explains the theoretical basis for the strategy.

The strategy is based on establishing a 'main interval' for the sliders which is usually an octave, (making an octave / 1V change between the 'off' state when the slider is hard against its left limit stop and the 'on' state when the slider is at the other end of its travel) and then deliberately setting sliders at carefully chosen semitone intervals. A model of the pitches in the sequence produced by VOLTAGES in any such configuration is presented, using a graphical approach, which will develop understanding and allow you to visualise how VOLTAGES builds up sequence pitches. This is accompanied by a mathematically efficient formulation. This model allows the identification of useful tunings, many of which are listed in a 'catalogue'.

Before describing the tuning strategy, some background material on the Turing Machine and the VOLTAGES expander is presented. This is not here to suppose that you don't know about these modules – rather it is here to establish a 'language' for what follows. The math might look a bit heavy, but it isn't. All you need to take away from the background sections is the simple idea that VOLTAGES (and the Turing Machine) generate their outputs by combining the gate sequence (*i.e.* the flashing LEDs) with WEIGHTS. These 'weights' are fixed in the Turing Machine. But they are under user control in VOLTAGES, courtesy of those eight sliders, which we'll be adjusting soon.

Before all that, perhaps we might ask if we really need a 'tuning strategy' for VOLTAGES. The truth is – of course - we don't!

2 Justification

It is evident that people understand how "VOLTAGES" works and how to use it. But it is also clear that most people do not want to over-analyse the operation of modules, preferring instead to experiment creatively and to be taken musically where interaction leads. A thread on the Modwiggler forum [1] discusses the "VOLTS" expander, demonstrating that understanding (VOLTS is VOLTAGES' smaller sibling).

In that thread, 'Bregnier' wrote:

> "*Volts is very much a 'play with it and find out' set up - if you're looking for predictable behavior behind tweaks you probably want to look elsewhere*."

In the same thread, 'Bregnier' had already (wisely) said:

> *"Don't confuse knowing how it works technically …. with knowing how to use it".*

Bregnier clearly has complete understanding of how VOLTS works but sees the distinction between a 'technical understanding' (of the sort which might be analysed and used to set up tuning strategies) and a more profound 'ownership', which is founded on knowing *why* VOLTS works and how that can be used artistically.

Bregnier advocates playing with VOLTS and sees play as *means to an end*.

My friend and co-conspirator, Pete, speaks eloquently of the "therapeutic" benefits of interacting with synthesisers (and sequencers in particular) and advocates mindful play with these devices. I agree; I spend hours playing with the knobs and seeing what happens. I spend even more hours "hands-off", listening to the evolution of the resulting patterns, bathing in the "therapy" and pleasure it brings. Pete advocates playing with sequencers, including VOLTAGES, and sees that play as the same *means to an end* described by Bregnier but, additionally, Pete sees play as directly rewarding and therapeutic.

According to Pete, play can be an *end in itself*.

A further, valuable dimension of my own play with synthesisers is the engagement of other parts of myself, bringing into my play not just a musician but also an engineer, wanting to intervene with and design circuits and modules, and a dreamer, wanting to understand how it can be that "logic and mathematics" creates the sounds that heal and entertain me.

Synthesisers are, for me, a special (perhaps a unique) playground, in which the 'toys' prompt a particular kind of 'play'. The play is sometimes the exploratory, *means to an end* engagement advocated by Bregnier, that I also indulge in on my guitars and other instruments. It is sometimes the *end in itself* quest for therapy advocated by Pete, that brings me pleasure, and keeps me (nearly) sane. But it is also a different kind of *end in itself* play; the play of an engineer, conducting 'tests' and 'experiments' and asking: "what if"?

That "analytical", engineer's play prompts me to look for what becomes a 'tuning strategy' – but I am still happy simply to "play with the knobs" and see what happens!

3 Background: *the Turing Machine*

Music Thing Modular's 'Turing Machine' [2] is a 'Looping Random Sequencer',

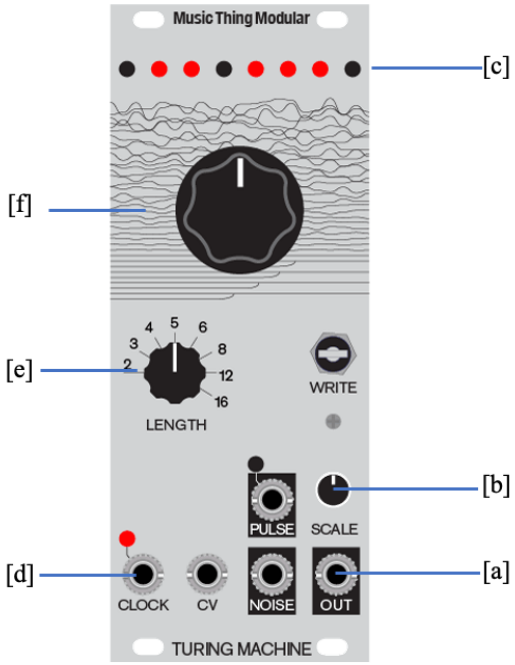


Figure 1 The Turing Machine

which produces a voltage sequence from its output [a]. This output *TuringOut* is proportional to:

$$TuringOut \propto scale.[G]^T[W] \qquad (1)$$

where:

scale is a factor associated with the trim pot 'SCALE' [b],
G is a column vector of Boolean gates (8 of which show on the LEDs [c])

$$G = \begin{bmatrix} gate_1 \\ \dots \\ gate_8 \end{bmatrix}$$

and

W is a column vector of weights.

> This proportionality (equation 1) only works if the SCALE control isn't set too high.
> If SCALE is set too high, the output will saturate at the maximum level when several gates are active, and the proportionality will be lost.

The Gate vector is maintained in a shift register by triggers applied to the CLOCK input [d], such that, at each trigger index, k, the gate vector G is updated:

$$gate_{n,k} = gate_{n-1,k}|_{n>1} \qquad (2)$$

The Looping feature is established by feedback to $gate_1$ from a gate selected by the LENGTH switch [e]. This feedback is controlled by the large rotary control [f] which allows the user to scale and optionally invert the feedback (inversion causes the loop length to double) and to mix in a random signal, R, following (in principle):

$$gate_{1,k} = round(\pm\alpha.gate_{length,k} + (1-\alpha).R) \qquad (3)$$

where α is derived from the main control [f].

Importantly, in the Turing Machine, the weight vector W which forms the output signal from the instantaneous gate values is realised as the input weights to a DAC. This is an exponential series: 1, 2, 4, …. 128.

Whilst an exponential series is important in musical pitch relationships, applying an exponential series in a context where pitch already is represented in a 1V/octave system makes the weight vector choice in the Turing Machine of limited use for the direct generation of melodies. The Turing Machine generates attractive pitch profiles and useful exponential patterns, but it is not immediately 'tuneful'. It does not produce 12TET pitch sequences; rather it is micro-tonal. Most examples of use on the internet revert to the use of pitch quantizers [3].

4 Background: *the VOLTAGES Expander*

Music Thing Modular's 'VOLTAGES' module [4] is an expander, developed for use with the Turing Machine.

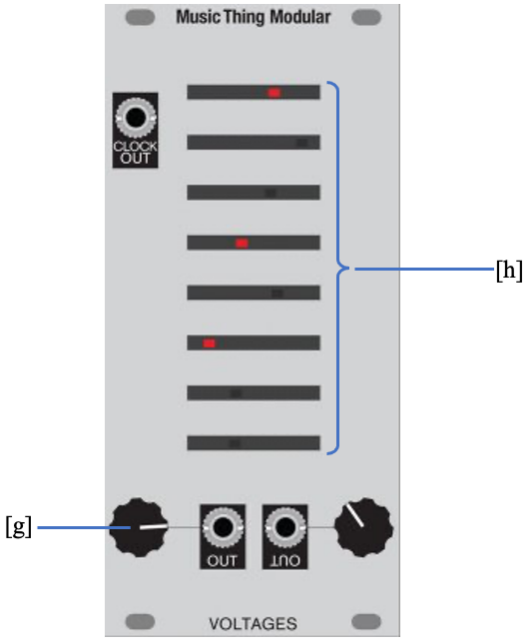


Figure 2 The VOLTAGES Expander

Like the Turing Machine, VOLTAGES produces an output signal in response to the gate vector *G*, which is passed from the Turing Machine to VOLTAGES. It is also controlled by the same type of equation (see equation 1):

$$VoltagesOut \propto scale_V . [G]^T [W_V] \qquad (4)$$

VOLTAGES takes the gate vector and operates on it with a different local scaling factor, $scale_V$, and a different weight vector, W_V.

The scaling factor $scale_V$ is obtained from VOLTAGES own SCALE potentiometer [g] and the weight vector W_V is derived from a set of eight conspicuous slider potentiometers [h], provided exactly for this purpose.

> Again, VOLTAGES' output will saturate if its SCALE control is set too high, the sliders are pushed too far to the right, and several gates are active, violating the proportionality of equation 4.

This is not an entirely new idea; Tom gives credit to the inspiration behind VOLTAGES by calling out Scott Stites' "Klee Sequencer" [5, 6] and the "Triadex Muse" on the silkscreen of VOLTAGES' PCB.

With the independent scale control and freedom from the exponential weight vector, VOLTAGES offers the user the *potential* to explore weight vectors W_V which produce more 'musical' pitch sequences than those produced by combining the gate signals with the fixed exponential weight vector in the Turing Machine.

However, this potential is not easy to realise and, again, many users [7] revert to quantizers to regain control of pitch sequences generated by VOLTAGES, just as they did with the original Turing Machine. The significance, operation, and selection of the slider settings (/weight vectors) in generating pitch sequences directly from VOLTAGES, with special focus on the generation of 12TET 'musical' tunings, is the subject of the remainder of these notes.

This all started not with a grand design or invention or a 'piece of maths'; it started with an observation of how I was tuning my own VOLTAGES expander. I realised that I was doing something that was working quite well. Then I looked at what I was doing and figured out how and why it worked. I found it could work a whole lot better than I realised and could do a lot more than I dared hope. The first step was important, and I'll share it now, in the next section.

5 First Step toward a Tuning Strategy

I'm assuming you have a Turing Machine and the VOLTAGES expander set up in front of you, with the output from VOLTAGES going to a VCO. If you don't, you can see examples in videos (indexed here), or simply follow it as a thought experiment.

Set up your Turing Machine with a repeating, one-bit pattern (Tom shows you how to do this in his introductory video [8]).

Then set all the sliders hard off (*i.e.* to the left hand end of their travel) except one – perhaps the top one – which should be fully on (*i.e.* to the right hand side). Make sure all the sliders are right on their end stops, as any small error will mess up the tuning.

Listen to the resulting pitch sequence (with the Turing Machine clocked slowly – perhaps around 2Hz). Then adjust VOLTAGES' SCALE control until you hear exactly an octave change between the pitch on the "active" step (when the slider is set full 'ON') and the others (when the sliders are set to 'OFF'). You need to be able to hear this interval by ear and you need to tune it carefully.

You will find that the SCALE control must be set quite low to achieve this octave and that the SCALE control has poor resolution around the octave setting, making tuning of this important interval rather difficult. Spend some effort in getting this step as close as you can.

Here's a photo of my VOLTAGES module in this configuration:

Figure 3 Setting an Octave on the First Slider

Notice that slider 1 is hard right, all the others are hard left and that the SCALE control (bottom left) is at about “9 o’clock”. You can ignore the other control (“OFFSET”, at bottom right), as we’re not going to be using the inverted output.

If you now look at the voltage output on an oscilloscope, you’ll see this sort of trace:

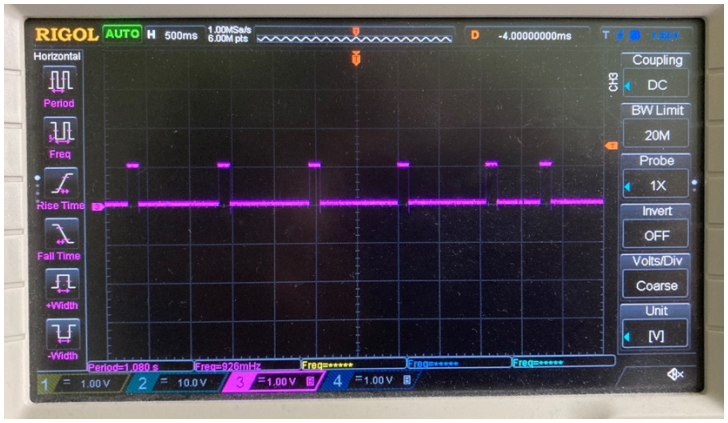


Figure 4 Output from VOLTAGES in configuration of Figure 3
(with single bit repeating pattern from Turing Machine)

Notice that the scope is set to 1V per division and that the voltage is pulsing up to 1V from 0V.

You now have the range of the sliders on VOLTAGES set to one octave (or 12 semitones) – think of that as the ‘Main Interval’. We could have chosen other values for this interval (and we shall do so later) but – as we shall soon see – an octave is a convenient value, because it is going to let us add an octave (to increase the span of a sequence) just by flicking an unused slider from fully “off” to fully “on”.

Now, if you change the main control on the Turing Machine away from the 5 o'clock position towards upright, to allow the gate pattern to change from the one-bit pattern, you'll get a random sequence, comprised of two notes – but they're the same notes you had playing as a regular octave a moment ago. Boring!

> Once you get used to it, setting up SCALE to produce the main interval octave is easy, even with a random gate pattern running. And sometimes – once you're experienced - it will help to have more than one slider set to fully "on", as you will be able to hear the tuning of multiple-octave jumps more accurately.

So – let's go back to the one-bit regular gate pattern on the Turing Machine for a moment.

Now, on VOLTAGES – in addition to slider 1, which you already have set to an octave - set another slider (perhaps slider 3) on a fifth (that's 7 semitones) and a third slider (perhaps slider 5) on a minor third (three semitones) above the tone sounded in response to all the other sliders which remain in the 'off' position.

Here is my VOLTAGES configured for this pattern:

Figure 5 Setting sliders to an Octave, a Fifth and a minor Third

At this point you'll probably notice that whilst the sliders in voltage are linear taper, they certainly do not behave in a linear fashion, (see Appendix A) making it difficult to set values (wait until you try to set a minor seventh!). It is difficult to set the tuning. Give it your best shot and remember – this can be fine-tuned as we go and it's all for fun! I'll show you how you can fix the non-linearity in Appendix B – but that's for another day.

Now the ‘scope trace looks like this:

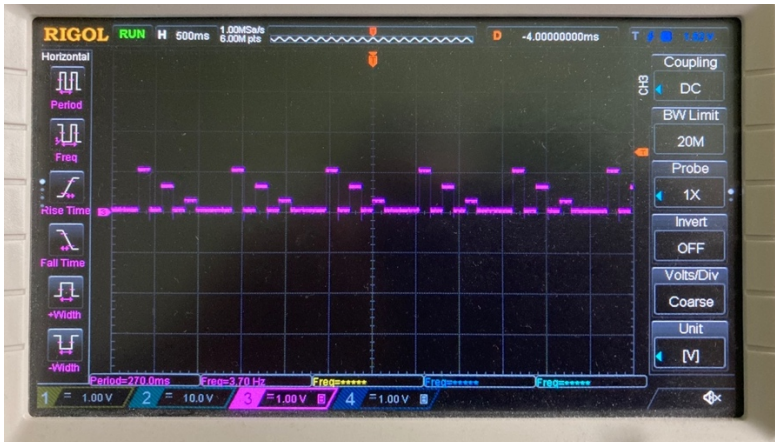


Figure 6 Output in Configuration of Figure 5
(with single bit repeating pattern from Turing Machine)

with its repeating three notes (as well as the continuing ‘root’).

Now – release the Turing Machine to make a random gate sequence and you’ll have something much more interesting. A pitch sequence of up to 8 notes which (if you tuned your slider accurately enough) will be recognisable as randomly arpeggiated inversions of a minor 7th chord.

