Jacques Chareyron
Laboratorio Informatica Musicale
Dipartimento di Scienze della Informazione
Universita degli Studi di Milano
Via Moretto da Brescia, 9
I-20133 Milano, Italy
MUSIC@IMISIAM.bitnet

Digital Synthesis of Self-modifying Waveforms by Means of Linear Automata

Linear Automata Synthesis (LASy) is a new sound synthesis technique that uses cellular automata as models for the evolution of the synthesized sound. The technique has two positive characteristics: simplicity of the computational algorithm (real-time synthesis is possible using a common microprocessor); and the wealth of musical results that can be achieved. The LASy technique allows the creation of a large range of timbres, from simple tones (e.g., the Karplus-Strong plucked string algorithm may be simulated by some automata) to very complex ones. Due to the computational properties of the algorithm, the generation of a large family of sounds presents a unique character not found in classical synthesis techniques.

This article gives some guidelines for practical implementation and musical applications of the LASy algorithm. The drawbacks of the LASy technique are examined too, along with some methods to avoid them. Elements for the analysis of the synthesis algorithm are given in Appendix 1.

Cellular Automata

Cellular automata were first introduced in the late 1940s and have since aroused interest among a large number of mathematicians, physicists, and computer scientists (Burks 1970; Farmer, Toffoli, and Wolfram 1984). Even people who don't belong to this limited group of specialists will have heard of a popular example of a cellular automaton, John Conway's "Game of Life." "Life" is a model for the evolution of a community whose members occupy cells on a rectangular grid. Simple rules determine the birth, death, and survival of a single creature according to the numbers of live beings in neighboring cells, leading to fascinating, evolving configurations (Gardner 1970).

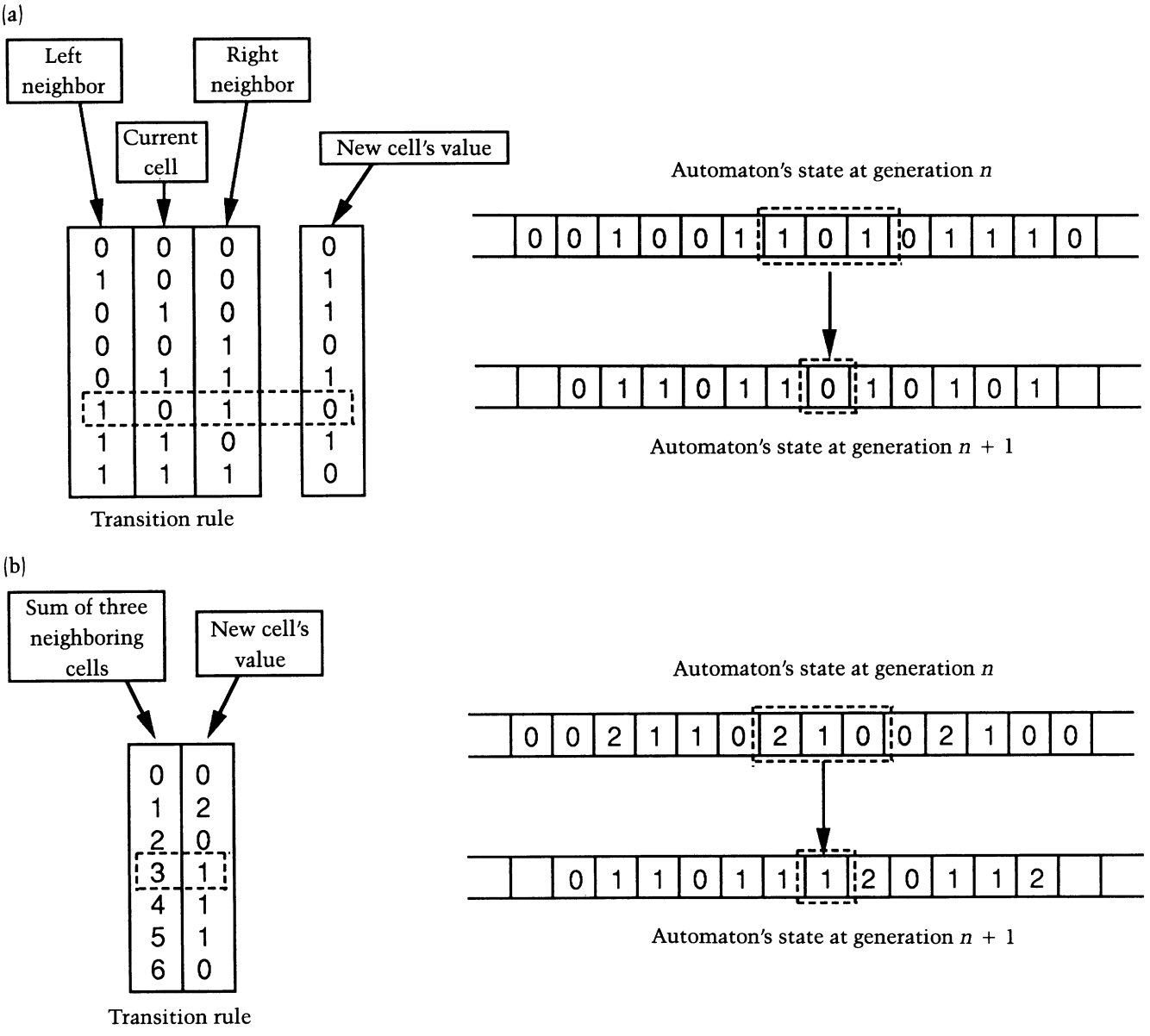
More generally, a cellular automaton may be seen as a grid of cells that represent a particular state at a given moment. At the tick of an imaginary clock, every cell computes a new state according to its previous state and the states of the cells in its neighborhood. The transition rules that determine the new state and the initial state of every cell of the grid are the elements needed to calculate the evolution of the automaton, generation after generation.

