DYAD CONTROLLED ADDITIVE SYNTHESIS

by James Dashow

The rather dramatic increase in computational power of the newest CPUs for personal computers breathes new life into additive synthesis as a practical means for the generation of complex time-varying electronic sounds. To be effective, additive synthesis will often require a large number of oscillators, or in more advanced languages, the use of a multiple oscillator unit that does the work of internally summing several oscillators via an optimized loop structure.

There still remains the problem of controlling each oscillator with its own amplitude envelope and frequency variance, requiring that the composer prepare and manipulate a large amount of data: separate amplitude, pitch and function tables, if desired, for each oscillator.

The composer can, of course, design a high level score data routine that automatically adjusts these parameters based on global input variables; the group of oscillators and associated amplitude envelopes can be considered as a single unit with a small amount of data as input which is elaborated by the score routine into all the specific information necessary for the additive synthesis instrument to execute as desired. Even then, the composer will have to specify the individual frequency components in some way, and while additive synthesis offers what amounts to infinite variety of frequency combinations, this very infinity can prove an obstacle for sustained compositional use.

The method suggested here considers several oscillators as a single unit while the frequency of each oscillator is determined by the application of mathematical procedures that produce a set of results (here interpreted as frequency components) as a function of a very small number of variables (no more than 5, often less). There is, in this approach, no separate amplitude control over the individual components, only an amplitude re-scaling of the final output sum of the several oscillators. That is, one of the constraints of this concept is to accept equal amplitudes (numerically equal, not necessarily perceived as equally loud) of the separate frequencies as a characteristic of the generated sound, in the same way we accept the constraints of the partial amplitudes that make up the sound of the oboe or the flute as being simply those instrumentsí characteristic timbre. It will become clear that this particular limitation hardly puts us at any disadvantage with respect to the variety of sound obtainable from this method. Nevertheless, an adventurous user can easily apply separate amplitude controls to each oscillator, or group the oscillators into frequency zones (by filtering or by programmed logic) and apply amplitude envelopes to those zones rather than to each oscillator.

One standard use of additive synthesis is to declare certain frequencies, or even a single frequency, along with others that might have particular relationships (elaborated under program control) to the principle frequencies; these derived frequencies would be the timbral context for the principle frequencies. Further timbral manipulation can be achieved varying the function table(s) being used for the various oscillators making up the spectrum as a whole. The composer could conceivably design combinations of frequencies, harmonic and, especially, inharmonic, in order to characterize specific musical intervals, such that each interval has its set of characteristic timbres. Musical development of this notion would be through the mutual exchanging, sharing or excluding of timbral components while various combinations of intervals are being played.

The approach suggested here is based on dyads, specific intervals (in this case the intervals of the Western chromatic scale) that are made to generate sets of frequencies, often inharmonically related to each other, as the result of algorithmic procedures. The dyads, or intervals, needn't be limited to the chromatic scale, of course, but since the resultant frequency collections are generally inharmonic, it is hardly necessary to think in terms of intervals outside the well-tempered system. The advantage in this case is that the sounds have at least two frequencies, the dyad, that can in one way or another be picked up timbrally by live traditional instruments. Needless to say, any interval, Western-chromatic or non, can be used as input to the algorithms described below.

The composer chooses or invents an algorithm that will generate a set of frequencies. The equations of the algorithm are re-arranged so that it is the interval, or what will be called also the «generating dyad», that determines the way the algorithm generates the frequency components. The interval itself determines the frequency content produced by algorithm execution, and hence it is the interval that determines its own timbre. The two frequencies of the dyad are always present as two of the components of the generated frequency set. As we shall see, for any generating dyad there are a variety of frequency sets available (depending also on the type of algorithm), such that we can get many different groups of frequencies generated by the same dyad; and since the frequencies of the dyad are always two of the components we have a method not only for generating varieties of timbres, but also for structurally relating timbres by means of frequencies in common, the generating dyad itself. Using the same generating dyad with different algorithms yields a wide variety of electronic sounds all of which have at least the generating dyad in common among their frequency components.