Here’s the theoretical distribution of these notes (assuming our Turing Machines produce unbiased gate patterns, which they might do over the long haul with the main control in the upright 12 o’clock position):

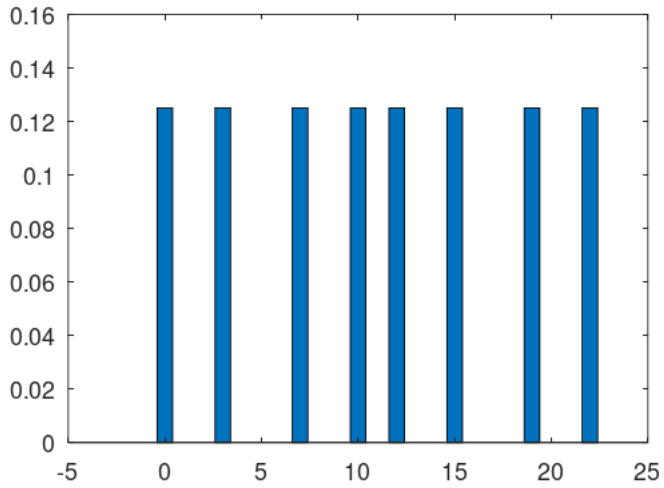


Figure 7 Probability Distribution of a minor 7 Sequence (see text)

The x axis is the note (the axis labels show the semitone number above the root, which is semitone ‘0’) and the y axis is the probability density, which we’ll discuss later – for the moment you can see that all notes are equally likely. Understand that this is NOT the probability distribution when the one-bit sequence is playing – rather it is the distribution that

is expected when the main control is in the “12 o’clock” position and a random sequence of gates are fed to the VOLTAGES expander – see sections 10 & 11.

Now, take any one of the sliders that are currently ‘off’ and slide it over to the fully ‘on’ position (being careful not to knock any of the carefully tuned sliders – perhaps it would be safest to use slider 8).

You will hear the entire sequence grow in pitch span, whilst still being an arpeggiated min7.

Here’s the new probability density:

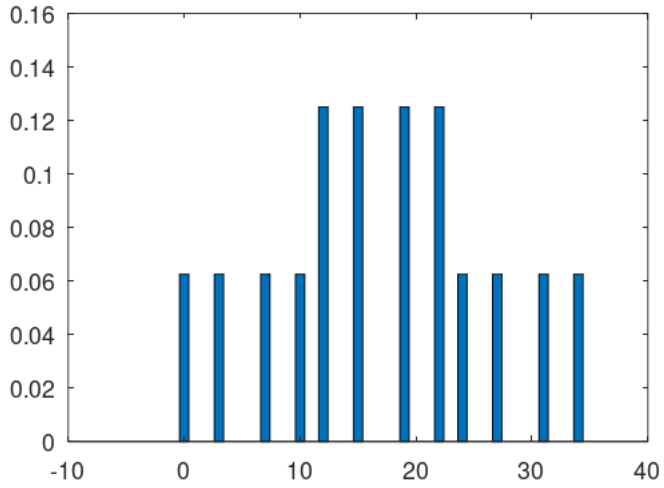


Figure 8 Probability Distribution of a minor 7 sequence with additional Octave Span

Notice it has lost its flat probability density - you’ll understand why later (in section 10).

Here’s part of the pattern I was getting, as seen on my ‘scope:

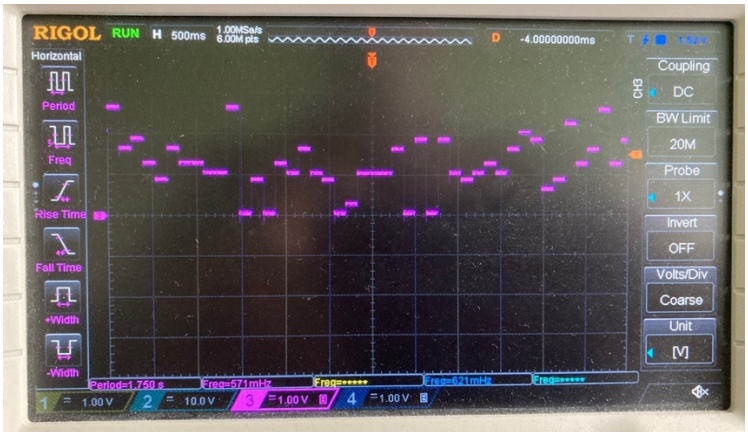


Figure 9 Output in Configuration of Figure 8
(with random pattern from Turing Machine)

You can even pull up another inactive slider from 'off' to 'on' and add another octave of pitch span into the sequence – you'll expose any errors in your original tuning as you do this.

> My sequence with three added octaves of span still sounded reasonable, but I would want to tweak it if it was going to be recorded or otherwise 'show its face in public'.

So – you've seen how to 'calibrate' the sliders on VOLTAGES to a practical 'Main Interval' and how simple it is to generate a complex pattern that arpeggiates notes from a min7 chord over a controllable pitch span.

Now let's take time to understand how this is working, so we can figure out how to generalize the idea to a practical tuning strategy.

6 Transforming from Voltage to Pitch

To further understand the operation of 'VOLTAGES' in the setting we've seen above we shall simplify equation (4).

We saw equation (4) describing system operation in the 'voltage domain'; voltage was the output variable. However, 'calibrating' the sliders to our 'Main Interval' in the previous section allows us to work directly in pitch, with weight vector settings in semitones.

Equation 4 is reproduced here (for convenience):

$$VoltagesOut \propto scale_V . [G]^T [W_V] \qquad (4)$$

Expanding the 'scalar product' of the two vectors allows this to be written in the more accessible form:

$$VoltagesOut \propto scale_V . \sum_{i=1}^{8} gate_i weight_{V,i} \qquad (5)$$

Our experiments in 'calibration' in the previous section set the local SCALE control and the sliders to establish relationship between the $scale_V$ parameter and the $weight_V$ vector such that this is equivalent to:

$$Pitch = \sum_{i=1}^{8} gate_i sliders_i \qquad (6)$$

where *Pitch* is the pitch (in semitones) associated with the output of VOLTAGES, $gate_i$ is the *i'th* Turing Machine gate, and $sliders_i$ is the setting of the *i'th* slider IN SEMITONES according to our calibration.

> Actually, *Pitch* is *VoltagesOut* divided by 0.08333V; the voltage representing one semitone.

Let's try to make it even clearer – especially for those who don't like the summation in equation (6). If we expand the summation of equation (6) it will result in a longer expression – but a simple one:

$$Pitch = gate_1 sliders_1 + gate_2 sliders_2 + gate_3 sliders_3 + gate_4 sliders_4 + gate_5 sliders_5 + gate_6 sliders_6 + gate_7 sliders_7 + gate_8 sliders_8 \quad (7)$$

Equation (7) makes it obvious that the output of VOLTAGES – the PITCH output – the note produced in our pitch sequence - is formed of eight terms. Each term is a gate value times a slider value.

Now – the gate values are set by the Turing Machine. And they are Boolean, On/Off values. They have 'numerical' value of 1 or 0, so they make the terms of the right-hand side of equation (7) either "1 multiplied by a slider value" (which equals the slider value) or "0 multiplied by a slider value" (which equals zero).

So, equation (7) really says that the output of VOLTAGES – the PITCH output - is formed by up to eight slider values which are added together. The selection of which of the 8 slider values will be added together is made by the Turing Machine in its determination of the gates. The Turing Machine is effectively turning on (or 'activating') some of the sliders in setting the gates.

What's more, since the slider values are now directly encoded in semitones (thanks to our efforts in calibrating the 'Main Interval' in the previous section), the pitch output is made of up to eight pitches which are added together. The pitch sequences built by VOLTAGES are constructed by randomly adding together up to eight pitches. Our task is now to select those pitches (slider settings), such that they add up usefully.

> Since we want to construct a 12TET sequence, it is necessary that the slider settings themselves must be at ET semitone tunings, such that their sums also will fall on ET semitones. This is a fundamental rule of the tuning strategy.

Since the gates are determined by the Turing Machine – which are effectively out of our control – we can go forward characterising the whole process using only the slider settings.

7 From Slider Settings to Sequence Generation

Now we have seen how *weights* can be transformed to *sliders* vectors in semitones, we can look at how the VOLTAGES module generates other notes than those implied by individual sliders. We shall look first at the examples in section 5, before trying to establish more general rules.

We started with the sliders set to an octave, a fifth and a minor third. This corresponds to the vector:

$$sliders = [12, 0, 7, 0, 3, 0, 0, 0] \qquad (8)$$

To make things easy, we'll assume that the Turing Machine is generating a fixed gate pattern.

If there are no gates active in the pattern generated by the Turing Machine (*i.e.* if all the LEDs are not lit), then the output *Pitch* (see equation 7) will be one semitone value:

$$Pitch_0 = [0] \qquad (9)$$

If there is only one gate active in the pattern generated by the Turing Machine, then there are two possible cases to consider. If the one active gate is hitting one of the sliders which are set to zero, then the output is the same as we already have for the $Pitch_0$ solution. Or, if the one active gate is hitting any of the non-zero gates, in which case the output *Pitch* (see equation 7) could be one of three possible semitones:

$$Pitch_1 = [12,7,3] \qquad (10)$$

If there are two gates active in the pattern generated by the Turing Machine, then there are three possible outcomes to consider. Firstly, the two active gates might both hit sliders with zero settings, in which case we repeat the $Pitch_0$ solution. Or, secondly, one of the two active gates might hit a slider with zero setting and the other hit a slider with a non-zero setting, in which case we repeat one member of the $Pitch_1$ solution. Or, finally, both active gates might hit sliders with non-zero settings, in which case the pitch output is a sum of those two settings, So, $Pitch_2$ is the set of all possible sums of two non-zero slider settings in *sliders*.

In this case, the set of all possible sums of two non-zero slider settings is:

$$Pitch_2 = [19,15,10] \qquad (11)$$

> We will see at how to solve problems like 'all possible sums of two slider settings" and the more general "all possible sums of *n* slider settings" in Appendix E, when we take a look at the "old school" approach.

If there are three gates active in the pattern generated by the Turing Machine then there are four possible outcomes to consider.

This time, I'll enumerate them to make it clearer:

1) If the three active gates hit sliders with zero settings, we repeat the $Pitch_0$ solution, outputting $Pitch = 0$
2) If two of the three active gates hit a slider with a zero setting and one hits a slider with a non-zero setting, we output one member of the $Pitch_1$ solution.
3) If one of the three active gates hits a slider with a zero setting and two hit a slider with a non-zero setting, we output one member of the $Pitch_2$ solution.
4) If all three active gates hit sliders with non-zero settings, the pitch output is a sum of those three settings. In this example, *sliders* has only three non-zero elements, so there is only one way of adding these three terms; $Pitch_3$ is (in this case) the sum of all the non-zero slider settings in *sliders*:

$$Pitch_3 = [22] \qquad (12)$$

Having more than three active gates in the pattern doesn't change anything in terms of pitch generation (although it certainly changes other aspects of the resulting sequence). So – we can now say what pitches are generated from our original setting for *sliders* by taking all elements from $Pitch_0$ to $Pitch_3$:

$$Pitch = [0]||[12,7,3]||[19,15,10]||[22] \qquad (13)$$

where the vertical line ('|') denotes concatenation (*i.e.* listing the elements one after another in a longer vector).

You will see that these elements coincide with those in Figure 7, as the procedure used in the calculation of Figure 7 follows the logic above.

You might like to walk through those steps again to make sure you grasped the argument. It is important, because it is the key to understanding how VOLTAGES (and, indeed, the Turing Machine itself) makes complex CV waveforms from only a few settings.

The important thing to grasp is that VOLTAGES can only produce a FINITE number of allowed pitches (/VOLTAGES) and that this number is dictated by the number of sliders that are in "non-zero" settings.

If you have no sliders in non-zero settings (*i.e.* they're all off), you'll get only ONE output value. If you have one slider in a non-zero setting, you'll get an output sequence with TWO values. If you have two sliders in non-zero settings, you'll get an output sequence with up to FOUR distinct values and (as we saw in the example we worked through above) if you have three sliders in non-zero settings, you'll get an output sequence with up to EIGHT distinct values. Perhaps you see a pattern emerging?

Remember you have 8 sliders… VOLTAGES can certainly make rich patterns – but it is the purpose of these notes to intentionally limit the notes of the sequences to lie on musically meaningful scales, chords, and series. Accordingly, well only be using a few of the sliders at a time.

Music Thing Modular produces a 'stripped down' version of VOLTAGES, called "VOLTS". It has only five sliders but – for the reasons above – it can still generate sequences with rich amplitude distributions.
"Volts" is, however, less useful in our application, as it has no 'scale' control, making it harder to set the 'Main Interval' range to a practical value.

Next, we'll look again at what happened when (in section 5) we added another non-zero slider tuned to the octave, (which I suggested should be on the last slider, so you didn't accidentally nudge your other carefully tuned sliders) to give:

$$sliders = [12, 0, 7, 0, 3, 0, 0, 12] \qquad (14)$$

We'll follow the logic above, but now you're familiar with it, we'll do it without so many words. We'll introduce a variable g for the number of active gates.

We start with the obvious "no active gates" case, g=0, which remains as before:

$$Pitch_0 = [0] \qquad (15)$$

There's a new non-zero slider, so you might have thought that there should be a new member when g=1. But notice that we already have 12 there as a pitch, so there's no point in repeating it in the list of pitches (we will deal with this type of repeat occurrence separately in calculating the probability density in section 10). So, this too stays the same:

$$Pitch_1 = [12,7,3] \qquad (16)$$

For two active gates, g=2, we do have a new possibility to add, but also some repeats, which are discounted:

$$Pitch_2 = [24,19,15,10] \qquad (17)$$

For three active gates, g=3, we again have new possibilities to add, but also some repeats, which are discounted:

$$Pitch_3 = [22, 31, 27] \qquad (18)$$

Finally, for four active gates, g=4, the sum of all non-zero sliders is different since we added the additional octave:

$$Pitch_4 = [34] \qquad (19)$$

When you switched the Turing Machine to a random sequence with the setting for sliders (see Figure 9), the set of pitches in the sequence is now obtained by concatenating $Pitch_0$ to $Pitch_4$:

$$Pitch = [0,12,7,3,24,19,15,10,22,31,27,34] \qquad (20)$$

These pitches correspond with those which appear in Figure 8.

8 Ordering and Visualising Pitches

It is rather difficult to understand the significance of the members of the *Pitch* vector when it extends over several octaves. To help with this, we will take a detour in this section to introduce a technique to transform all the pitches back down to the original octave in which the sliders operate and a visualisation method to allow us to 'see' the set of notes produced by the sequence. We shall start this with two equivalent approaches, first from a mathematical perspective and then from within musical notation.

Those comfortable with math will appreciate that the semitone elements of the *Pitch* vector can be identified easily using modular arithmetic, allowing a pitch in the 'root octave' to be identified.

> The root octave is that range of pitches corresponding to the original range of CV outputs from 0 to 1V, arising from a single slider.

This is done by taking a pitch and converting to root octave pitch using the expression:

$$rootPitch = \mathrm{mod}(Pitch,12) \qquad (21)$$

where y = mod(x,n) indicates modulo division: y = x - floor(x/n)

By this method, the slider settings of the octave, fifth, min third and additional octave in pitch semitone values of 12, 7, 3 and 12 correspond to "root pitch" values of 0, 7, 3 and 0. More importantly, for those discrete pitches formed by multiple gates generating pitches outside the root octave, the associated pitches inside the root octave are, in the example of equation 20 are:

$$rootPitch = \mathrm{mod}(Pitch,12) = [0,\ 0,\ 7,\ 3,\ 0,\ 7,\ 3,\ 10,\ 10,\ 7,\ 3,\ 10] \qquad (22)$$

It is seen that <u>all</u> the root pitches are either the root, the minor third, the fifth or the minor seventh; the sequence is an arpeggiated minor seventh chord.

Further, the root pitches [0:11] might be understood as musical notes; those pitches named by the letters A, B, C, D, E, F & G and modified by accidentals ♯ and ♭ to give a set of notes which name a pitch sequence which repeats in modulo 12.