In the most common examples of cellular automata, the grid is either two-dimensional and rectangular, as in "Life," or one-dimensional and linear. In the latter case, the cells lie on an unbounded line, often replaced by a wrapped-around segment. The number of states is usually a small integer n. (In "Life" there are only two states, life and death.) Every state is codified by an integer from $0-n-1$ and may be represented graphically by painting the cell with one of n colors. The cells that play a role in determining the future state of a given cell form the neighborhood of this cell. Common neighborhoods for linear automata are made of three cells (a cell and the ones touching it), five cells (a cell and the two nearest cells on each side) and more generally of the $2n + 1$ cells centered on the current cell. Naturally, one may choose more exotic neighborhoods, such as asymmetrical ones, or neighborhoods that exclude the current cell.

The largest freedom is allowed for establishing the transition rule. One may build a transition matrix by assigning a particular state to every possible combination of the states of the cells forming the neighborhood. With n states and p cells in the neighborhood, we need a table of n^p elements to characterize the transition rule. This method is acceptable only when n and p are very small (see Fig. 1a). It is

Computer Music Journal, Vol. 14, No. 4, Winter 1990,

Fig. 1. Examples of linear automata. An automaton with two states and a "three-cells" neighborhood. The transition rule is defined by assigning a new state to every possible combination of states in the three neighboring cells (a). A "sum-of-three-neighboring-cells" automaton with three states (b).