This represents a very useful reduction of data that the composer has to worry about. Once the sound synthesis process has been programmed with the algorithm, the composer interfaces with the additive synthesis in terms of musical information rather than numeric: the generating dyad (as a musical interval or as a pair of pitches) and the necessary subsidiary «orchestrational» data: where in the spectrum the generating dyad is to be produced (register) and the timbral quality as influenced by no more than two additional factors (the function table and/or a timbral factor for the algorithm). The composer does not need to know all the frequencies generated by the procedure, on the contrary, at this point what counts is the varieties of sound, of timbre, that a generating dyad can be made to produce: work for the ears.

The first and easiest of these algorithms to implement is a set of frequencies generated by the following equation:

f(N) = f(0)+N*A [1] for N = 1,2,3....

This reads: the Nth frequency f(N) is generated by a starting frequency f(0) plus N times some factor A. If the starting frequency f(0) = 100 and A = 75, then

frequency 0 f(0) = 100 + 0 * 75 = 100 (the starting hz) frequency 1 f(1) = 100 + 1 * 75 = 175 frequency 2 f(2) = 100 + 2 * 75 = 250 frequency 3 f(3) = 100 + 3 * 75 = 325 etc.

Note the intervallic relationships between the frequency components. Between the starting frequency f(0) and f(1) the interval is 175/100 = 1.75; between f(2) and f(1), the interval is 250/175 = 1.4286; between f(3) and f(2), 325/250 = 1.3; the intervals get smaller the higher the N. This algorithm generates frequencies with equal differences between components which means the intervals will always get smaller the higher the N. This is, of course, the same frequency relationship for the upper sidebands of FM synthesis. Here, however, there is no algorithmic dependent time-varying dynamic change of the amplitudes or phases of the components as with FM, except what the composer wishes to add with the appropriate envelope controls. And these controls can be freely designed and manipulated, again, in contrast to the fixed necessary evolution of sidebands with FM.

To use the generating dyad idea, we say: let any two of the frequencies generated by equation (1) be the two frequencies of the dyad. For example, lets use an interval of an octave and a minor third, which has a ratio of 2.3784; if the f(0) is 100 Hz, the component an octave and a minor third above it is 237.84 Hz. We can have then the octave and a minor third as component f(2) or as f(3) or as f(4) or any. Let’s put it on f(2). Then since we know the frequency for f(0) (100) and for f(2) (237.84), we get:

f(0) = 100 = 100 + 0 * A
f(1) = ? = 100 + 1 * A
f(2) = 237.84 = 100 + 2 * A

and we can calculate the A factor from the expression for f(2):

A = (237.84 - 100) / 2
= 68.92

Now we can generate a set of frequencies f(N) = 100 + N * 68.92 that will yield 100 as f(0) and 237.84 as f(2):

f(0) = 100 = 100 + 0 * 68.92 f(1) = 168.92 = 100 + 1 * 68.92 f(2) = 237.84 = 100 + 2 * 68.92 f(3) = 306.76 = 100 + 3 * 68.92 etc.

We can do the same thing if 100 Hz is f(0) and 237.84 is f(5):

f(5) = 237.84 = 100 + 5 * A
A = (237.84 - 100) / 5 = 27.568

And the frequency set f(N) with the generating dyad at position 0 and 5 (f(0) and f(5)) in the spectrum is produced by applying general equation (1) with A and f(0) set appropriately,

f(N) = 100 + N * 27.568.

This will get us a different set of frequencies but with the 100 and 237.84 components in common. A different timbre is produced but maintains a continuity between them, that of the common generating dyad.

Note, however, that the way we calculated A was by dividing the difference between the two frequencies of the generating dyad by the difference between their position numbers. So we can generalize by writing that the A factor is:

A = (UP - LP) / (NU - NL) [2]

where UP (or Upper Pitch) and LP (Lower Pitch) are the frequencies of the two members of the generating dyad, and NU (Upper component Number) and NL (Lower component Number) are the positions of the dyad (their «register» in more musical terms) in the frequency set, or spectrum.