Those not so comfortable with math might prefer simply to see the original *Pitch* vector mapped out in musical notation. Supposing the oscillator to be tuned such that the note produced in response to 0V CV is bottom C, the notes in *Pitch* (eqn 21) are these:

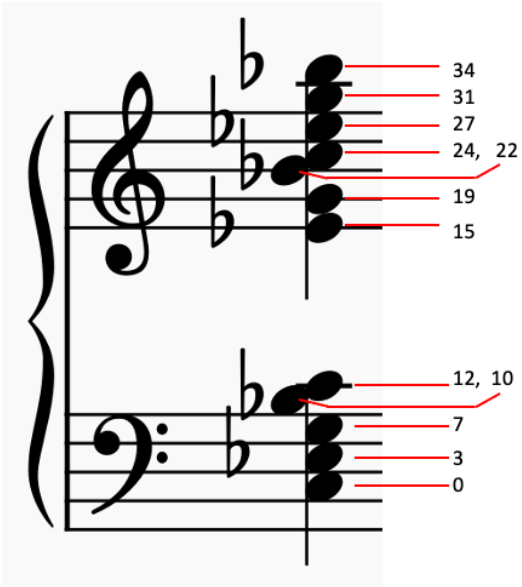


Figure 10 Discrete notes in the sequence resulting from *sliders* = [12,0,7,0,3,0,0,12] (shown with their semitone interval above the root note)

It is seen that the pitches sound on notes form an Cm7 chord, ranging across three octaves above the original 'root octave' in which the original slider notes were defined (C3 to C4 in the notation above).

> Of course, a VCO is not usually tuned to give C3 in response to a 0V input, but we are simply transposing (/ retuning the VCO / adding an offset) for the convenience of the familiar notation in the graphic of Figure 10. It is all the same!

It is convenient to 'collapse' all the pitches back into pitches within this root octave (by shifting down the appropriate number of octaves), which will reduce the set of notes to a clear Cm7 chord (as in eqn. 22):

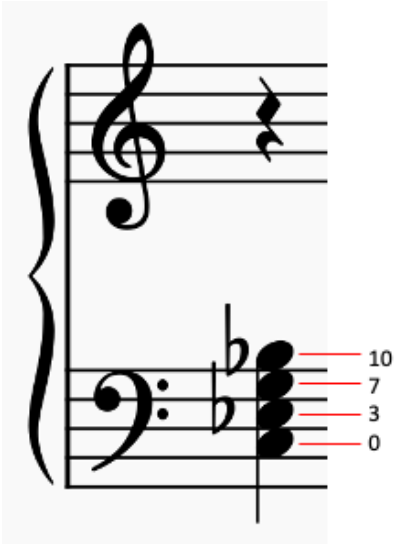


Figure 11 Discrete pitches in the sequence resulting from *sliders* = [12,0,7,0,3,0,0,12], collapsed into the 'Root Octave' (shown with their semitone interval above the root note)

Notice in both the *rootPitch* vector (eqn. 22) and in Figure 11 that the octave (12 semitones above the root note) is considered part of the octave above the root octave and so this pitch is mirrored down to the root note. We'll see why now when we look at pitch constellations.

Looking at the set of pitches used in a sequence listed only in the 'root octave', whether derived by mathematical or musical methods, is conveniently achieved using a visualisation method called a 'pitch constellation' [9].

A pitch constellation is a graphical representation of a set of pitches over an octave interval. It can be used to describe scales, intervals, chords, etc.. We shall use it to allow us to 'see' the notes in our root octave.

The pitch constellation consists of a circle, equally divided along the circumference to indicate the twelve pitches in an octave and uses radial lines from the centre to the circle to indicate each pitch which is present. The pitches can be labelled in semitones, intervals or named notes, as in the examples of Figure 12 below, which show the pitch constellation for the discrete pitches in the m7 sequence resulting from *sliders* = [12,0,7,0,3,0,0,12], collapsed into the 'Root Octave' (i.e. the data of Figure 11).

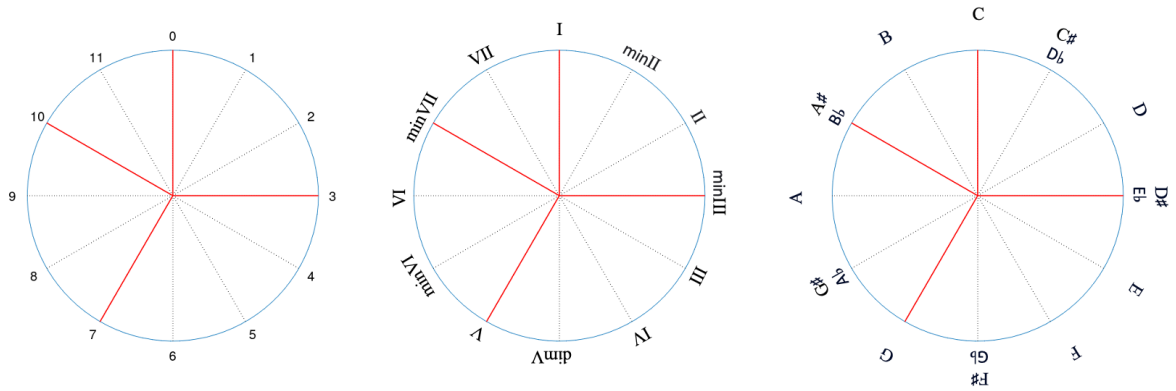


Figure 12 Pitch Constellation representations of a minor 7 chord, with three alternative pitch overlays

The right-most pitch constellation of Figure 12 assumes that the oscillator has been tuned such that the pitch sequence is sounding the minor 7th chord on C. Obviously, the sequence can be 'transposed' to any other chord by adding a fixed offset to the CV sequence coming from the "VOLTAGES" expander.

Now, armed with our method of reducing the Pitch vector to the root octave and the Pitch Constellation visualisation method, let's continue to develop a feel for the tuning strategy.

9 Small Changes

You might still have the same setting on your VOLTAGES sliders which gave the arpeggiated minor 7th sequence that has served as our example back in equation (8):

$$sliders = [12, 0, 7, 0, 3, 0, 0, 0] \qquad (23)$$

If you don't, set it up now as the starting point for more experiments. We're going to make small changes and see that they make a significant difference.

First, change the slider that's on '3' (the minor third) to '4' (a major third above the note sounded by a slider set to "off"). Remember that you need to change the Turing Machine back to a one-bit repeating sequence to do this easily.

You should now have:

$$sliders = [12, 0, 7, 0, 4, 0, 0, 0] \qquad (24)$$

Now release the Turing Machine to a more random sequence and you will hear a sequence which arpeggiates a major 7th chord.

The notes in your sequence should be associated with a vector:

$$Pitch = [\ 0 \quad 4 \quad 7 \quad 11 \quad 12 \quad 16 \quad 19 \quad 23] \qquad (25)$$

and the associated pitch constellation is shown as Figure 13

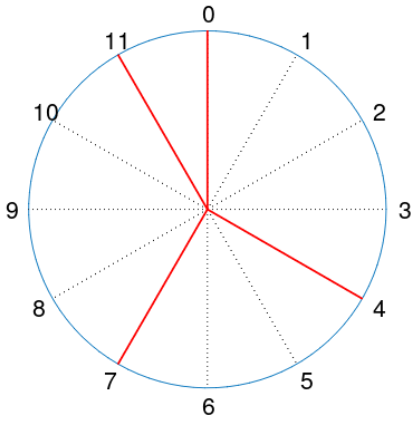


Figure 13 Pitch Constellation of a major 7 chord, resulting from the sequence generated by *sliders* = [12, 0, 7, 0, 4, 0, 0, 0]

Changing another slider from '0' to '12' will extend the pitch span of this major 7th sequence by another octave, as before.

You can see how easily the min7 sequence has been transformed into a maj7 sequence. However, be warned. It isn't usually this easy!

Next, take your third slider (which should be set to '7') and move it down to '6' (the diminished fifth above the note sounded by a slider set to "off"). This should leave you with:

$$sliders = [12, 0, 6, 0, 4, 0, 0, 0] \qquad (26)$$

Now release the Turing Machine and you'll be surprised by the change in the character of the sequence. You move from the calm maj7 to something open and insecure.

This new sequence has a *Pitch* vector:

$$Pitch = [0 \quad 4 \quad 6 \quad 10 \quad 12 \quad 16 \quad 18 \quad 22] \qquad (27)$$

It has quite a lot of whole-tone (*i.e.* two semi-tone) intervals, which gives it a mysterious sound; it's root pitch vector is part of a "whole-tone scale". In fact, you can make the sequence cover all elements of a whole-tone scale, by dropping the first slider from '12' to '10' (the minor seventh above the note sounded by a slider set to "off"), leaving you with:

$$sliders = [10, 0, 6, 0, 4, 0, 0, 0] \qquad (28)$$

This third new sequence has a *Pitch* vector:

$$Pitch = [0 \quad 4 \quad 6 \quad 10 \quad 14 \quad 16 \quad 20] \qquad (29)$$

and the pitch constellation, Figure 14, identifies the sequence as being the complete whole-tone scale. Again, the pitch span can be extended by moving any 'unused' slider from '0' to '12'.

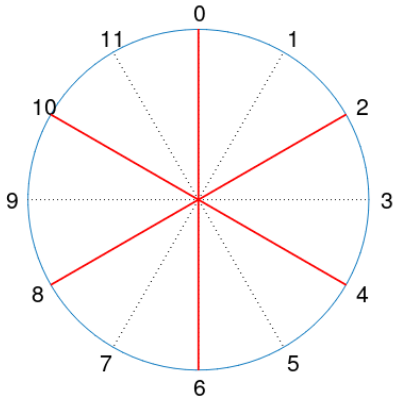


Figure 14 Pitch Constellation of the Whole-tone scale resulting from the sequence generated by *sliders* = [10, 0, 6, 0, 4, 0, 0, 0]

I could have suggested moving slider1 from '12' to '2' (instead of '10') to give:

sliders = [2, 0, 6, 0, 4, 0, 0, 0]

This would also have generated a whole-tone sequence AND the second is a lot easier to tune than the minor seventh (see Appendix A). However, the 'distance' between vectors

[12, 0, 6, 0, 4, 0, 0, 0] and [10, 0, 6, 0, 4, 0, 0, 0]

is smaller than the distance between

[12, 0, 6, 0, 4, 0, 0, 0] and [2, 0, 6, 0, 4, 0, 0, 0]

and this section is all about *small* changes!

You have seen that in three small moves we have gone from a min7 sequence to:

i) a maj7 sequence,
ii) a "sparse" sequence, and
iii) a whole-tone scale.

All used only three active sliders, and all could be extended in pitch span by simply moving additional 'unused' sliders from '0' to '12'. You can imagine the complexity available on just these three sliders by choosing other settings – but there are five more to play with!

Before we dare bring more sliders into play, we ought to formalise the logic we introduced in section 7, which explains how pitches are created by combinations of contributions from multiple sliders. We'll do that slowly (with most of the tougher math relegated to an Appendix) and on the way we'll look at how moving an 'unused slider' from '0' to '12' is working.

10 Octave Expansions, Frequency Shift and Convolution

You have seen several times how taking a tuning of the sliders and moving one 'unused' slider (meaning a slider set to its left-hand limit, giving it a value in semitones of '0') to a setting of '12' will extend the pitch span of the sequence.

We met this in sections 5 and 7 using the tuning:

$$sliders = [12, 0, 7, 0, 3, 0, 0, 0] \qquad (30)$$

(eqn. 8) which gave rise to the minor 7th sequence (eqn. 13).

What I did not point out in those earlier sections is that this tuning already has been subject to that pitch span expansion; we can move slider 1 down from the '12' setting to '0' and the resulting tuning:

$$sliders = [0, 0, 7, 0, 3, 0, 0, 0] \qquad (31)$$

will still give rise to a minor 7th sequence – although this one is comprised only of the Pitch vector:

$$Pitch = [\,0 \quad 3 \quad 7 \quad 10] \qquad (32)$$

i.e. it occupies only one octave – the 'root octave' (compare to the result in eqn. 13).

Accordingly, for this case (where there is no 'translation' required from higher octaves):

$$rootPitch = Pitch \qquad (33)$$

Now, if we look at the "probability density function" of the sequence, we see the result shown in Figure 15.

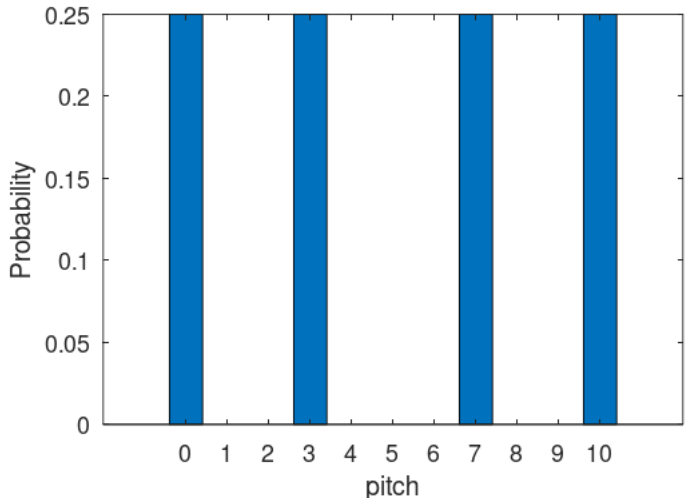


Figure 15 Probability Density of pitches in the sequence generated by *sliders* = [0, 0, 7, 0, 3, 0, 0, 0]

Notice that all four elements of the Pitch vector are equally likely to occur (*i.e.* they have equal probabilities). This refers not to the probability in any one fixed gate sequence coming from the Turing Machine, but to ALL possible fixed gate sequences or – equivalently – to the response to a random sequence (*i.e.* the Turing Machine main control in the 12 o'clock position). The shape of the probability density of the minor 7th sequence in Figure 15 is a fixed property of the tuning:

sliders = [0, 0, 7, 0, 3, 0, 0, 0]

or of any other tuning comprised of a 7, a 3, and six zeros, such as:

sliders = [7, 0, 3, 0, 0, 0, 0, 0],
sliders = [7, 3, 0, 0, 0, 0, 0, 0],
sliders = [0, 3, 0, 0, 0, 0, 7, 0],

etc..

Think of it as the 'signature' or 'fingerprint' of the tuning.

Now, if we add back the '12' to the first slider to go back to the tuning we were using in sections 5 & 7, we have seen the *Pitch* vector and we know it spans over two octaves.

In fact, you've already seen the probability distribution of the *Pitch* vector (Figure 7), but I shall repeat it here as Figure 16, with an additional feature:

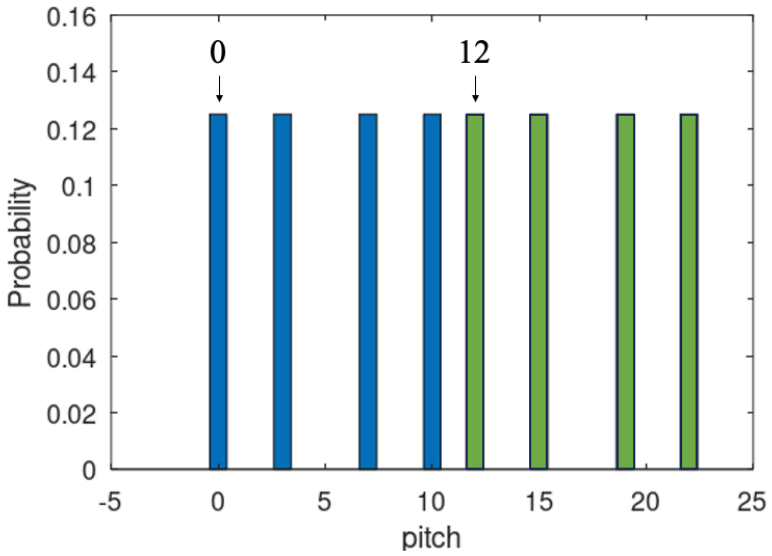


Figure 16 Probability Density of pitches in the sequence generated by *sliders* = [12, 0, 7, 0, 3, 0, 0, 0]

Figure 16 shows the probability density has now halved in 'height', relative to Figure 15, indicating that any individual pitch is now half as likely to occur at any point within the sequence, as there are twice as many pitches.