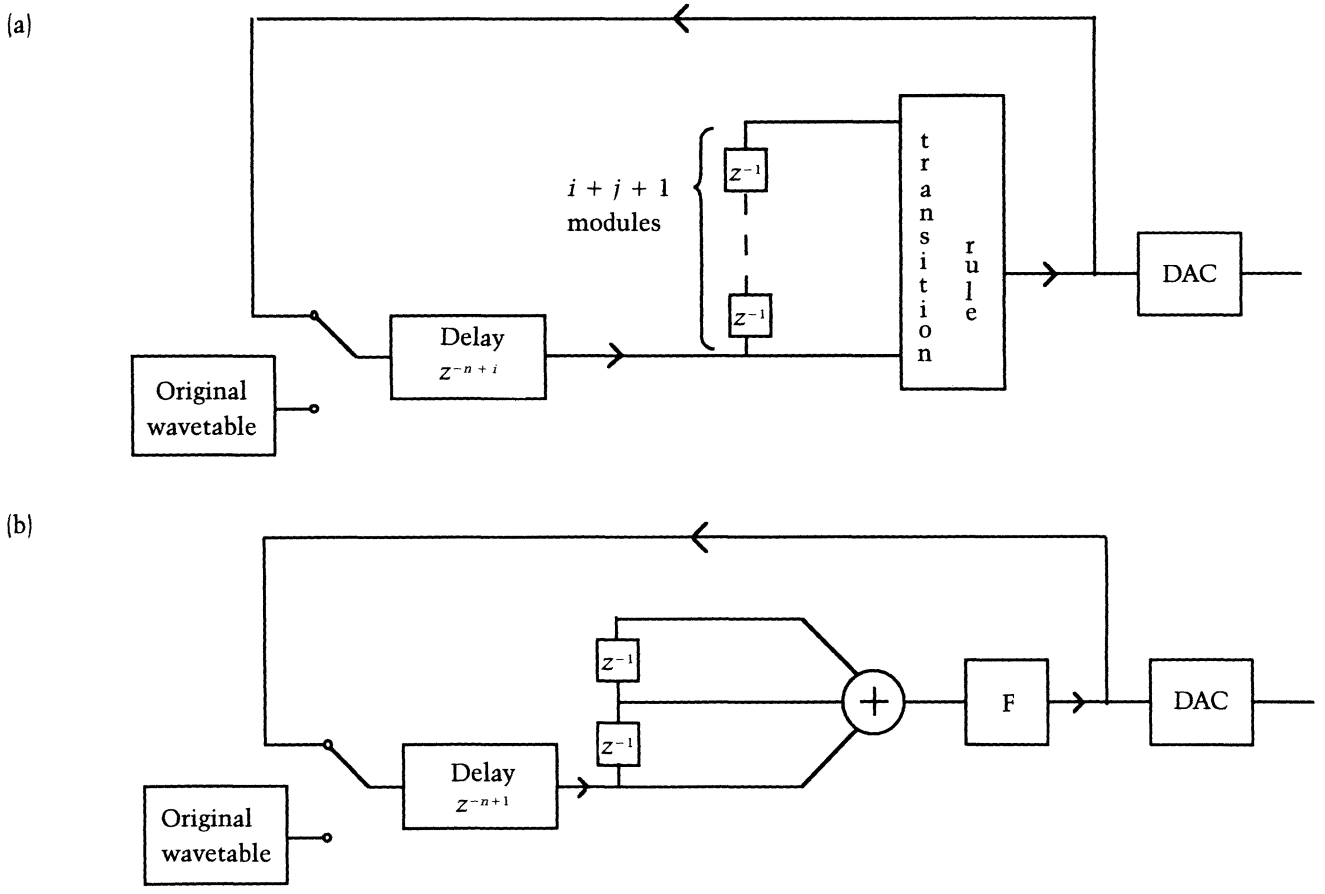


often better to build a smaller table and use some kind of computational algorithm to establish the new state of a cell. In the most common of these algorithms, the sum of the states of the cells of the neighborhood determines the new state of the current cell. With this kind of transition rule, a table of only $p(n-1)+1$ elements is sufficient to allow the computation of the new generations (as shown in Fig. 1b).

The Synthesis Algorithm

Graphical representations of cellular automata have been used early on. Musical translations of automata seem particularly appropriate, since an automaton, like a music surface, is a temporal process. However, there is no intuitive correspondence between the components of an automaton and the elements of a musical process. Some attempts to

Fig. 2. The LASy algorithm seen as a digital filter. The general algorithm. i *and* j *are the numbers of cells in the neighborhood on the left and on the right of the current cell (a). The "sum-of-three-neighboring-cells" algorithm.* F *is the function which defines the transition rule (see text) (b).*



build such mapping have been made recently: e.g., Edgar and Ryan (1986) and Battista and Giri (1988). In both of these cases, an algorithm translates a configuration of states of the automaton in one or more MIDI note commands to be sent to a sound generator.