In these two examples, we've always called for the LP, the bottom pitch of the dyad, to be generated as f(0), the starting frequency of the spectrum. This not need be the case. We can just as easily call for LP = 100 Hz on f(1) and UP = 237.84 Hz on f(3). In that case we get:

f(0) = f(0) + 0 * A
f(1) = 100 = f(0) + 1 * A
f(2) = ? = f(0) + 2 * A
f(3) = 237.84 = f(0) + 3 * A

Now we have two unknown quantities in the $f(3)$ expression, but with a little algebra we can eliminate one of them: subtract

$$f(3) - f(1) = (f(0) + 3 * A) - (f(0) + A)$$
$$f(3) - f(1) = 2*A$$

since $f(3)$ and $f(1)$ are the generating dyad frequencies,

$$237.84 - 100 = 2*A$$

$$A = 68.92$$

and to find $f(0)$, we can go back to $f(1)$:

$$f(1) = 100 = f(0) + 68.92$$
$$f(0) = 100 - 68.92 = 31.08$$

So the frequency set $f(N)$ for the generating dyad at positions 1 and 3 will be

$$f(N) = 31.08 + N * 68.92,$$

and the general expression for $f(0)$ is then

$$f(0) = LP - NL*A, \qquad [3]$$

or, the beginning Hz of the frequency set is the lower pitch of the generating dyad minus it’s position number times the constant difference factor between frequencies in the set. If NL is 0, then $f(0) = NL$, as we saw in the first examples.

So for any dyad, any interval, any pair of frequencies, equations [2] and [3] provide values for A and for $f(0)$, which are then plugged into repeated uses of equation [1] to generate a set of frequencies $f(N)$ that will have the dyad at the declared positions in the spectrum. Changing the position of the dyad frequencies changes the $f(0)$ and A values and thereby changes the timbre. But the dyad is always there. You can play as many other frequencies above and sometimes below the generating dyad members as you like depending on the desired timbre. If the generating dyad is on components 3 and 7, for example, then you might decide to play frequencies $f(2)$ through $f(8)$ or $f(0)$ through $f(11)$, depending on the sound quality required for the musical context.

Additional timbral variety is obtained by using complex wave tables for the oscillators. This method accepts the constraint that all oscillators playing use the same wave table; the sum of oscillators is considered a unit instrument here, to keep the score data within more practical dimensions. Again, the composer can specify a different wave table for each oscillator, and together with time-varying amplitude control or the re-generation of the wave table during performance, achieve a high degree of transformational control over the timbral evolution.

In this instance, we not only can generate a wave table with different harmonic partials starting with the first partial, say the sum of the first 4 odd partials (using gen10: 1,0,0.333,0,0.2,0,0.167), we can also generate an arbitrary set of partials that have inharmonic relationships between themselves, for example using gen09: 6 1 0 11 0.3 0 19 0.15 0. Here the wave table contains the 6th, 11th and 19th partial (with diminishing amplitudes) whose ratios 11/6, 19/6 and 19/11 are not those of the equal tempered scale.

To use this wave table in the additive synthesis instrument, the composer decides which of the 3 partials will actually play the frequency assigned to each oscillator, and then divide that frequency by the chosen partial number. The other two partials will play that frequency (the f(N) divided by the chosen partial number) at the ratios between the partials generating inharmonic frequency relationships >for each frequency f(N) in the additive synthesis set<. If we call for 5 frequencies, f(0) through f(4), and play each frequency with the wave table containing the partials 6t,11 and 19, then the additive synthesis sound will be consist of 15 frequencies, not just five, with complex inharmonic and harmonic relationships between them. Timbral variety is assured.

In the first example above, if we decide to play all each frequency in each oscillator on the 6th partial, the lowest in this wave table, we can adjust the f(N) set of frequencies by dividing each one by 6, or f(N) = f(N)/6. But to save computation time, it's more efficient to divide the generating dyad's frequencies by 6 (the chosen partial number) before beginning the f(N) calculations. We can also play the f(N) frequencies on the 11th or the 19th partial (perhaps adjusting the amplitude of the latter a bit) as well, dividing the generating dyad frequencies by 11 or 19; this will produce frequencies below the f(N) frequency (the frequency generated on the 6th partial will be 6/11 lower than that generated by the 11th partial (etc.) which can yield some very rich and musically useful timbres. Designing various wave tables for a single dyad and a single algorithmic realization yields a rich variety of timbres all generated around the same two frequencies, the generating dyad.