Figure 16 also suggests that the probability is made of two instances of the 'characteristic' probability density (Figure 15) of the simpler sequence spanning only one octave. One is in the original 'root octave' (seen in blue in Figure 16) and the other (shown in green) is shifted up to the next octave. Let's see if this chance or the basis of a general rule.

To do this, we'll briefly re-tune the whole 'main interval' of section 5, such that we can set TWO OCTAVES on each slider.

To re-tune, first set the sliders so that seven are fully "off" (*i.e.* on their left hand stop) and one is fully "on" (*i.e.* on the right hand stop). Then set up a one-bit repeating pattern on the Turing Machine and adjust the SCALE control so that there are two octaves between the note sounded by your VCO on the active slider and all the unused sliders. This corresponds to a 2-volt CV output from VOLTAGES on the active step.

Now you've calibrated the sliders such that they run from 0:24 in semitones.

Now, set up the same tuning of:

$$sliders = [0, 0, 7, 0, 3, 0, 0, 0] \qquad (34)$$

It is surprisingly easy to do, especially if you leave slider 1 'on' and use it as a pitch reference, so you start with:

$$sliders = [24, 0, 7, 0, 3, 0, 0, 0]$$

Here (in Figure 17) is the tuning:

$$sliders = [0, 0, 7, 0, 3, 0, 0, 0]$$

set up on my VOLTAGES expander when the main interval is two octaves:

Figure 17 Setting a Fifth and a minor Third (with each slider having a two-octave span)

Now, set the first slider to full on (‘24’), such that:

$$sliders = [24, 0, 7, 0, 3, 0, 0, 0] \qquad (35)$$

and then release the Turing Machine to make a random gate sequence. You will hear something different to the patterns we’ve explored to this point. Rather than being uniformly distributed over pitch, there are now two distinct ‘clusters’ of pitch in the sequence; a low arpeggio interleaved with a higher arpeggio of the same chord (if your tuning has been accurate).

The pitch is distributed according to Figure 18

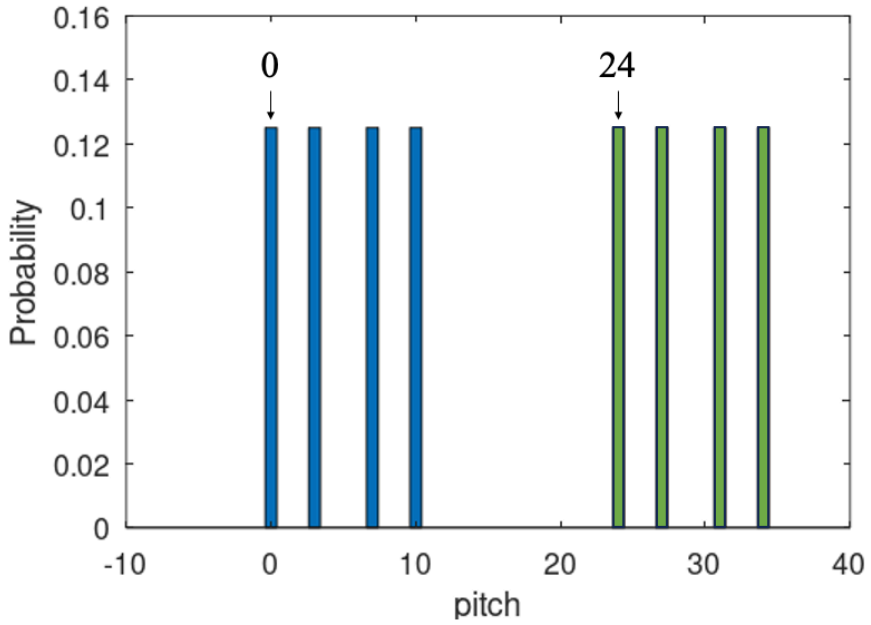


Figure 18 Probability Density of pitches in the sequence generated by *sliders* = [24, 0, 7, 0, 3, 0, 0, 0]

Figure 18 shows again that the probability is made of two instances of the 'characteristic' probability density of the simple sequence of Figure 15. One is in the original 'root octave' (seen in blue) and the other (shown in green) is shifted up two octaves. There is now a clear gap between the two sequences and – if you lower slider 1 from its '24' full-scale value – you can detune the upper min7 sequence independently of the lower min7. In fact, you can modulate the upper minor 7th sequence continuously (including at micro-tonal intervals).

This strongly reinforces the idea that adding a slider at value 'n' to the basic 'minor 7th' tuning:

$$sliders = [n, 0, 7, 0, 3, 0, 0, 0] \qquad (36)$$

adds an instance of the characteristic pattern of the probability density seen in Figure 15, but **shifted up to pitch *n***. We shall look at two more examples to further test this conjecture before accepting it as generally true.

To do this, we'll return to the familiar territory of a 'main interval' tuning of 12 semitones (see section 5) and return to the tuning:

$$sliders = [12, 0, 7, 0, 3, 0, 0, 0]$$

We know (from Figure 18) what happens when we add a '24' to the basic minor 7th tuning in one step. We shall now take another look at moving a second slider from '0' to '12' to give e.g.:

$$sliders = [12, 12, 7, 0, 3, 0, 0, 0] \qquad (37)$$

We've seen the probability distribution before (Figure 8) but now we're ready to understand more features in Figure 19.

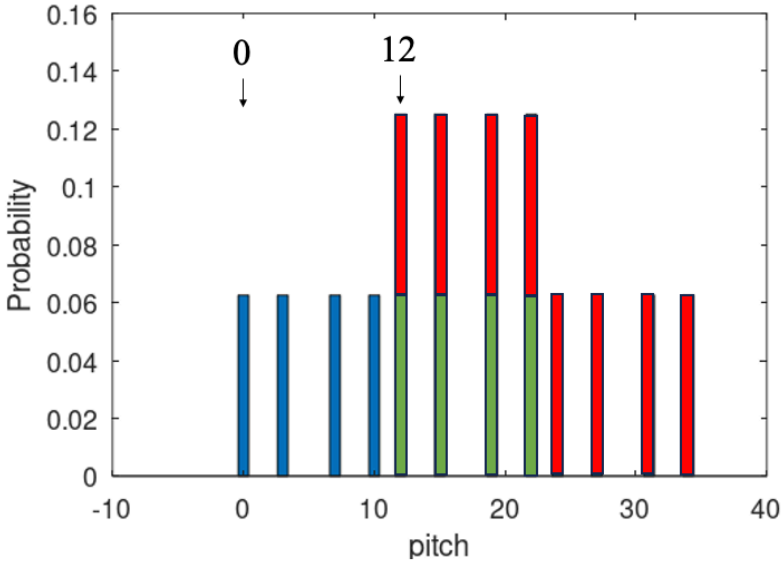


Figure 19 Probability Density of pitches in the sequence generated by *sliders* = [12, 12, 7, 0, 3, 0, 0, 0]

The probability density distribution in Figure 19 is generated in three stages.

First, there is the basic statement of the characteristic distribution of the minor 7th sequence associated with:

sliders = [0, 0, 7, 0, 3, 0, 0, 0] (38)

This is shown in blue in Figure 19.

Next, there is the repeated statement of the characteristic distribution of the minor 7th sequence, shifted up to pitch = 12 by one of the sliders which is set to 12.

This is shown in green in Figure 19.

Finally, there is the repeated statement of the combination of the elements above (*i.e.* the blue and green pitches), shifted up to pitch = 12 by the second of the sliders which are set to 12.

This is shown in red in Figure 19.

Note that where these overlap, the probabilities add, such that pitches in the central octave are more likely than in the extremes – this is easy enough to hear if you set up VOLTAGES to play the sequence.

Also notice that – because of the careful choice of all the slider settings – the composite sequence retains the overall minor 7th structure used in the slider original settings. It is the intelligent choice of slider settings that make tunings for VOLTAGES which preserve 12TET musicality and integrity over a wide pitch span. For a final example, we shall look at how easily this can be exploited, to make sequences which sound more challenging.

Go back to the basic minor 7th tuning we've been using for most of these examples:

$$sliders = [0, 0, 7, 0, 3, 0, 0, 0]$$

Next move one of the inactive sliders up a minor second (a semitone) to setting '1':

$$sliders = [0, 0, 7, 0, 3, 0, 0, 1] \qquad (39)$$

The resulting sequence has a probability distribution shown in Figure 20.

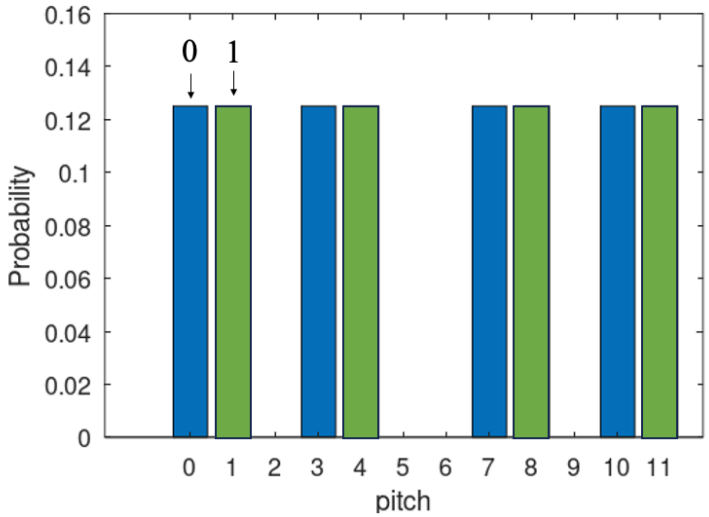


Figure 20 Probability Density of pitches in the sequence generated by *sliders* = [0, 0, 7, 0, 3, 0, 0, 1]

The probability density distribution in Figure 20 is generated in two stages.

First, there is the basic statement of the characteristic distribution of the minor 7th sequence associated with:

$$sliders = [0, 0, 7, 0, 3, 0, 0, 0]$$

familiar from Figure 15. This is shown in blue in Figure 20.

Next, there is the repeated statement of this characteristic distribution of the minor 7th sequence, shifted up to pitch = 1 by the slider which is set to 1. This is shown in green in Figure 20.

Notice that the blue and green elements of Figure 20 coexist in the root octave. They add up to give the sequence pitches from an octatonic scale which is much more complex than the simple generating minor 7th chords which created it. The sequence is still 12TET, but the juxtaposition of the m7 sequence in the root octave and the same 'chord' starting on the flattened 2nd is 'interesting' to say the least (actually, I quite like it!)

These examples above are sufficient to prove the idea that the set of pitches in a sequence produced by VOLTAGES is built up by an iterative process, in which the action of each additional slider can be seen as introducing a copy of the pitches produced by the previous slider settings shifted in pitch to the additional slider's setting. Each additional slider generates a modulated copy of what was there before, where the modulation is controlled by the setting of the slider. There is only an "upper sideband" generated by this modulation process (*i.e.* the pitches only add) and there is no phase sensitivity.

Mathematically, this is equivalent to a convolution [10] process, in which the *Pitch* vectors generated by sub-sets of the *sliders* vector are convolved with simple functions associated with additional elements of the *sliders* vector. We'll look at the math behind this convolution next.

11 Calculating Sequence Pitches using Convolution

In this section we shall use convolution to identify the pitches in the sequences produced by a VOLTAGES expander tuned to semitone intervals. It follows from the proposition we saw demonstrated by example in section 10; that each 'new' slider introduced to an existing tuning adds to the sequence a copy of the set of pitches produced by the existing active sliders but shifted up in frequency by the pitch of the new slider.

We shall seek to express the idea in conventional mathematical notation and give an implementation in Python.

Given the vector of slider settings used previously, *sliders*, we shall construct from each of the elements of *sliders* a 'unit sample function':

$$\delta[pitch - sliders_i] \qquad (40)$$

The unit sample function is a special class of Kronecker delta function [11], which is zero valued everywhere except when the argument is zero, when the function has value one. In this case, the argument is zero value at:

$$pitch = sliders_i \qquad (41)$$

In other words, we generate functions of pitch which are zero valued, but have unit "spikes" at the values of the slider settings.

Let's look again at the first example we used in section 5 for a slider tuning from which to generate a set of these unit sample functions. In this familiar tuning,

$$sliders = [12, 0, 7, 0, 3, 0, 0, 0] \qquad (42)$$

the first slider has value: $sliders_1 = 12$, so the first of the 'sample functions' is zero at all pitches except 12, where it is unit valued.

$$\delta_1 = \delta[pitch - 12] \qquad (43)$$

The second slider has value: $sliders_2 = 0$, so the first of the 'sample functions' is zero at all pitches except 0, where it is unit valued. This is also true of samples 4, 6, 7 & 8, so (in this case):

$$\delta_2 = \delta[pitch - 0] = \delta[pitch] = \delta_4 = \delta_6 = \delta_7 = \delta_8 \qquad (44)$$

The third slider has value: $sliders_3 = 7$, so the third of the 'sample functions' is zero at all pitches except 7, where it is unit-valued and the fifth slider has value: $sliders_5 = 3$, so the fifth of the 'sample functions' is zero at all pitches except 3, where it is unit-valued. The sample functions for our example tuning (eqn. 42) are shown in Figure 21 for all values of *i*.

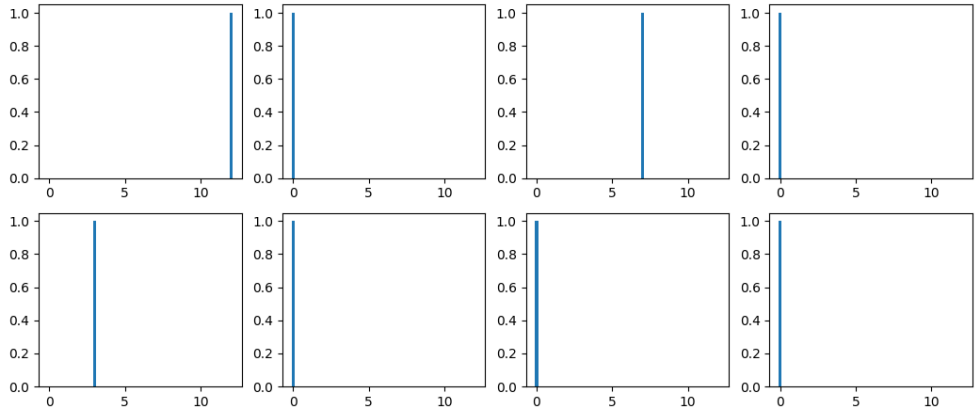


Figure 21 Sample Functions, $\delta[pitch - sliders_i]$, for the tuning *sliders* = [12, 0, 7, 0, 3, 0, 0, 0]

We shall also define a function which counts the occurrence of an individual pitch within the range of pitches available for sequence generation (as we already have noted in Section 5 that a slider tuning can give rise to repeat instances of a pitch through different combinations of slider contributions). We shall call this function '*cpitch*'.

> The function *cpitch* does not directly identify a pitch within VOLTAGES' output sequence. Rather the pitches implied by non-zero counts in *cpitch* are identified by the indices at which the non-zero counts occur. So, for example, a count of 1 at $cpitch_0$ indicates one occurrence of a pitch at semitone 0 and a count of 6 at $cpitch_7$ indicates 6 occurrences of a pitch at semitone 7.

Our rule for counting occurrences of the pitches produced by a slider tuning shall start by assigning a first value to *cpitch* to be a single count at pitch zero, after which an iterative procedure will add extra terms to *cpitch* as defined below:

$$cpitch_0 = [1] \qquad (45)$$

$$cpitch_j = cpitch_{j-1} + conv(cpitch_{j-1}, \delta[pitch - sliders_j]) \qquad (46)$$

where *conv*(a,b) is the convolution of functions a and b.