The algorithm presented in this article, linear automata synthesis (LASy), works at the signal level, not at the note level (Chareyron 1988a; 1988b). A wavetable stored in memory is seen as a linear automaton: i.e., every element (sample) of the wavetable is a cell of the automaton. The value of an element of the waveform is the state of the corresponding cell of the automaton. Generation after generation, the state of every cell is computed according to the transition rule of the automaton, and these values are used to feed a Digital-to-Analog Converter (DAC) for sound output. This operating scheme (a self-modifying waveform) presents an analogy to the Karplus-Strong algorithm (Jaffe and Smith 1983; Karplus and Strong 1983). Indeed, we'll see below that the Karplus-Strong algorithm may be seen as a particular case of the synthesis technique presented here.

The LASy process may be seen as the operation of a digital filter over a time-limited signal, as illustrated in Fig. 2. Figure 2a covers the general algorithm, while Fig. 2b corresponds to the particular case of a "sum-of-three-neighboring-cells" transition rule. The basic Karplus-Strong algorithm is obtained by substituting (in Fig. 2b) the sum of three samples and the function F with the average of two samples, and it would be simulated by means of a "sum-of-two-neighboring-cells" automaton.

Elements of deeper analysis of the algorithm are presented in Appendix 1. We will focus now on the

practical realization of the LASy technique. After loading the computer memory with two numerical tables—the first is the numerical representation of the initial waveform and the second is the transition rules—we launch the computing process that calculates the samples and feeds them to the DAC (as illustrated in Fig. 3a). For most common kinds of transition rules, the necessary computing work is small enough to be carried out in real-time by any common microprocessor. For correct operation of the algorithm, the samples are linearly coded in memory as positive integers from zero to the largest value available. If the DAC expects another kind of coding (such as twos-complement signed numbers), the values are converted (by addition of a constant) before the feeding of the converter.

To get a more precise picture, assume we are using one of the popular "sum-of-three-neighboring-cells" transition rules. Pseudo-Pascal code for this algorithm is given in Fig. 4. The computing process is rather straightforward. When the DAC needs a new sample, we sum the contents of the corresponding cell with those of its two neighbors. The result is then used as an index to fetch the new value of the cell from the transition rule table. This value is sent to the DAC and stored in a new wavetable. When we reach the last cell, we exchange the pointers to the old and new wavetables and restart the process (as in Fig. 3b). The operations needed for computing every sample are two additions and a fetch in a lookup table, which are easily accomplished during a sampling period even by a slow microprocessor. The generation of two or three voices is then possible, although the use of a specialized processor—or a very fast one—would be necessary to realize extended polyphony.

For practical purposes, the memory requirements of the LASy algorithm must be taken into consideration since the transition rule table can be large. Its length is equal to the number of possible sums of three sample values: i.e., $3(2^n - 1) + 1$, where n is the definition (number of bits) of the DAC. This means only 766 8-bit words for an 8-bit DAC conversion, but 12K words for a 12-bit one and a mere 192K 16-bit words for 16-bit conversion! (Everybody knows that memory prices will drop dramatically within the next three months; we have heard it said 100 times in the last two years!) In the meantime we need some way to save memory if we want to have a large number of transition rules at our disposal and avoid the chore of countless exchanges with mass storage devices: i.e., using virtual memory paging for transition rule tables. An efficient method of allowing real-time computation of the transition rule from a small set of parameters will be examined below. We may also take into consideration general compression systems that allow the storage of the table in a reduced format. For example, we might use an algorithm whereby we are able to carry out 16-bit conversion but can store the table in a compressed form corresponding to a 12-bit DAC (a size reduction factor of 16). The first solution is the linear expansion of the reduced table at the beginning of the sound-generation process. Another solution consists of carrying out the whole computing process in the reduced version. Samples in the old wavetable are clipped to the lower resolution before the computation, and the new sample is then adapted to the full resolution of the DAC. The benefits of the greater resolution are not lost if we store the clipped part of old samples and use it to adjust the new sample value by interpolation.