We can make a small Csound instrument out of this equation with the following code; here the maximum size is 6 oscillators, but you can easily expand this.

```
sr      =   44100
kr      =   44100  ;<- note sr=kr!!!!
ksmps   =   1
nchnls  =   1
```

```
        instr   1
; p4    = amplitude, in absolute values
; p11   = overall re-scale after reducing
;    amplitude by number of oscillators playing
; p11 is in the range 1.3 to 1.6
; p5    = UP, p6 = LP, p7 = NU, p8 = NL, p9 = partial number, p10 = wave table number
        iup     =   cpspch(p5)/p9
        ilp     =   cpspch(p6)/p9
        iafac   =   (iup-ilp)/(p7-p8)
        if0     =   ilp - iafac*p8
        inf     =   p10;
; p11 = highest oscil num;  p12 = lowest oscil num;  1 thru 6 here
        inum    =   p11 - p12 + 1
; now rescale the amplitude by the number of oscils playing
        iamp    =   p4 * inum * p11
; now calculate the components for num Hz as init, and play in perf
        kcount      init inum   ; p-time always reset
        atotsig =   0           ; audio variable that accumulates the sum of oscili outs for
                                ; each sample
        if   p11 == 0   goto    x0
        if   p11 == 1   goto    x1
        if   p11 == 2   goto    x2
        if   p11 == 3   goto    x3
        if   p11 == 4   goto    x4
x5:
; this is the algorithm, equation (1) with specific
;  values for the f(5), the sixth Hz component
        if5     =   if0 + 5*iafac
        asig5       oscili   1, if5, inf, iphs
        atotsig =   asig5;
;   FIRST entry point doesn't need counter as there are always at least 2
;    oscili's playing
x4:
        if4     =   if0 + 4*iafac    ; this is the algorithm (1) generating f(4)
        asig4       oscili   1, if4, inf
        atotsig =   atotsig + asig4
        kcount  =   kcount - 1
        if   kcount == 0  goto xend
x3:
        if3       =    if0 + 3*iafac      ;  etc.
        asig3          oscili    1, if3, inf
        atotsig  =    atotsig + asig3
        kcount  =    kcount - 1
        if    kcount == 0  goto xend
x2:
        if2       =    if0 + 2*iafac
        asig2          oscili    1, if2, inf
        atotsig  =    atotsig + asig2
        kcount  =    kcount - 1
        if    kcount == 0  goto xend
x1:
        if1       =    if0 + iafac
        asig1          oscili    1, if1, inf
        atotsig  =    atotsig + asig1
        kcount  =    kcount - 1
        if    kcount == 0  goto xend
x0:
        asig0          oscili  1, if0, inf
        atotsig  =    atotsig + asig0
xend:
;  here you can rescale the atotsig and control it with an amplitude envelope.
        atotsig  =    atotsig * iamp   ; etc.
        out            atotsig
        endin
```

Perhaps with a score like this, using an interval of an octave and a minor seventh. Note the differences in timbre between f1 and f2:

```
f1    0    4096   9     6      1     0     11    1      0     19     1      0
f2    0    4096   9     7      1     0     11    1      0     18     1      0 \
           26     10

;1    2    3      4     5      6     7     8     9      10    11     12
;ins  act  dur    amp   up     lp    nu    nl    parts  fno   hiosc  loosc
i1    0    5      100   10.00  8.02  4     1     6      1     6      1
i1    +    5      100   10.00  8.02  4     1     7      2     6      1
```

Since this algorithm can be expressed also in the form

$f(N) = f(N-1) + A$ [1b]

you can program the Csound instrument in a somewhat more computationally efficient fashion avoiding the multiply for each f(N).