This means that, for example,

$$cpitch_1 = cpitch_0 + conv(cpitch_0, \delta[pitch - sliders_1]) \qquad (47)$$

and

$$cpitch_2 = cpitch_1 + conv(cpitch_1, \delta[pitch - sliders_2]) \qquad (48)$$

If we substitute for $cpitch_1$ (from eqn. 47) into equation 48:

$$cpitch_2 = cpitch_0 \\ +conv(cpitch_0 \delta[pitch - sliders_1]) \\ +conv(cpitch_1, \delta[pitch - sliders_2])$$

you can see the beginnings of a summation, which allows us to write the final solution for *cpitch* (= $cpitch_8$) in a compact form:

$$cpitch = cpitch_0 + \sum_{j=1}^{8} conv\left(cpitch_{j-1} \delta\left[pitch - sliders_j\right]\right) \quad (49)$$

where the summation operates over the sliders.

As an example, Figures 22 show the evolution of *cpitch* as this summation occurs for the eight slider values within our familiar tuning example:

sliders = [12, 0, 7, 0, 3, 0, 0, 0]

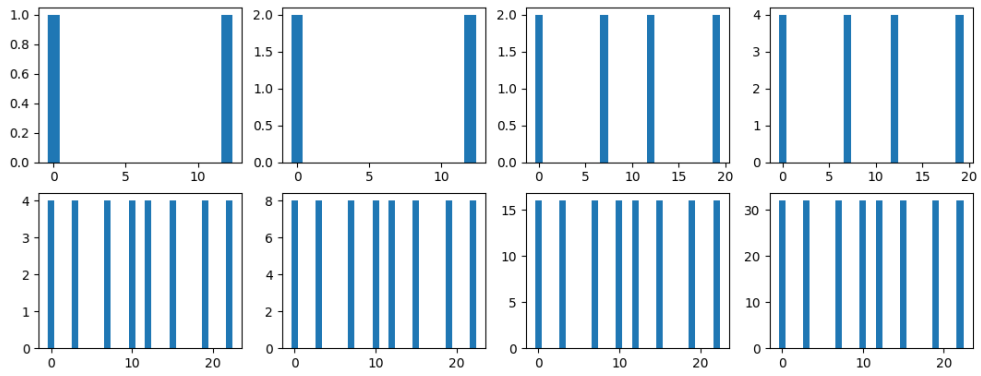


Figure 22 Evolution of $cpitch_j$ for the tuning *sliders* = [12, 0, 7, 0, 3, 0, 0, 0]

Notice from Figures 22 that the patterns of distinct tones only change each time a new, "active" slider is introduced (*i.e.* on the first, third and fifth values of *k*). However, the counts (seen on the "y axis" of the graphs) change even on the zero-valued sliders when tones are repeated.

The final solution for *cpitch* in the case of this example is $cpitch_8$ at the bottom right of Figure 22, when the summation is complete.

Changing from the count of pitch occurrences, *cpitch*, to the probabilities reported previously (e.g. in Figures 8 & 9) is achieved by dividing *cpitch* by the total number of pitch counts reported in *cpitch*:

$$P(pitch) = \frac{cpitch}{sum(cpitch)} \quad (50)$$

This has the effect of scaling *cpitch* - it does not change the shape of the distribution. For the case reported in Figure 22, the probability is shown in Figure 23 (which is exactly as seen before in Figures 7 and 16):

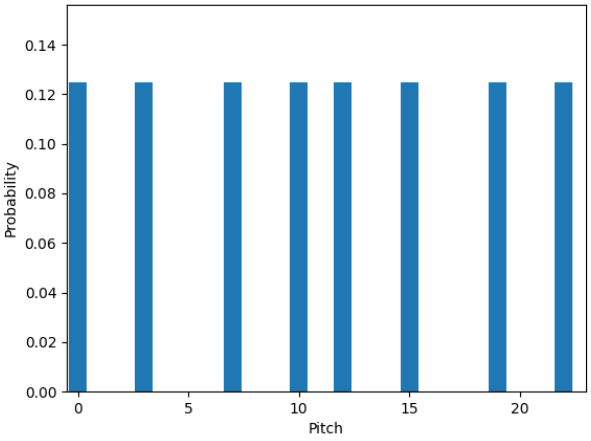


Figure 23 Probability of pitches in the sequence resulting from *sliders* = [12, 0, 7, 0, 3, 0, 0, 0]

We have already noted that *cpitch* does not directly identify pitches – rather it identifies pitches present in a sequence indirectly through the presence of non-zero counts. Accordingly, to find all the pitches 'encoded' within *cpitch* it is necessary to find the indices of all the non-zero elements of *cpitch*. This is a problem most easily written in the programming language used to implement the task, rather than mathematical notation.

For example, in Python [12], the function `numpy.nozero(a)` is a function in the Numerical Computing library "NumPy" [13] which return the indices of the elements of a that are non-zero. Using this function, a Pitch vector can be constructed in Python to identify the pitch instances in *cpitch* as:

```
Pitch = np.array(np.nonzero(cpitches))
```
(51)

(where 'np' is the 'alias' of numpy). A complete Python function listing to solve for the pitch count for any semitone slider setting is shown below to emphasise the simplicity of the task:

```
def findpitches(sliders):
    lsliders = len(sliders)
    cpitches = np.array([1])
    addone = np.array([1])
    for j in range(0, lsliders):
        if sliders[j] > 0:
            y = np.zeros(sliders[j])
            delta = np.append(y, addone, axis=0)
        else:
            delta = addone
        cpitchesadd = np.convolve(cpitches, delta)
        cpitches = [sum(n) for n in zip_longest(cpitches, cpitchesadd, fillvalue=0)]
    return cpitches
```

The function `findpitches( )` has been written for clarity rather than efficiency. It uses the "NumPy" and "itertools" libraries. A few words of explanation are justified, to explain the function in terms of the words and mathematics which have already been presented and which it implements.

The function starts by getting the length of the `sliders` vector (usually 8 in the case of the VOLTAGES module), which it saves as `lsliders`. Two vectors, `cpitches` and `addone` are initialised to [1]. Then, in a loop over values of j which indexes all the sliders, the 'unit sample functions' of equation (40) are created – they are called `delta` in the function. These `delta`s are then convolved with the current value of `cpitches`, using the Numpy function '`convolve()`', to create the additional component `cpitchesadd`, which summed (element-wise) with the current `cpitches`.

The significance of `convolve()` in terms of equation (49) is obvious to see. Less obvious is the summation of equation (49), which is realised by the element-wise addition of `cpitchesadd` and `cpitches`, in the 'for loop' over j.

A full set of Python resources to implement and use the 'convolution' method described in this section are described in Appendix C.

Although the 'convolution method' above was described in the context of semitone (*i.e.* integer) settings of VOLTAGES' sliders, it can be extended to reflect the fact that the sliders are continuous controls, able to be set to any position within their range of travel. The 'convolution' approach can still be applied in this case, but it must be re-formulated and becomes clumsier than the simple integer/semitone tuning case. This re-formulation is not important to our story, so it is treated in Appendix D.

Traditional approaches to solving for the sequences produced by KREE-type sequencers depend upon combinatorial strategies for solving the "all possible sums of *n* slider settings" problem we saw in Section 7; these also are not vital to our story, so they have been hidden away in in Appendix E for completeness.

We have mentioned several times that the probability distributions for the pitches produced by slider settings (such as that presented) is not the instantaneous probability for the sequence in any setting of the Turing Machine's main control (Figure 1 (a)). We shall now turn to a brief discussion of the influence of the Turing Machine's gates on selecting which of the possible pitches are selected from those we have identified as possible.

12 Playing all the Right Notes

In 1971, comedian Eric Morecambe described his (intentionally comic) performance of the opening bars of Grieg's piano concerto with the words "*I was playing the right notes, but not necessarily in the right order*". This comment is relevant to our model of the sequences produced by the VOLTAGES expander.

We have a way of describing exactly the set of all the pitches that can be produced when the sliders of the VOLTAGES expander are placed in a given semitone setting (or a more general setting if you venture into the micro-tonal methods of Appendix D). We know the set of all possible notes – we know "all the right notes". But we have no idea what notes will be played at one instant - we do not know "the right order".

Those pitches sounded from within the set of allowed pitches is entirely decided by the Turing Machine (TM) and by the gates it asserts.

As I sit here writing, I have my modular playing, with VOLTAGES set to a major 7th tuning,

$$sliders = [12, 12, 7, 4, 0, 0, 0, 0]$$

which produces (as you can confirm) 12 possible pitches.

As I listened just before typing this sentence I noticed a repeating pattern of 2 illuminated LEDs, separated by a single 'blank' LED, cycling round the TM's shift register:

$$\begin{aligned} gate_k &= [1, 0, 1, 0, 0, 0, 0, 0] \\ gate_{k+1} &= [0, 1, 0, 1, 0, 0, 0, 0] \\ gate_{k+2} &= [0, 0, 1, 0, 1, 0, 0, 0] \\ &\cdots \\ &\cdots \\ gate_{k+8} &= [1, 0, 1, 0, 0, 0, 0, 0] \end{aligned} \qquad (52)$$

I counted at least 14 full repeats of the entire pattern (*i.e.* 14*8 clock pulses) before anything changed and certainly not all 12 of the possible pitches were sounded.

Just now, another cycle with the same pattern of two 'active gates' separated by a blank emerged on the TM. I think I counted distinct 5 pitches sounded within that 'sub-sequence'.

The main control of my TM (Figure 1 [a]) is currently set at about '1:30 pm', and the LENGTH control is set to 8, so it will generate and hold some repeating gate sequences of length 8, which it randomly refreshes. This is – after all, the beauty of the Turing Machine.

Occasionally, you will hear your TM drift to a setting where all the gates are active:

$$gate_k = [1, 1, 1, 1, 1, 1, 1, 1] \qquad (53)$$

and, if your main control is anywhere to the right of “12 o’clock” that will persist for some time, with the result that you’ll hear only one note from your sequence, specifically that high pitch given by the sum of all the slider settings.

Similarly, you will occasionally hear your TM drift in the opposite direction to a setting where all the gates are off:

$$gate_k = [0, 0, 0, 0, 0, 0, 0, 0] \qquad (54)$$

The output – of course – is (see equation 5) that the output will be 0V and (see equation 7) that the pitch will be at the pitch on you have tuned your oscillator to sound at in response to 0V input.

One last familiar example is worth mentioning before we make the point. You have become used to setting up the TM with a ‘one-bit’ repeating cycle, which involves setting the main control at “5 o’clock”, and setting the gate pattern:

$$gate_k = [1, 0, 0, 0, 0, 0, 0, 0] \qquad (55)$$

This has the effect (see equations 5 and 7) of producing outputs associated with the individual actions of each of the sliders as they are sequentially activated.

This set of examples, associated with the gate patterns of equations 52:55, highlights how the Turing Machine’s gate outputs will select a sub-set of the available set of pitches associated with a *sliders* tuning. The TM can easily produce gate sequences that produce only one output level (equations 53 & 54) and 8 or less output levels (*e.g.* equations 52 & 55) from the entire set of possible outputs levels of which it is capable.

It is only when we listen over a long enough period, with the TM’s main control close enough to the vertical, that we

i) hear all the members in the set of possible pitches and, then -
ii) start to hear them in their true probability distribution.

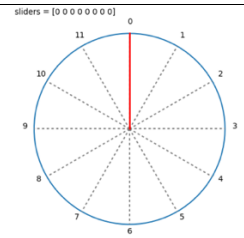
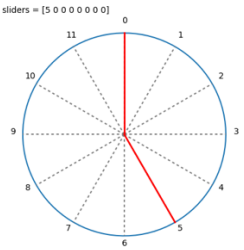
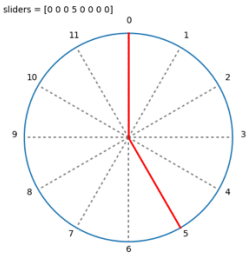
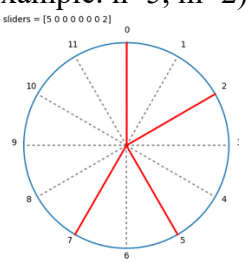
One final comment. A set of allowed notes for a sequence is incomplete without understanding of the probability at which these notes will occur. This is emphasised by the ‘major scale’ tunings (#18 and #19 in the catalogue) which, although they have the notes of the major scale, are distributed in a manner which favours the Dorian mode (which has a minor feel). That’s why so much effort has been expended in the study of probability in these notes!

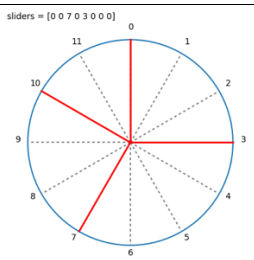
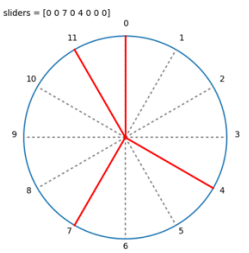
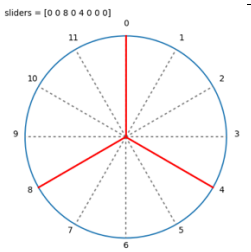
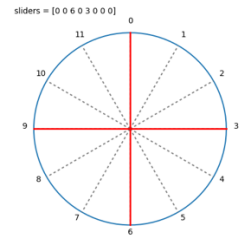
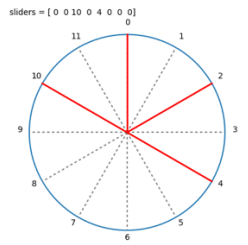
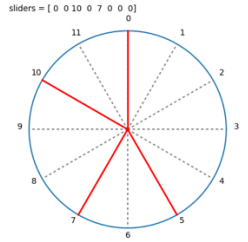
We are now able to turn to the list of practical tunings which emerge from our new tuning strategy.

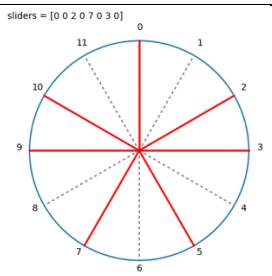
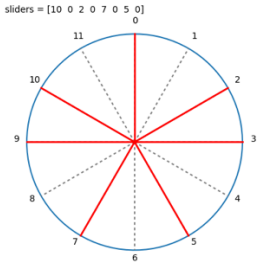
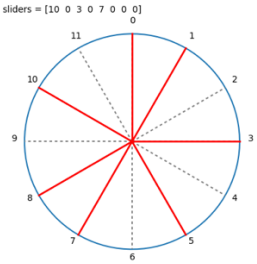
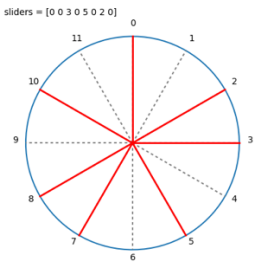
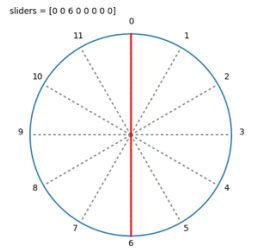
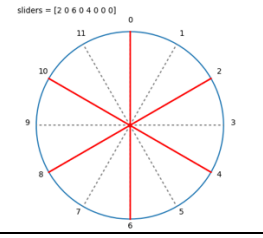
13 A Catalogue of Tunings for the VOLTAGES expander

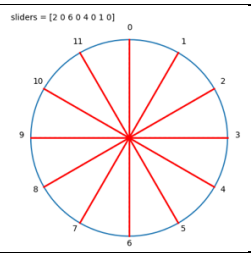
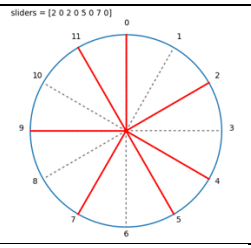
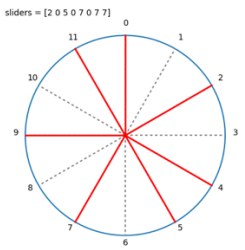
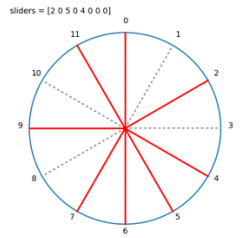
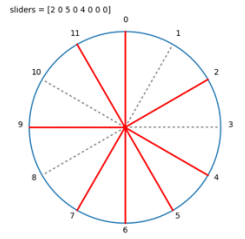
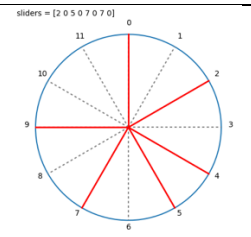
In this section we present a list of tunings for the VOLTAGES expander, using the strategy of placing sliders on semitone intervals. This gives rise to a sequence of 12TET notes occupying a predictable distribution over a defined pitch span and root octave pitch constellation. In each example given the slider setting is given and the pitch span and pitch constellation is reported.