Aside from the practical problems, the size of the transition rule table is an indication of the potential of the LASy synthesis technique. If we decide to limit ourselves to a "sum-of-three-neighboring-cells" transition rule, a 12-bit DAC, and a 256 word wavetable, we may choose among $(2^{12})^{256} * (2^{12})^{(3*2^{12})}$, or about 10^{4500} different automata! Naturally choosing one of these automata randomly will probably lead to rather uninteresting results. Useable automata are only a tiny part of the whole set, but their number remains far beyond comprehension.

Searching for Automata

The huge number of possible automata is evidence of the great potential of the LASy algorithm. This wealth of possibilities has its dark side, however, since it is difficult to extract the ones capable of

Fig. 3. Implementation of the LASy algorithm. General case (a) and "sum-of-three-neighboring-cells" case (b).

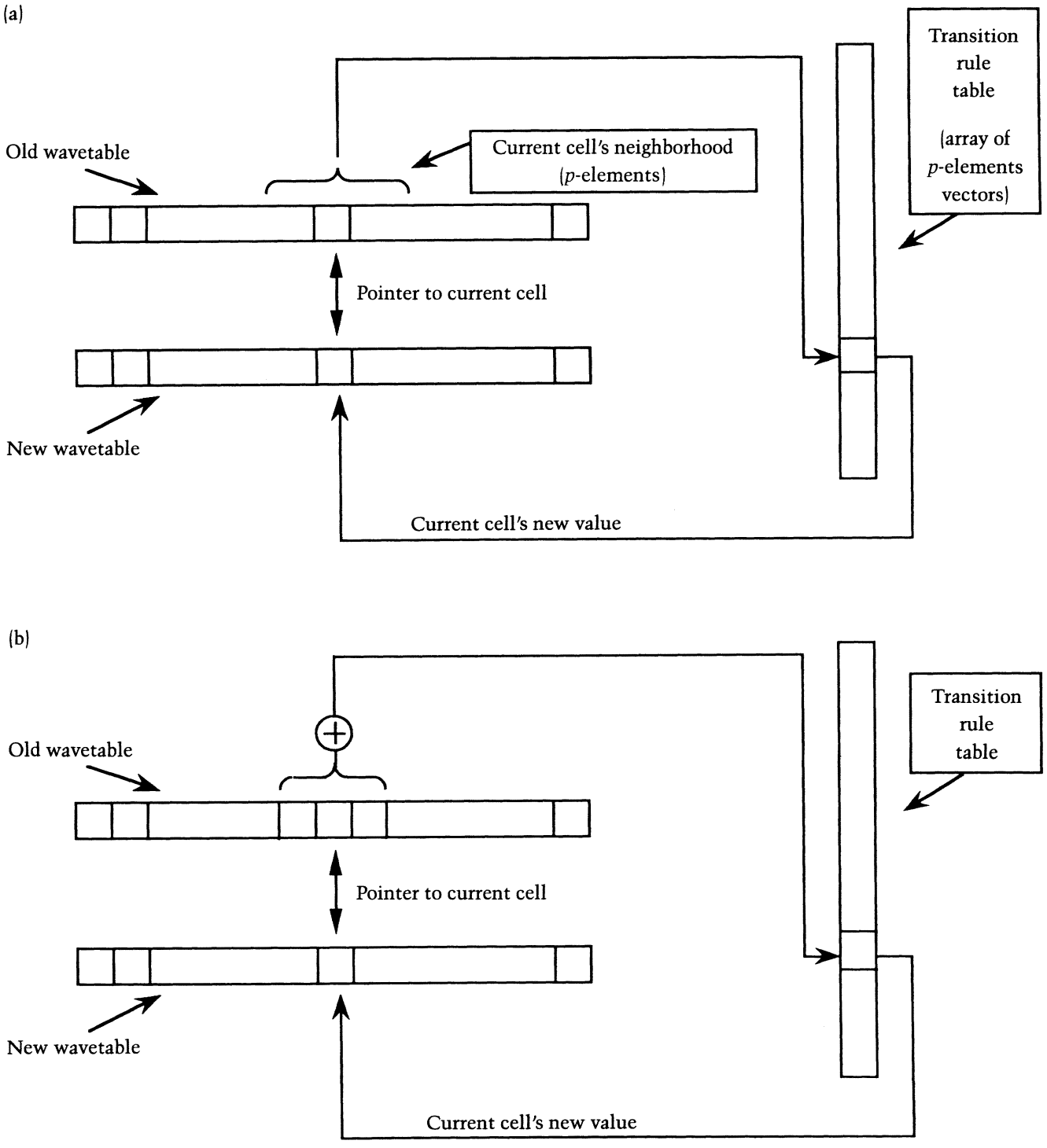


Fig. 4. The pseudo-Pascal version of the basic algorithm for the computation of new samples (case of a "sum-of-three-neighboring-cells" automaton).

```
CONST
  wavelength = . . . {length of the wavetable}
  resolution = . . . {number of bits of the DAC}
  maxsamp = 2^resolution - 1; {maximum value of a sample}
TYPE
  sample = 0..maxsamp;
VAR
  oldwavetable, newwavetable : ARRAY [1..wavelength] OF sample;
  trtable : ARRAY [0..3*maxsamp] OF sample;

PROCEDURE lasy;
    { Carries out the computing of the new state of the automaton,
      i.e. the new elements of the wavetable to feed the DAC }
  WHILE playing
{ The first and last elements of the wavetable need special
  processing }
  sum := newwavetable[wavelength] + oldwavetable[1] +
  oldwavetable[2]; newwavetable[1] := trtable[sum];
  FOR i:=2 TO wavelength-1
    sum := oldwavetable[i-1] + oldwavetable[i] + oldwavetable[i+1];
    newwavetable[i] := trtable[sum];
  ENDFOR
  sum := oldwavetable[wavelength-1] + oldwavetable[wavelength] +
  newwavetable[1];
  newwavetable[wavelength] := trtable[sum];
  swapwave (oldwavetable, newwavetable); { call a procedure which
                                           exchanges t pointers to
                                           the two wavetables }
    checkplay; { update the boolean "playing" }
  ENDWHILE
END; (lasy)
```

producing useful results. Let us examine some basic guidelines for undertaking this task.