You can also use small frequency values, like 1.1 Hz, to get interesting phasing and chorus effects. You must always divide by the partial number being used in the wave table, even with these small Hz values.

Or, you can use *linseg* or a data table to control a succession of intervals based on the same LP; the variables f1, f2, f3, f4 etc. would be controlled by the *linseg* output in k-time or re-initialized (*reinit*) with new values for each interval read off a data table. Other variants can easily be invented.

A closely related algorithm is one which produces frequencies that are equal proportions away from each other, not equal distances, or

$f(N) = f(0) * M^N$ [4a]

which can also be written in the form

$f(N) = f(N-1) * M,$ [4b]

where M is a constant multiply factor.

Using form [4b] in the instrument avoids having to do an exponential operation for each f(N) calculation, not an efficient procedure by any means.

This equation is the prototype for the equal-tempered scale, or for that matter, any scale whatsoever. For the Western chromatic scale, the keys on the piano are tuned according to

$f(N) = f(0) * 1.0594631^N$

or $f(N) = f(N-1) * 1.0594631$

the multiply factor M is simply the 12th root of 2, the semitone. If we use the generating dyad approach we are in effect generating «scales» that have inharmonic intervals as «scale steps». The procedure is substitute the known values for the generating dyad, UP and LP, along with their (composer chosen) position, NU and NL, in the f(N) collection, or spectrum, do a little algebra and derive expressions, as before, for the f(0) and M values that are plugged into equation [4b] which is programmed into the Csound instrument.

UP = f(0) * M^NU
LP = f(0) * M^NL

now we eliminate the f(0) variable by dividing

UP/LP = M^NU/M^NL = M^(NU-NL)

and solve for M (ln is the natural logarithm, to the base e):

M = exp(ln(UP/LP) / (NU-NL)). [5]

Then we can find

f(0) = LP/M*NL [6]

In the Csound instrument above, where we've written Afac and f(0), equations [2] and [3], substitute

```
   imfac  =  exp(log(iup/ilp))/(p7-p8))
   if0 =  ilp/(imfac^p8)
```

and change every reference to *iafac* in the program to *imfac*.

Finally where the algorithm [3] is written specifically for each f(N) in the instrument, substitute [4a] - or substitute [4b] for [1b] if you've worked that version out -

```
if5  =  if0 * pow(imfac^5)
```

etc.

There are many many more algorithms that can be explored. For example, try an adaptation of the «partial stretching» algorithm suggested by Steve MacAdams. The equation for the harmonic partial series is

f(N) = f(0) * N.

MacAdams developed the notion of stretching the partials by writing the equation as

f(N) = f(0) * N^S [7]

where S is the «stretch factor». Setting S to various values will generate a group of frequencies whose relationships quickly become inharmonic, producing some interesting timbres, especially if each f(N) is played by a complex wave table.

The relevant generating dyad equations are worked out as follows:

UP = f(0) * NU^S
LP = f(0) * NL^S

UP/LP = NU^S / NL^S = (NU/NL)^S

S = ln(UP/LP) / ln(NU/NL) [8]

and then f(0) = LP / NL^S. [9]

Again, in the Csound instrument, you substitute [8] and [9] for [2] and [3], and in the body of the instrument substitute the appropriate form of [7] for [1b], the calculation of the algorithm for each specific frequency.

You can work with more than one extra variable; one series of algorithms use a multiply, add procedure as the basis, another series uses add, multiply as the basis. These algorithms have both a Multiply and an Add factor, which requires the composer to specify not only the generating dyad and the position of the dyad frequencies in the sound, but either an Add factor, and the algorithm finds the Multiply factor, or the reverse - the Multiply factor and the algorithm finds the Add factor.