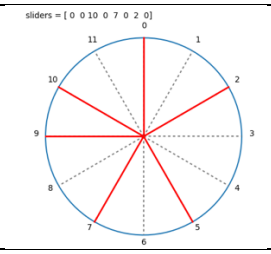
The examples start with some general rules and then proceed to give examples in different categories.

#	**Sliders Tuning** (semitones)	**Description**	**Span** (Octaves)	**Pitch Constellation**
General Rules				
1	0, 0, 0, 0, 0, 0, 0, 0	Sequence comprises one note at semitone 0	0	
2	n, 0, 0, 0, 0, 0, 0, 0	Sequence comprises notes at semitones 0 and n	floor(n/12)	(example: n=5) 
3	0, 0, 0, n, 0, 0, 0, 0 *or* 0, 0, 0, 0, 0, n, 0, 0 *or* 0, 0, 0, 0, 0, 0, 0, n *etc*	Sequence comprises notes at semitones 0 and n	floor(n/12)	(example: n=5) 
4	n, 0, 0, 0, 0, 0, 0, m	Sequences comprises notes at semitones 0 and n plus semitones 0+m and n+m	floor((n+m)/12)	(example: n=5, m=2) 

Tunings for Arpeggiated Chords				
5	0, 0, 7, 0, 3, 0, 0, 0	minor 7th chord	0	
6	0, 0, 7, 0, 4, 0, 0, 0	major 7th chord	0	
7	0, 0, 8, 0, 4, 0, 0, 0	augmented chord	1	
8	0, 0, 6, 0, 3, 0, 0, 0	dim7 chord	0	
Tunings for (incomplete) Arpeggiated Chords				
9	0, 0, 10, 0, 4, 0, 0, 0	"9th feel"	1	
10	0, 0, 10, 0, 7, 0, 0, 0	"sus7 feel"	1	
Tunings for Modes				

11	0, 0, 2, 0, 7, 0, 3, 0	DIORIAN (with uniform probability distribution)	1	
12	10, 0, 2, 0, 7, 0, 5, 0	DORIAN (with non-uniform probability distribution)	2	
13	10, 0, 2, 0, 7, 0, 5, 0	PHRYGIAN	1	
14	0, 0, 2, 0, 3, 0, 5, 0	AEOLIAN (also “Natural Minor”)	0	
see also #18 & #19, IONIAN (‘aliasing’ as Dorian through their Probability Distributions)				
Tunings for arpeggiated Symmetric Scales				
15	0, 0, 2, 0, 3, 0, 5, 0	Tritone	0	
16	2, 0, 6, 0, 4, 0, 0, 0	Whole-tone scale	1	

17	2, 0, 6, 0, 4, 0, 1, 0	Chromatic scale	1	
See also #7 (dim7 Chord) and #8 (Augmented Chord)				
Tunings on *or close to* **the Major Scale**				
18	2, 0, 2, 0, 5, 0, 7, 0	Major Scale	1	
19	2, 0, 5, 0, 7, 0, 7, 7	Major Scale	2	
20	2, 0, 5, 0, 4, 0, 0, 0	Octatonic Scale, comprising Major Scale with added ♭V	0	
21	2, 0, 5, 0, 9, 0, 7, 0	Octatonic Scale, comprising Major Scale with added ♭V	1	
22	2, 0, 5, 0, 7, 0, 7, 0	Hexatonic Scale, close to Major Scale, but missing the VIIth degree	1	
Miscellaneous Tunings				

23	0, 0, 10, 0, 7, 0, 2, 0	Hexatonic Scale with no third. This is a favourite and was *where it all began* for me!	1	

I had been searching for a tuning which led to the major scale for a while and was pleased to find these (catalogue #18 & #19), both of which were discovered during the production of these notes. I found #19 first, but #18 appears higher in the list as it is simpler and has smaller span.

The probability distribution of both 'major' tunings is far from flat. In the case of #18, the distribution favours the II, V and VI, giving the sequence and feel of Dorian mode. The same is true in the case of #19 (see Figure 24). The resulting sequence has a tonality which leans more to the (Dorian) minor than the (Ionian) major but is no less usable for that.

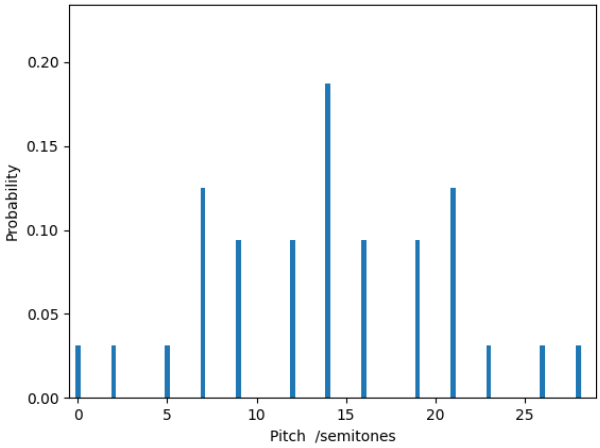


Figure 24 Probability Distribution of the Sequence associated with the "Major Scale tuning" (catalogue #18)

Appendix A *VOLTAGES' Non-Linear Slider Response*

The slider potentiometers on the "VOLTAGES" expander module are driven by the gate voltages and form a potential divider to generate individual slider voltages. These voltages pass currents through 10k resistors (realised as a SIL resistor network in the Expander module) to a summing node at the inverting input of an op-amp. The SCALE control is the feedback resistor of this inverting amplifier.

The relevant part of VOLTAGES' circuit diagram is shown in the upper section of Figure A.1 below, in which only one of the 8 sliders is considered for clarity.

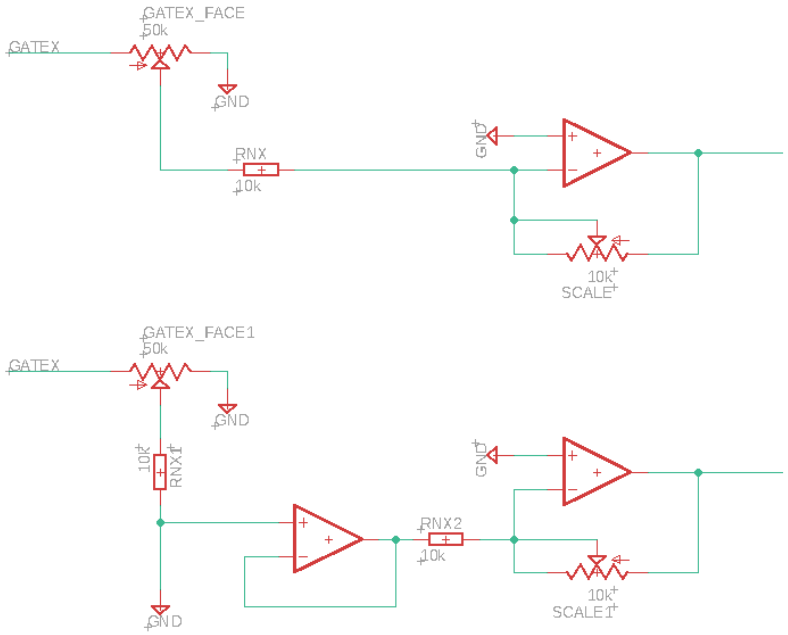


Figure A.1 Circuit around one of the Sliders in "VOLTAGES": Extract from the Schematic (above) and Equivalent Circuit (below)

The sliders each have overall resistance of 50kΩ and have linear taper. The op-amp input resistor(s) have value 10kΩ.

The action of the op-amp in the inverting configuration seen in the upper circuit of Figure A.1 is to maintain its inputs at equal potential. Given that the non-inverting input is tied to ground, the op-amp will adjust its output to hold the voltage at the inverting input at the same voltage. This makes the inverting input a "virtual earth"; it is held at ground potential when the op-amp is working. This will place the input resistor RNX in parallel with the lower part of the potential divider (the part between the slider and ground) and cause loading effects, which will influence operation of the slider as a potential divider.

Recognising the existence of the virtual earth at the op-amp's inverting input, the upper circuit segment in Figure A.1 is equivalent to the lower equivalent circuit, in which the loading effects of the input resistor (now drawn as RNX1) are explicit and an additional ideal buffer stage (not present in the actual circuit) drives the loaded slider voltage through the inverting amplifier to apply the scaling.

We need to understand how the resistor loads the slider to modify the potential divider action.

This is tackled in a further simplification of the circuit, Figure A.2, which exposes only the important elements:

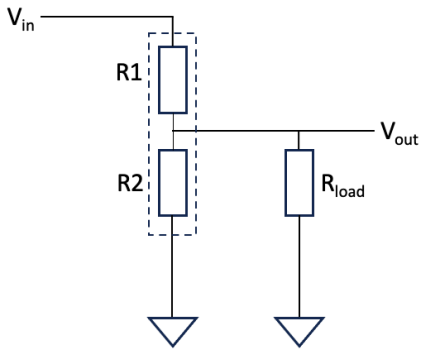


Figure A.2 The Loaded Potential Divider

Figure A.2 shows a network in which a potential divider, formed of two resistors R_1 & R_2, is loaded by a resistor R_{load}. The potential divider represents our slider, and we shall require that the sum of $R_1 + R_2$ always adds up to the total resistance of the slider (50kΩ). Further, we shall use a parameter to represent the 'setting' of the slider – in this case the Greek character α (alpha), which we shall allow to vary between 0 and 1.

$$R_1 = (1-\alpha)50k$$
$$R_2 = \alpha 50k$$

The parallel combination of R_2 and R_{load} is:

$$R_2||R_{load} \stackrel{\text{def}}{=} \frac{1}{\frac{1}{R_2}+\frac{1}{R_{load}}} = \frac{50k}{5+\frac{1}{\alpha}}$$

In this equation (above) the '5' in the denominator comes from the fact that:

$$\frac{R_1+R_2}{R_{load}} \stackrel{\text{def}}{=} r = \frac{50k}{10k} = 5$$

We shall use this ratio 'r' to allow us to extend the analysis to other ratios of slider resistance ($R_1 + R_2$) to load resistance (R_{load}).

Given the parallel combination of R_2 and R_{load}, we can easily write an equation for the loaded potential divider (by using the ordinary potential divider equation [14] between R_1 and this parallel resistance):

$$\frac{V_{out}}{V_{in}} = \frac{\frac{50k}{5+\frac{1}{\alpha}}}{(1-\alpha)50k + \frac{50k}{5+\frac{1}{\alpha}}}$$

which, after a few lines of tedious algebra, becomes:

$$\frac{V_{out}}{V_{in}} = \frac{\alpha}{1+5(\alpha-\alpha^2)}$$

or, in terms of a general ratio between potentiometer and load resistance, *r:*

$$\frac{V_{out}}{V_{in}} = \frac{\alpha}{1+r(\alpha-\alpha^2)} \qquad \text{A.1}$$

Equation A.1 shows that the behaviour of a loaded potentiometer is controlled only by its setting (α) and by the relative load (*r*).

This is a familiar problem, well described in literature. The loaded potential divider is well covered in power applications and solutions where *r* is between 6 and 10 are used to give linear taper potentiometers a more useable transfer characteristic in potential divider applications, approaching a log (/ ”audio”) taper when *r* = 8.33 [15].

Figure A.3 shows the transfer characteristic of the sliders in the Music Thing Modular “VOLTAGES” expander, according to Equation A.1.

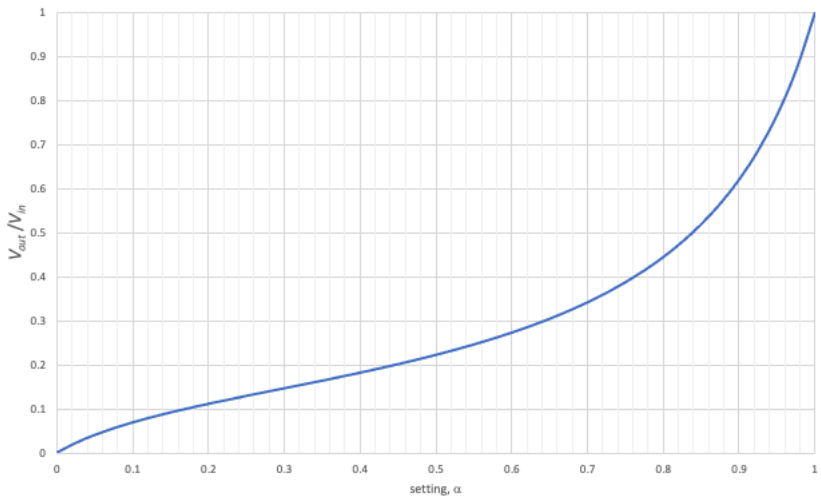


Figure A.3 Transfer Characteristic of VOLTAGES’ sliders

Although the 50kΩ sliders have linear taper, loading (by the 10kΩ resistors) generates the dramatically non-linear characteristic seen in the figure above. This makes it very difficult to set the sliders to discrete semitones in an octave, as we shall now see.

As has already been noted, this is a familiar problem (*although I’ve not yet found equation A.1 written explicitly in this form elsewhere*). More interesting for our purposes – and less familiar

in the literature – is the "inverse problem", where we solve for the setting, α, given a certain target voltage ratio.

Taking equation A.1 and rearranging generates (after some tedious algebra), a quadratic:

$$r^2\alpha^2 + \left(\frac{V_{in}}{V_{out}} - r\right)\alpha - 1 = 0$$

which can be solved with the usual equation:

$$\alpha = \frac{-\left(\frac{V_{in}}{V_{out}} - r\right) \pm \sqrt{\left(\frac{V_{in}}{V_{out}} - r\right)^2 + 4r^2}}{2r^2} \qquad \text{A.2}$$

> Equation A.2 gives two solutions for the potentiometer setting, α. In this case we should take the solution associated with adding the square root of "$b^2 - 4ac$" and neglect the solution associated with the subtraction.

Again, this is familiar enough material, but equation A.2 is original work; I haven't equation A.2 written anywhere but I don't claim any novelty.

Let's assume we have 'calibrated' the potential divider to cover an octave in CV pitch from 0 to 1V (as we did on the VOLTAGES module in section 5). Then we could assume that V_{in} = 1V and that the other notes within the octave are produced by V_{out}. Given A.2 we are now able to establish the settings which will generate all the semitones in the octave, for any loaded potential divider – including that in the "VOLTAGES" module.