An automaton is characterized by two elements, its initial state (the initial waveform) and a transition rule. Of these two elements, the first one is much easier to manage. It is not a problem to keep a large collection of waveforms in memory since each one needs a small amount of space (256 or 512Kbytes seem to suffice). This set should include the classical families of waveforms and some particularly adapted to our synthesis technique. For example, it is a good idea to have some waveforms whose elements are mostly of zero or near-zero value. Combining such waveforms with a "growing" transition rule, we can obtain sounds that begin with a smooth attack stage. The set of waveforms can also include excerpts of sampled sounds if the simulation of natural events is of interest. Special waveforms can be stored on disk and loaded when needed. In musical applications it may be a good idea to add small random modifications to the wavetables to allow the repetition of similar but not identical tones.

Managing the transition rules is a much more involved task. Knowing its previous state and those of its neighbors as a transition rule, we may use any

type of algorithm that allows us to assign a new value to the cell. In theory we may list all the possible combinations of three (or five, or more) neighboring states and assign to each the new value we like, but we have already seen that this process would be painful and the resulting table prohibitively huge. It is better for us to use a transition rule that combines the states of the neighboring cells by means of some arithmetic or logical operation and use the result to read the new state in a previously built table. We have seen that the more popular transition rules for linear automata assign a new value to a cell according to the sum of the three or more neighboring cells centered on the current cell. This is the reason why most of the experimentation using LASy has been done using such automata; the following discussion will focus on this particular family. In my opinion, this type of automaton is not the best suited for sound synthesis. Although useful results have already been obtained with such automata, it is likely that the full potential of the LASy algorithm will only be uncovered using more exotic transition rules. The following notes have general value as the problems we discuss occur in every type of automata.