The simplest Multiply, Add algorithm is

f(N) = f(0)*M^N + N*A [10]

while the simplest Add, Multiply is

f(N) = M^N * (f(0) + N*A) [11]

Now there are two generating dyad equations for [10] and for [11], depending on whether the composer supplies the M factor or the A factor. Equation [11] can be solved for «given A, find M» mode only if the lowest pitch of the dyad, LP, is also the f(0), the bottom frequency of the spectrum. Note that in almost all of these cases, where you might specify the Lower Pitch, LP and the Upper Pitch, UP, of the generating dyad at positions 1 through whatever, where 1 is the lowest frequency in the spectrum, you must refer to the bottom position as 0 in order for the mathematics to work properly, that is, you should subtract 1 from the given NU and NL values (unless you are already working with 0 through N-1 to indicate 1 through N).

For (10): given M (the multiply factor) find A (the Add factor), the dyad equations for the Csound instrument are («mx» is a temporary work variable):

```
mx    =      M^(NU-NL)

A = ( UP - (mx*LP) ) / ( NU - (mx * NL) )
```

or given A find M mode, algorithm (10) equations are:

```
mx = log( ( UP - (A * NU) ) / ( LP - (A * NL) ) )

M = exp( mx / (NU - NL) )
```

and in both modes solve for f(0):

```
f0 = LP;
if (NL != 0.0) {
      div = pow(M,NL);
      f0 = (LP - (A * NL) )/ div;
         }
```

For (11): given M find A mode, the equations are (again it is useful to use couple of extra work variables, uxpn and lxpn):

```
uxpn = M^NU
lxpn = M^NL
A = ( (UP / uxpn) - (LP / lxpn) ) / (NU - NL);
```

in given A, find M mode, the equations can be resolved only with LP being declared the f(0) frequency, NL = 0 for these equations:

```
mx = log( UP / ( LP + (A * NU) ) );
M =  exp(mx/NU);
```

and in both modes you solve for f(0):

```
f0 = LP;
```

```
if (NL != 0.0)  /* true only for given M find A mode */
f0 = (LP / lxpn) - (A * NL);
```

As a final specific example, another equation for a frequency set generator has proved extremely flexible:

f(N) = M * f(N-1)^2 / f(N-2)   [12]

This reads: the current frequency N in the spectrum is the product of some multiplier, or scalar factor, M times the preceding frequency in the set squared divided by the next-to-last frequency in the set. If M = 1, the ratio between the first and second frequencies, f(0) and f(1) determines the ratio between all succeeding frequencies. If that ratio is 1.0594631, we get the standard chromatic scale, for example.

If M is less than 1 then the ratios between frequencies get progressively smaller by exactly that value M for each frequency in the set, or a contracting interval series. If M is greater than 1, however, the reverse happens and we get an expanding interval series, which is quite unusual, and very useful for creating different kinds of timbre. In fact, even with M > 1, if the NU and NL are in the middle of a spectrum, say 5 and 3, respectively, it is easy to generate a frequency set whose lower f(N) frequencies have intervals that compact going toward the LP of the generating dyad and which then expand going beyond the UP of the generating dyad. In short, the M factor determines the character of the interval pattern in a way that offers the composer greater control over the evolution of the frequency set than in many other algorithms.

The program is a little more involved, as the first two frequencies, f(1) and f(2), of the set have to be derived from the generating dyad and the M factor. Note in this case we don't use f(0), or NL = 0, rather the lowest frequency in the set is N = 1.

```
mx = NU - NL;
lxpn = mx - 1.0;
if (lxpn == 0.0)
flp1 = UP;  /* in Csound, use if  go construct */
else {
expM = pow(M, (mx*lxpn*0.5));
lpxp = pow(LP, lxpn);

/* flp1 is the f(NL+1) */
flp1 = pow( (lpxp*UP/expM), (1.0/mx) );
        }
/* now find the first two frequencies in the spectrum set, f1 and f2  */
```

```
if ( NL == 1.0) {  /* if true, then LP is the bottom hz f1 */ f1 = LP; f2 = flp1;
      }
else  {
/* cycle through down progressively lower Hz until you get to f1
where each next f(L-1) = M * LP^2/flp1 */
MmU = LP;   /* initialize another auxiliary variable */ f2 = flp1;
for ( j = NL-1; j > 0; j—) {
f1 = M*MmU*MmU/f2;
f2 = MmU;
MmU = f1;
}  /* end of for loop */
}  /* end of else */
```

When you get here you have values for f1 and f2 which are the first two frequencies f(1) and f(2) in the spectrum; you then calculate f(3) on up to as many as desired using equation (12), the first few times would look like this:

```
f(3) = M * f(2)^2/f(1);
f(4) = M * f(3)^2/f(2);
f(5) = M * f(4)^2/f(3);    etc.
```

if the generating dyad frequencies were declared at NU = 5 and NL = 3, you would produce them as f(3) and f(5), above.