The semitones in the octave of VOLTAGES described in a 1V/8^{ve} system by a CV ranging from 0 to 1V are tabulated below (to 4 decimal places). If we plug these VOLTAGES into equation A.2, we can solve for the 'slider settings', α, that correspond to semitones on the VOLTAGES expander if the full-scale slider is tuned to an octave. These α settings are also shown in the table below.

n	V_{out} / V_{in}	α
0	0.0000	0.0000
1	0.0833	0.1307
2	0.1667	0.3583
3	0.2500	0.5583
4	0.3333	0.6899
5	0.4167	0.7773
6	0.5000	0.8385
7	0.5833	0.8835
8	0.6667	0.9179
9	0.7500	0.9450
10	0.8333	0.9669
11	0.9167	0.9849
12	1.0000	1.0000

These data are more informative when overlaid on Figure A.3, as seen in Figure A.4, below:

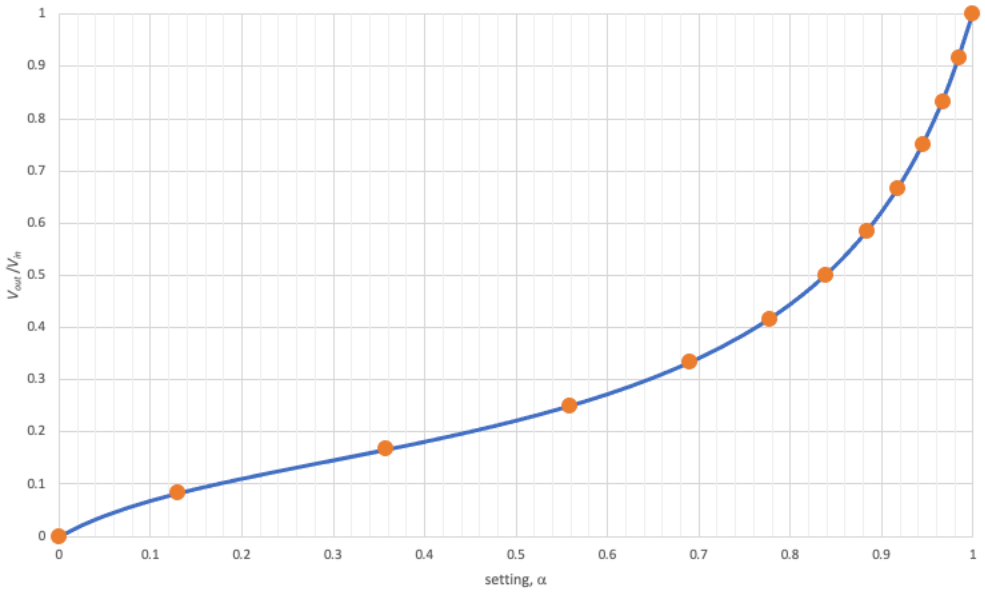


Figure A.4 Transfer Characteristic of the "VOLTAGES" module's sliders, including location of semitone intervals

Notice in Figure A.4 that although (by definition) the semitones are equally spaced in CV voltage, they are not at all evenly spaced over the setting positions on the slider. It is this that makes setting semitones on the "VOLTAGES" expander difficult – particularly the higher semitones within an octave.

The 11th semitone (the major 7th) is only about 1.5% down the slider's track from the end of travel, making it almost impossible to set accurately. Fortunately, it is unlikely that users will want to set the slider on the major 7th, as this will rarely contribute constructively to a useful sequence.

> I originally wrote "won't contribute usefully to a useful sequence" in the previous sentence, but I had to go back and edit it, because I just deliberately set up a tuning with:
>
> *sliders* = [0,0,12,11,0,3,0,0]
>
> and got an interesting 'oriental' sound, which I enjoyed. Never say never!

The 10th semitone (the minor 7th) is only 3.5% down from the slider's end of travel. It too is VERY difficult to find and to tune accurately – and this note is an important part of many useful sequences (see the table in section 13).

This non-linearity associated with loaded potential dividers makes the VOLTAGES module rather difficult to use in the tuning strategy described in these notes and motivates modifications to VOLTAGES to mitigate the loading effects. These modifications are described in Appendix B.

Appendix B *Linearising the Slider Response of "VOLTAGES"*

This appendix describes a simple modification to the VOLTAGES expander module, which minimises the non-linear slider response described in Appendix A, leaving an effectively linear characteristic. The modification requires only the change of two component values.

The potential divider loading of the sliders is caused by the relatively low resistance of the elements of the resistor network RN1, 10kΩ, compared to the resistance of the sliders, 50 kΩ. Any first attempt to reduce the loading effects of the resistors after the sliders might consider increasing the magnitude of the resistor network, perhaps to 100kΩ.

Using the network analysis outlined in Appendix A, the effects of 100kΩ loading on a 50 kΩ potential divider can quickly be evaluated, including identification of the location of thirteen 'semitone' locations along the length of travel of the slider if the entire range corresponds to one octave. Using this approach, an equivalent for Figure A.4 is easily produced, shown as Figure B.1.

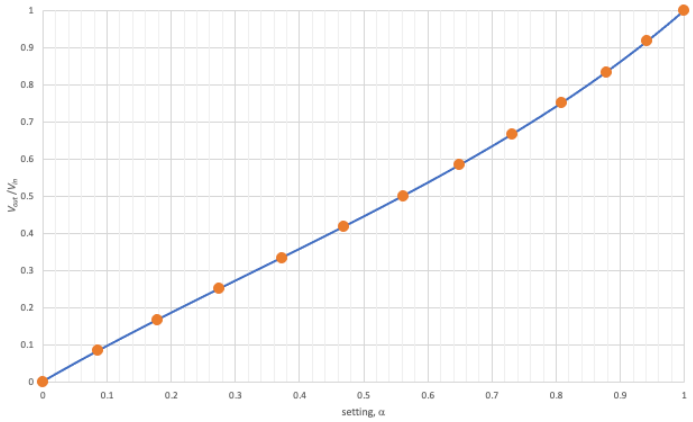


Figure B.1 Transfer Characteristic of 50kΩ sliders, with 100kΩ loading including location of semitone intervals

Figure B.1 shows that the simple expedient of replacing the 10kΩ resistor network in "VOLTAGES" with a 100kΩ replacement would be enough to make a dramatic improvement in the linearity of the slider response, placing semitones at almost perfectly regular spacing along the travel of the slider.

Unfortunately, this change to 10kΩ resistors in the 'RNX' position of Figure A.1 would also give a 20dB reduction in gain of the amplifier stage, which would need to be corrected.

Fortunately, this loss in gain could easily be restored by the equally simple change of replacing the 10kΩ SCALE potentiometer by a 100kΩ part, in which case overall gain is restored to the original design value and the new linearity of Figure B.1 is achieved.

I have tested this modification in a second "VOLTAGES" expander module and can confirm that it works very well.

Figure B.2 shows my original VOLTAGES module and my new, modified VOLTAGES module.

Figure B.2 VOLTAGES modules in [12, 0, 7, 0, 3, 0, 0, 0] tuning

You can see that I deliberately chose to buy a second module kit with a black front panel, so I could easily identify the modified version. Both modules are set in the [12, 0, 7, 0, 3, 0, 0, 0] tuning of Figure 5, but the new module's sliders are in positions that relate more directly to these semitone values - the linearisation works.

Unfortunately, it is not possible to apply this modification as a 'retrofit' to existing modules, as the first three sliders are soldered over the leads of the resistor network, RN1. This makes it impossible to de-solder the existing resistor network and to solder in a new one. It might be possible to unsolder the sliders, but this would be risky.

I decided to fit my resistor network (Bourns part number 4609X-101-104LF) in a 9-way turned pin SIP socket, just in case I should ever want to replace it with the original 10kΩ part. Having done that, I can't think of any reason I'd ever want to go back to the old resistor value, with its loading issues.

Appendix C *Python Resources*

This appendix introduces a set of software resources to support the 'Convolution' solution for integer (semitone) tunings, presented in section 11, extended to cover non-integer (microtonal) settings (the theory for which will be introduced in Appendix D) and placed into context by implementations of the standard ('conventional') approaches to be described in Appendix E.

These resources were written in the Python programming language and are available for download at: https://github.com/m0xpd/TuningStrategyForVoltages/tree/main/Code

The code was developed under the PyCharm IDE on a MacBook running macOS Ventura 13.5.

The code includes:

several functions to solve for the set of all possible sequence pitches, and their distribution, given the slider settings:

Function	Description
`findpitches()`	an implementation of the 'convolution method' taught in section 11
`findpitches2()`	an implementation in which the convolution operator is simplified
`findpitchescomb()`	a conventional combinatorial solution (see Section 7 and Appendix E)
`findpitchesbruteforce()`	a 'Brute Force' approach, which considers all options sequentially (see Appendix E)
`findnonintegerpitches()`	an implementation of the convolution method for non-integer (microtonal) tunings
`findcentpitches()`	an approximate method in which non-integer tunings are quantized to Cents

several Plotting functions to display results (developed using the matplotlib library):

Function	Description
`pitchconstellation()`	a function to display the pitches in a sequence in the root octave
`plotprobability()`	a function to display the probability distribution of the possible notes in a sequence
`plotdeltas()`	a function to display the displaced Unit Sample Functions associated with a tuning (see Fig 21)

and some utilities:

Function	Description
`bitget(value, n)`	a function to return the Boolean state of the n^{th} bit in the binary number 'value'
`cleansliders()`	a function to strip redundant zeros from a 'sliders' vector
`cleanoctaveshifts()`	a function to strip redundant octave shifts from a 'sliders' vector
`countpitches()`	a function to count occurrences of a pitch and build a frequency plot / histogram

The code is scruffy and badly written - but it works, and it is educative; the functions clearly illustrate the workings of the algorithms they implement.

It would run faster if it were re-worked by somebody more skilled in Python (e.g. by removing for loops), but elegance might be accompanied by risk of making the code less accessible and readable to an inexperienced audience.

It could also usefully be organised into a library - but I have neither the time nor the inclination to do this.

Appendix D *Extending the convolution method to microtonal tunings*

The "convolution method" for solving for all possible pitches in a sequence generated by a VOLTAGES slider presented in section 11 was based on the assumption that the sliders were placed at semitone (*i.e.* integer) positions. This appendix describes ways to extend the 'convolution method' to cover micro-tonal (*i.e.* non-integer) slider settings. The description will cover both approximate methods, which use quantised approximations of the pitches, and an exact method, which preserves the pitch. The latter "exact" approach provides a link to other, traditional "combinatorial" strategies for solving for allowed pitches, described in section 7, and covered in mathematical detail in Appendix E.

The easiest way to manage non-integer settings of the sliders is to "quantise" them such that, rather than representing slider position to arbitrary precision, they are allowed to describe only a finite set of allowed positions in the interval between the left and right stops of the slider. This would, for example, allow us to work with "cents", rather than semi-tones. Cents [16] are hundredth parts of a semi-tone, such that an integer slider tuning, *sliders,* is translated into the equivalent tuning in cents, $sliders_{cents}$, simply by the act of multiplication by 100. For example:

$$sliders_{cents} = sliders * 100 = 100 * [12, 0, 7, 0, 3, 0, 0, 0] \qquad (D.1)$$

More generally, any *sliders* tuning, including a non-integer tuning, may be converted to an integer tuning in cents by:

$$sliders_{cents} = \text{round}(sliders * 100) \qquad (D.2)$$

We construct again a set of unit sample functions (cf equation 40):

$$\delta[pitch_{cents} - sliders_{cents,i}] \qquad (D.3)$$

and use a familiar solution for the pitch count (cf equation 49):

$$cpitch_{cents} = cpitch_{cents,0} + \sum_{j=1}^{8} conv\left(cpitch_{cents,j-1}\delta\left[pitch_{cents} - sliders_{cents,j}\right]\right) \qquad (D.4)$$

We must remember that the $cpitch_{cents}$ vector of equation D.3 no longer has counts appearing at indices where the index indicates the pitch in semitones – rather the index indicates the pitch in cents.

As example, Figure D.1 shows the probability distribution of the sequence associated with the tuning *sliders* = [0, 0, 7, 0, 3, 0, 0, 0] (i.e. $sliders_{cents}$ = [0, 0, 700, 0, 300, 0, 0, 0]). This integer (semi-tone) tuning has been seen already (in Fig 15) and is included as confirmation of the validity of the method.

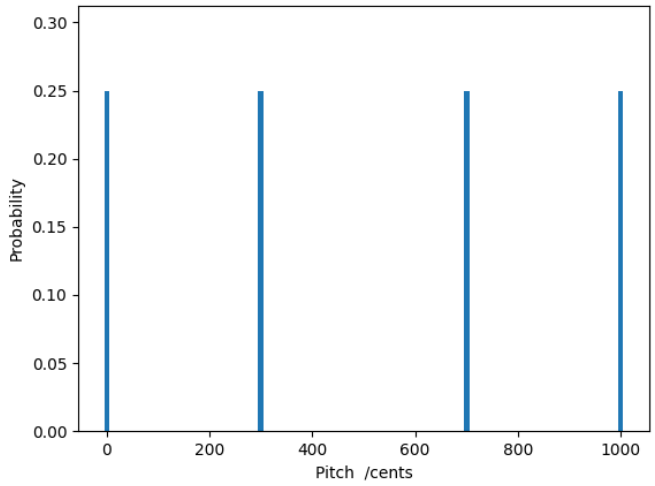


Figure D.1 Probability Density of pitches in the sequence generated by *sliders* = [0, 0, 7, 0, 3, 0, 0, 0]

If we now deliberately move the tuning to non-integer settings, e.g.

sliders = [0, 0, 7.25, 0, 2.666…, 0, 0, 0]

we see a new result, in Figure D2.

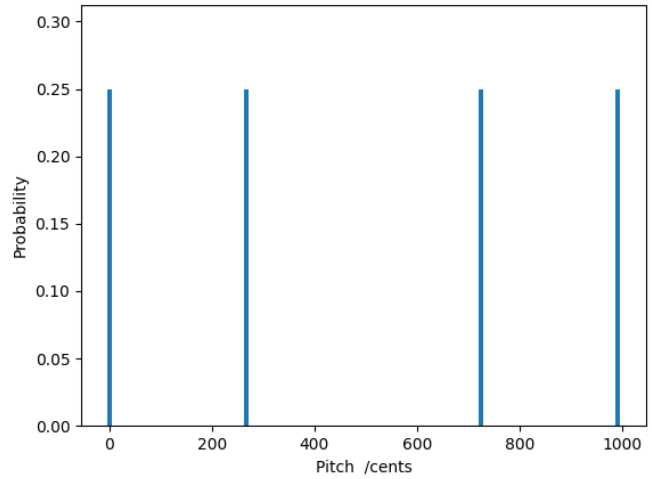


Figure D.2 Probability Density of pitches in the sequence generated by the non-integer tuning:
sliders = [0, 0, 7.25, 0, 2.6666, 0, 0, 0]

Although very similar, it is just possible to see the displacement of the tone near 1000 cents (*i.e.* 10 semitones; the minor 7th), in Figure D.2. The corresponding displacement of the other tones (a sharpening of the fifth by 25 cents and a flattening of the minor third by 33 cents) is less visible in Figure E.2, but it is accurately captured by the process.

The Python function `findcentpitches( )` in the listing below is very similar to the function `findpitches( )` given in section 11, but it quantises the slider settings to cents and gives the

results in cents. It is not recommended for practical use, as it is slow and computationally inefficient. It quickly generates very sparse result vectors (*i.e.* ones full of lots of zeros). Even for the simple case illustrated above in Figures D.1 and D.2 the result vector was about 1000 points long. If you start to add more sliders, the vector quickly gets longer.

```
def findcentpitches(sliders):
    sliders = np.round(100 * sliders)
    lsliders = len(sliders)
    cpitches = np.array([1])
    addone = np.array([1])
    for j in range(0, lsliders):
        if sliders[j] > 0:
            y = np.zeros(int(sliders[j]))
            delta = np.append(y, addone, axis=0)
        else:
            delta = addone
        cpitchesadd = np.convolve(cpitches, delta)
        cpitches = [sum(n) for n in zip_longest(cpitches, cpitchesadd, fillvalue=0)]
    return cpitches
```

The math (equation D4) and its implementation (the function `findcentpitches( )`) works well enough, but it cannot be justified in terms of 'efficiency'. Also, the idea of quantizing to the resolution of cents is overkill in the case of the VOLTAGES expander. We already have seen that it is hard enough setting the sliders to one of 13 positions along their length (to encode the semitones in an octave). Can you rally imagine trying to set them to one of 1201 positions to encode all the cent values in an octave?

The truth is that – whilst the cent is a convenient 'academic' concept – it is practically too fine a resolution to be useful. It is finer than the Just Noticeable Difference ('JND') limen in frequency [17], meaning that it is inaudible as a frequency change without other cues, such as beating.

Rather than dismiss the "cent" approach on grounds of unnecessarily high resolution and low computational efficiency, we shall leave it here as an important lesson and move on to another important modification of our 'convolution method'.