Sum-of-Three-Neighboring-Cells Automata

Let us examine the process of building the transition rule table. The elements of this table may be seen as the values $F(X)$ of a function F. The variable X varies from $0-3*(2^n - 1)$, where n is the definition (number of bits) used in the computational process. The function F returns an integer from $0-2^n - 1$. To make things clearer, let us see what these numbers mean in the case of a 12-bit definition. Samples may take a value from 0–4095. The mean value 2048 corresponds to the DAC output level zero. Summing the values of three elements of the wavetable, we obtain a result in the 0–12,285 range and must assign a new integer value from 0–4095 to each of these results. Our transition rule will then be codified with a table of 12,286 integers in the range [0, 4095]. This table may be seen as the vector of elements $F(X)$, images of the integers from 0–12,285 by some function F.

Every table of values in the correct range is a reasonable candidate, but in most cases we choose a function whose values may be expressed as an arithmetic and/or logical expression of the variable X. I have found it very convenient to build families of functions that may be computed from a general expression, assigning different values to a limited number of parameters. This approach presents several advantages. For example, the construction of new sounds is greatly facilitated. The user only needs to set the values of the parameters of the function instead of filling out the transition rule table. Because theoretical results about the evolution of the sound are not available for most of the automata, this evolution may only be predicted using empirical methods. If the transition rule is characterized by a limited number of parameters, it will be possible to establish relationships between the variation of every parameter and the evolution of the harmonic spectrum of the sound output. The results of such a study will allow better control over the results of the synthesis. The use of such functions also solves the problem of the space necessary to store the large, lookup tables. The transition rule may be stored as a tiny table containing only the values of the parameters, expecting to be expanded "on the fly" in the corresponding lookup table. We may keep a large family of rules in memory and experiment with combinations of them using the available waveforms.

This approach presents the drawback of limiting the field of automata by eliminating odd transition rules, so the best solution seems to be a mixed one. During the first stage, we fix the parameters of one of the built-in functions to construct the table. Then, we handcraft the final table by modifying some of its values at will. This approach maintains most of the advantages of the automatic method while allowing greater liberty of action. The transition rule will be stored as a table with two parts: one for the parameters and the other for the modifications.

A special class of transition rules, called *symmetrical rules*, may be introduced as a further step towards the simplification of the process of building transition rules. A symmetrical transition rule leads to a similar evolution of the elements of a

wavetable symmetrical with respect to the zero level, as is the case with most natural sounds. Mathematically speaking, the corresponding function F would be an odd one if the wavetable were coded with signed values centered around zero. In this case, one half of the table would be enough to characterize the transition rule. The function could be computed for the positive values of the variable (assuming the zero level were effectively coded with a zero value), and a conversion algorithm could be used to expand the table and restore the needed type of coding. (See the function *expand* in Fig. 5.)

There are many possible forms for the parameterized function used to build our table. Figure 5 shows the algorithms for some example functions in Pascal-like form. The simplest family of functions takes the form

$$F(x) = a(x/3) + b,$$

with two parameters a and b (the function f and procedure *makerule* of Fig. 5). The value of a will be kept generally in the neighborhood of 1 to avoid abrupt changes of the value of a single cell. With a symmetrical rule, a value of $a > 1$ and/or a value of b positive will lead to sounds whose amplitude and spectral complexity grow with time; other values of these parameters insure the opposite behavior. When $b = 0$ and $a < 1$, we obtain transition rules whose actions may be seen as a linear filter operating upon the original wavetable (see Appendix 1 for a more accurate study of this case). When $a = 1$ and $b = 0$, the resulting automaton simulates a variation of the Karplus-Strong algorithm.

To get more complex results, we may replace $x/3$ in the previous equation by $G(x/3)$, where G is a given function (the function g of Fig. 5). Typically, G will be of polynomial or trigonometric form. A larger variety of sounds can be obtained by dividing the range of the variable's value into subranges and applying a function with a unique set of parameters upon each interval (the procedure *makerule1* of Fig. 5).

The next level of complexity can be achieved by choosing two (or more) functions inside every subrange of x-values. For example we may apply one function for the even values of x and a second one for its odd values (function *ff* of Fig. 5). Naturally the distribution of the two functions may follow a more complex proportion or even obey some statistical or stochastic scheme. Interesting results have been obtained by combining this construction method with elementary properties of modulus arithmetic. By choosing the values of such functions carefully, it is possible to build never-ending sounds that combine the growing and decreasing properties presented above in various modes. This discussion can only present a point of departure, since many variations and refinements may be added to this basic scheme. It is amazing to see that even simple functions are able to give a large variety of results by combining small variations of the parameters with a good choice for the initial waveform.