Other versions of Multiply, Add and Add, Multiply suggest themselves. In working out the generating dyad usage for these algorithms, I found that many have a wide range of possibilities, while others are more «temperamental» working better with certain intervals in specific frequency ranges and conditions (close NU and NL positions, or very wide NU NL positions, very small M factors, or very large Add factors, etc.).

In the generating dyad mode, some of these equations do not have solutions for «given Add, find Multiply», while others, like [11] above, have solutions for «given Add, find Multiply» only if f(0) is LP, i.e. the starting pitch is also the low pitch of the generating dyad, effectively eliminating one of the variables. These are marked in the list below.

It has also been found extremely useful to write a separate C program, independent of Csound, that displays on screen the f(N) results of any configuration of any of these algorithms. It helps to see where an f(0) frequency is a sub-audio value, say .003 Hz (but the algorithm produces interesting frequencies in the f(2) through f(7) range), or where some of the generated values are in the hundreds of thousands. For this reason the

Csound instrument parameters p11 and p12 allow you to avoid some of the frequency values that are useless, often the first ones or last ones in the array of values.

A compiled c-routine and its source code (inhntrvl.c inhntrvl.exe inhnmenu.man), that does exactly as described for PC can be downloaded from the writer's web site. The routine contains the generating dyad equations for these algorithms which can be easily adapted to the prototype Csound instrument illustrated above.

MULTIPLY then ADD

f(N) = f(N-1)*M^N + A*N -> ONLY in given M find A mode;
(the add and multiply version, f(N) = f(N-1)*(M^N + A*N),
yields virtually identical spectra)

f(N) = f(0)*N^M + A*N -> Stretch Factor, no restrictions;
these next two work correctly ONLY with NL > 1 and xx & yy > 0;
xx & yy can be fractions
f(N) = f(0)*M^N + A*X -> X = N+xx/N+yy

f(N) = f(0)*M^N + A*N^X -> X = N+xx/N+yy

f(N) = f(0)*M^N + A*N^M -> ONLY in given M find A mode;

ADD then MULTIPLY

for the next two, in «given A find M» mode, NL can ONLY be 1 (i.e. f(0) is LP). to avoid overflow errors, your init routines should default NL to 1 automatically in this mode. There are no restrictions for «given M find A» mode.

f(N) = M^N*(f(0) + A*N)
f(N) = N^M*(f(0) + A*N) -> Stretch Factor proc
f(N) = f(0)*(M + A*N) -> all OK, but watch the range of M & A for small LP and large NU, A can become negative and generate useless frequency values.
f(N) = f(N-1)*(M + A*N) often generates a set of expanding intervals going up..

the above is limited to NU = NL + 1 ALWAYS, i.e. 2 and 3, 4 and 5, 5 and 6. using A find M gets huge Hz values; use VERY small A and small intervals. also very effective with smallish M (1.1 to 1.8 range). Small intervals in the gen dyad work very well here also.

f(N) = M^N*(f(0) + A*N^M) -> ONLY in given M find A mode these work correctly ONLY with NL > 1 and xx & yy > 0,

xx and yy can be fractions. can be used only in «given M find A» mode.
f(N) = M^N*(f(0) + A*N^X) -> X = N+xx/N+yy
f(N) = M^N*(f(0) + A*X) -> X = N+xx/N+yy
f(N) = M^N*(f(0) + A^X) -> X = N+xx/N+yy
f(N) = N^M*(f(0) + A^X) -> X = N+xx/N+yy

There are many others.