In this next modification, we shall not make any approximations to pitch (*e.g.* by quantising) Rather, we shall be content to store the exact pitches resulting from all the combinations of sliders. This will be usable even when some sliders are at 'irrational number settings' (as another candidate approach to the 'non-integer settings' problem might have been to quantize at a common divisor of all slider settings). The cost of storing the exact pitches will be that we will have to maintain a separate pitch vector as well in addition to the count vector, *cpitch,* whose indices encode the pitch.

Our new method might appear at first sight to have little advantage in terms of computational efficiency. However, it will be noted that the greater number of the tunings in the catalogue of section 13 have several sliders in the '0' position. These sliders do confer efficiency savings in this new method, so an overall advantage is retained for tunings typical of our application.

We start, as before (equation 45), with a *count* vector, which is to be initialised to 1 but now add a *pitch* vector, initialised to 0:

$$count_0 = [1] \tag{D.5}$$

$$pitch_0 = [0]$$

The physical justification of this initialisation is the fact that, even with no sliders in non-zero settings, VOLTAGES produces output at 0; a starting count = 1 at pitch = 0 is required as starting point.

Then, for all sliders, we follow the basic 'recipe' for the 'convolution method', viz:

> *each additional slider adds a copy of the sequence notes that were produced by what was there already, modulated up to the pitch corresponding to the setting of the new slider.*

For slider 1, we have an existing situation of [$count_0$, $pitch_0$] and we must add to this a copy modulated in pitch up to the setting of slider 1: $sliders_1$.

We shall deal with the pitch vector first.

The existing pitch vector is $pitch_0$ and modulating it up to frequency $sliders_1$ is simply a matter of adding $sliders_1$. So, the new pitch vector is:

$$[pitch_1] = [pitch_0] + [pitch_0]sliders_1 \qquad \text{(D.6)}$$

Those little square brackets look harmless enough, but they conceal the fact that $pitch_i$ is a vector (a list of numbers), whereas $sliders_1$ is just a scalar (a number). So, equation D.6 really ought to be written out long-hand for those who don't understand how scalars are added to vectors – $sliders_1$ is added to each individual element of $pitch_0$ to make $pitch_1$. It doesn't matter yet, because $pitch_0$ only has one element (see eqn D5), so we'll deal with this later.

Now we'll look at the count vector.

The existing count vector is $count_0$ and we need to count the number of instances there are at the new modulated frequencies we've just created. Since these were copies of what already happened before we modulated them up, the counts must be copies of what we had before ($count_0$) so the new count vector is:

$$[count_1] = \big[[count_0]|[count_0]\big] \qquad \text{(D.7)}$$

where the vertical line between the two $count_0$ vectors denotes concatenation (*i.e.* listing the elements one after another in a longer vector).

Similarly,

$$[count_2] = \big[[count_0]|[count_0]|[count_0]|[count_0]\big]$$

Having seen how this works for the first slider, it is easy to generalise to the nth slider and (given there are 8 sliders) to find the solution for the pitch:

$$pitch_n = pitch_0 + \sum_{j=1}^{n}\big[pitch_{j-1}\big]sliders_j \qquad \text{(D.8)}$$

and the count:

$$[count_n] = \left[[count_0]|[count_0]|[count_0]|[count_0] \ldots \ \ldots |[count_0]\right] \qquad \text{(D.9)}$$

We should note (from D.5) that $count_0$ = [1], so $count_n$ is simply a vector of ones. There are, by definition, as many elements in $count_n$ as there are in $pitch_n$ which, in this case is 2^n. For our final solution, n=8 (the number of sliders on VOLTAGES). So 2^n amounts to 256 elements.

At this stage, it doesn't sound as if our new approach, expressed in equations D.8 and D.9 is very efficient; it generates vectors 256 elements long. However, we are going to add a twist, which makes things better.

When we stated our basic recipe for the convolution method, we could have noticed a simplifying step:

> *each additional slider adds a copy of the sequence notes that were produced by what was there already, modulated up to the pitch corresponding to the setting of the new slider unless that additional slider is set to zero, in which case we simply add one to the existing count.*

So, we shall produce a new function which implements equations D.8 and D.9 with the 'twist' that we miss out the duplication of pitches by sliders set to zero (the 'modulation' produced by the zero slider leaves the frequency the same, so the new slider produces a copy of the existing notes, which is reflected by adding one to the count – no new elements are added to the *pitch* or *count* vectors).

The function `findnonintegerpitches( )` is shown listed below.

```
def findnonintegerpitches(sliders):
    count = np.array([1])
    pitch = np.array([0])
    lsliders = len(sliders)
    for j in range(0, lsliders):
        if sliders[j] == 0:
            count = count + 1
        else:
            countex = count
            pitchex = pitch + sliders[j]
            count = np.append(count, countex, axis=0)
            pitch = np.append(pitch, pitchex, axis=0)
    # Now sort to unique pitch values:
    uniquepitches = np.array(np.unique(pitch))
    newcount = np.zeros(len(uniquepitches))
    for i in range(0, len(uniquepitches)):
        found = np.array(np.where(pitch == uniquepitches[i]))
        newcount[i] = np.sum(count[found])
    pitch = uniquepitches
    count = newcount
    return count, pitch
```

The act of limiting the 'convolution' to include only the non-zero sliders limits the length of the resulting *pitch* and *count* vectors significantly.

For example, in a test with a typical tuning:

sliders = [12, 12, 7.1, 0, 2.97, 0, 0, 0]

the vectors are length 256 when all sliders are included in the convolution and length 16 when only the non-zero sliders are included. The computational advantage is obvious.

A disadvantage of this implementation of the 'convolution method' over those where the pitch is encoded in the *cpitch* vector indexing (both the original integer 'convolution method' introduced in section 11 and the quantised version introduced earlier in this section) is that the pitch vector has pitch occurrences in non-ascending order. This is rectified in the function `findnonintegerpitches( )`, which arranges the pitches into unique pitches in ascending order and calculates the number of occurrences of each pitch by summing over all the relevant counts and re-ordering these in the same pattern as the pitches.

Appendix E *Conventional Approaches*

We saw in section 7 a laborious description of how slider settings add together to produce pitch sequences in the context of two simple slider settings (equations 8 and 14). This process is part of the conventional mathematical tools used to solving these types of problems within the field of combinatorial optimisation. Given this context we should spend some time looking at conventional solutions – not least to generate a yardstick against which to judge the performance of the 'convolution method' which we are using in their place.

We noted in section 7 (see the box after equation 11) that we would like to understand ways to calculate pitches associated with "all possible sums of *n* slider settings".

There are two important ways to consider all possible sums of the settings of n sliders, where the sliders are individually activated by independent gate signals and the outputs of all sliders are summed.

The first of these two approaches lists all possible gate activation patterns and then 'blindly' computes the sum for each pattern. The gate vector (see equations 1 & 4) is eight bits long, meaning that it can have 256 distinct values [0:255]. Therefore, it is possible to identify all possible output sums by counting through the 256 values of the gate vector sequentially:

$$\begin{gathered} G_0^T = [0,0,0,0,0,0,0,0] \\ G_1^T = [0,0,0,0,0,0,0,1] \\ G_2^T = [0,0,0,0,0,0,1,0] \\ \cdots \\ G_4^T = [0,0,0,0,0,1,0,0] \\ \cdots \end{gathered} \qquad \text{E.1}$$

$$\begin{gathered} G_{128}^T = [1,0,0,0,0,0,0,0] \\ G_{129}^T = [1,0,0,0,0,0,0,1] \\ \cdots \\ \cdots \\ G_{255}^T = [1,1,1,1,1,1,1,1] \end{gathered}$$

If this is applied to equation 4, all the outputs of VOLTAGES are computed given the slider setting, W_V. This approach is long-winded and inelegant – but it is very easy to program on a computer and (perhaps surprisingly) takes very little time to compute.

A second, more elegant approach exploits some of the combinatorial functions widely available in modern programming languages and benefits from the intuitive understanding of the operation of the summation presented in section 7.

This second approach recognises that one of the gate patterns (G_0) will always produce a zero output (the 'root' note, which will always be a member of the 'pitch' sequence – even if no sliders are set to zero).

Similarly, eight of the gate patterns [G_1 G_2 G_4 G_8 G_{16} G_{32} G_{64} G_{128}] are guaranteed to be able to activate only one slider, as there is only one active bit in these gate patterns.

28 of the patterns (including G_{129}) are guaranteed to be able to activate only two sliders, as there are only two active bits in these gate patterns.

56 of the patterns (including G_7) are guaranteed to be able to activate only three sliders, as there are only three active bits in these gate patterns.

70 of the patterns (including G_{15}) are guaranteed to be able to activate only four sliders, as there are only four active bits in these gate patterns.

56 of the patterns (including G_{31}) are guaranteed to be able to activate five sliders, as there are five active bits in these gate patterns.

28 of the patterns (including G_{63}) are guaranteed to be able to activate six sliders, as there are six active bits in these gate patterns.

8 of the patterns (including G_{254}) are guaranteed to be able to activate seven sliders, as there are seven active bits in these gate patterns.

Finally, 1 of the patterns (G_{254}) is guaranteed to be able to activate eight sliders, as there are eight active bits in this gate patterns.

The general rule for the number of ways to combine r sampled items from a group of n objects is:

$$C(n,r) = \frac{n!}{(r!\,(r-n)!)}$$

This rule is also known as the 'Binomial Coefficient': $\binom{n}{r}$

Now, if we were to treat these groups together, we would be disappointed, because we would notice that:

$$1 + 8 + 28 + 56 + 70 + 56 + 28 + 8 + 1 = 256 \qquad \text{E.2}$$

In other words, we would have to consider the same total number of cases as before, when we counted through all the gate patterns sequentially. It would take just as long to compute, and we would have gained nothing! So, what – if any – is the merit of looking at this clever sorting of the gate patterns?

I mentioned above that the gate patterns were "guaranteed to be able to activate n sliders".

What if there weren't n sliders in non-zero settings? And what if the n active bits of a particular gate pattern coincide with sliders set to zero?

In both these cases, the gate pattern would not activate *n* sliders. It would activate fewer than *n* sliders and would cause a repeat of an output generated by a previous gate pattern.

The precise identification of the groups of gate patterns described above allow tests to be made to detect such repeats, which could potentially save computational effort. But these tests themselves are not 'free' and it turns out that it is easier just to take the hit and calculate the sum of all 2^n patterns; the whole strategy becomes dubious in this application. However, we shall look at the implementation of this method in code for the sake of completeness before looking at one means of reducing the computational load for practical tunings.

The sums of slider settings taken *n* at a time is computed using functions associated with the Binomial Coefficient for real arguments.

In MATLAB (/Octave) this is offered by the function `nchoosek()` and an example of the problem of finding all possible sums is presented in [18]. In Python, the function `math.comb()` evaluates the Binomial Coefficient, but it does not identify the indices of the samples that must be joined in any combination as `nchoosek()` does. Instead, the `combination(n,r)` function within the itertools library can be used.

The following Python function, `findpitchescomb()`, has been written to identify all possible pitches using the conventional combinatorial method:

```
def findpitchescomb(sliders):
    # find pitches from integer sliders tuning
    # using conventional combinatorial method
    lsliders = len(sliders)
    pitches = np.array([0])
    for k in range(1, lsliders):
        intermed = list(itertools.combinations(sliders, k))
        pitchesadd=np.zeros(len(intermed))
        for j in range(0, len(intermed)):
            pitchesadd[j] = sum(intermed[j])
        pitches = np.append(pitches, pitchesadd)
    return pitches
```

This function is seen to be short, in that the code is compact – notice it is slightly shorter than our integer 'convolution method', implemented as `findpitches()`(see section 11). However, `findpitchescomb()` has been poorly written. The nested 'for' loops could probably be handled more efficiently by better use of the indexing resources of the vectors (/lists).

Returning to the first of the 'conventional' approaches listed in this Appendix, a Python implementation of the crude 'consider all gate vector patterns one-at-a-time' strategy, enacted by stepping through the 2^n possible values associated with the *n* bits in the gate vector coming from the Turing Machine gives even more opportunity for poor programming to slow things down. This time, the difficult is in forming the gate vector which, in the following function, `findpitchesbruteforce()` , is done within another nested pair of 'for' loops:

```
def findpitchesbruteforce(sliders):
    # find pitches from integer sliders tuning
    # using brute-force consideration of
    # all 2^n gate patterns
    lsliders = len(sliders)
    pitches = np.array([0])
    bits = range(0,lsliders)
    for k in range(1, (2 ** lsliders)):
        pitchadd = np.array([0])
        for bit in range(0, lsliders):
            pitchadd = pitchadd + bitget(k, bit)*sliders[bit]
        pitches = np.append(pitches, pitchadd)
    return pitches

def bitget(value, n):
    # get bit n from the binary integer 'value'
    # (see MATLAB's 'bitget')
    return (value >> n) & 1
```

Python has no direct equivalent of MATLAB's 'bitget( )', so the bitget() function listed above was called by `findpitchesbruteforce()`.

This 'brute force' code again is compact in terms of lines of source. But inefficient in operation, – worse than `findpitchescomb()` (by a factor of ~10 in execution speed), even though they both consider all 2^n gate patterns.

Even though it has not been well written, `findpitchescomb()`, is found to run surprisingly quickly. However, it is not as fast as the convolution method, implemented in `findpitches()`.

In an informal test of ten repeat comparisons of the tuning:

$$sliders = [2, 2, 5, 0, 7, 0, 0, 0] \qquad \text{E.3}$$

`findpitches()` was found to run in 62% of the time taken by `findpitchescomb()`.

There is, however, one method which will improve the speed of the 'conventional' techniques which already is being exploited by `findpitches()`.

This involves removing the inactive sliders which are set to zero, such that the tuning of equation E.3 becomes:

$$sliders = [2, 2, 5, 7]$$

Under this tuning, the performance changes, such that in another experiment over 10 repeat comparisons, `findpitches()` was found to run in 212% of the time taken by `findpitchescomb()`, as a result of significant speed increase in `findpitchescomb()`, and no change in the speed of `findpitches()`.

It is seen that the convolution method of section 11 is computationally efficient compared to conventional methods. However, in the 'sparse' tunings which often arise in the present tuning strategy (see section 13), this efficiency advantage is lost if 'redundant' zeros are stripped from the *sliders* vector before it is passed to the conventional search strategies.

This observation led to production of the `cleansliders()` function as a utility in the software resources (see Appendix C). Similarly, any sliders set to 12 (or integer multiples of 12) to effect octave expansions (see section 10) should be removed from the *sliders* vector before it is processed by the 'conventional' search methods – this is achieved by the `cleanoctaveshifts()` function; such change will NOT impact the performance of the 'convolution method' or its implementation in `findpitches()`.

References

[1] https://modwiggler.com/forum/viewtopic.php?t=198085

[2] https://www.musicthing.co.uk/Turing-Machine/

[3] https://youtu.be/Le26BIqB8Y8?t=333

[4] https://www.musicthing.co.uk/Turing-Voltages-Expander/

[5] https://www.birthofasynth.com/Scott_Stites/Pages/Klee_Birth.html

[6] http://electro-music.com/forum/phpbb-files/know_the_klee_draft4_198.pdf

[7] https://www.youtube.com/watch?v=Yeei_xaUhoE

[8] https://www.youtube.com/watch?v=uNkGzJhYWbA

[9] https://en.wikipedia.org/wiki/Chromatic_circle#Pitch_constellation

[10] https://en.wikipedia.org/wiki/Convolution#Definition

[11] https://en.wikipedia.org/wiki/Kronecker_delta

[12] https://www.python.org/

[13] https://numpy.org/

[14] https://en.wikipedia.org/wiki/Voltage_divider

[15] https://sound-au.com/project01.htm#s1

[16] https://en.wikipedia.org/wiki/Cent_(music)

[17] https://en.wikipedia.org/wiki/Just-noticeable_difference

[18] https://uk.mathworks.com/matlabcentral/answers/1437029-how-can-i-get-the-sums-of-all-possible-combinations-of-the-values-of-a-vector