Other Automata

This is a wide-open area with many exciting paths to explore. One possibility lies in the modification of the "sum-of-three-neighboring-cells" rule class. We may modify the neighborhood (including five or more cells), or we may scale the cells of the neighborhood by some weight before summing them. A 1-2-1 weighting scheme is an interesting variation of the basic algorithm; an asymmetric weighting (such as 2-1-1) leads to a less classic evolution. To take this a step further, we may replace the addition with another operation or combination of operations that don't need to be arithmetical. Logical operations, for instance, are able to help the creation of some unusual sounds. A bit of conditional testing may make the results of the algorithm more interesting. We may test the parity of the cell's value or control whether this value is greater or lesser than its left neighbor's and branch to a different operation according to the result. These are only preliminary excursions; I have tried different kinds of nonclassical transition rules, but the experiments have not been pursued with enough depth to tell what are the most promising paths to follow.

Fig. 5. Some functions and procedures (in pseudo-Pascal) needed to build the transition rule table.

```
CONST
  resolution = 16; {DAC's resolution; put the right value here}

PROCEDURE init;
{ Sets some global variables }

  numstates := 2^resolution;
    {number of states of the automaton}
  maxsamp := numstates - 1;
    {max value of a cell}
  maxsymsamp := numstates/2 - 1;
    {max value of cell for a symmetrical rule}
  trtablelength := 3*maxsamp + 1;
    {length of the transition rule table}
  symtrtablelength := 3*maxsymsamp + 1;
    {idem for a symmetrical transition rule}

END; (init)

FUNCTION clip (x: INTEGER);
{ This function clips the computed values for the new samples to the
  allowed range. The code presents the most straightforward way to
  deal with out-of-bounds values ; there are other solutions leading
  to interesting results. }

  clip := x;
  IF clip>maxsamp THEN clip := maxsamp;
  IF clip<0 THEN clip := 0;

END; (clip)

FUNCTION expsym (tabsym, tab: array of integer);
{ Expands a symmetrical transition rule table to the general
  transition rule structure. }

  tabcenter := 3*numstates/2;
  meanvalue := numstates/2;
  FOR i:=0 TO symtrtablelength
    tab[tabcenter+i] := meanvalue + tabsym[i];
    tab[tabcenter-i] := meanvalue - tabsym[i];
  ENDFOR

END; (expsym)
```

(cont'd)

```
FUNCTION f (a: real; b,i: INTEGER);
{ The simplest building function }

 f := a*(i/3) + b;

END; (f)

FUNCTION g (a,c,d: REAL; b,i: INTEGER);
{ An example of a little more complex function }

 g := a*( i/3 + c*sin(d*i) ) + b;

END; (g)

PROCEDURE makerule (a: real; b: INTEGER; sym: BOOLEAN);
{ The basic algorithm to build a transition rule }

 init;
 IF NOT(sym) THEN
    FOR i:=0 TO 3*maxsamp
      newsamp := f(a, b, i);
        { Instead of f you may use any function you have defined }
      tab[i] := clip(newsamp);
    ENDFOR;

  ELSE
    FOR i := 0 TO 3*maxsymsamp
      newsamp := f(a, b, i);
      tabsym[i] := clipsym(newsamp);
    ENDFOR;
    expand(tabsym, tab);

  ENDIF

END; (makerule)

PROCEDURE makerule1 (a: real; b: INTEGER; sym: BOOLEAN);

{makerule1 computes the automaton transition rule by using different
 functions f0, f1, . . . on every subrange of values of the variable
 i. The following code has only a demonstrative value. In a real
 application, the only parameter passed to the procedure would
 probably be a pointer to a table including all the values needed to
 construct the transition rule. The structure of this table will
 determine the organization of the procedure's code. }
```

(cont'd)

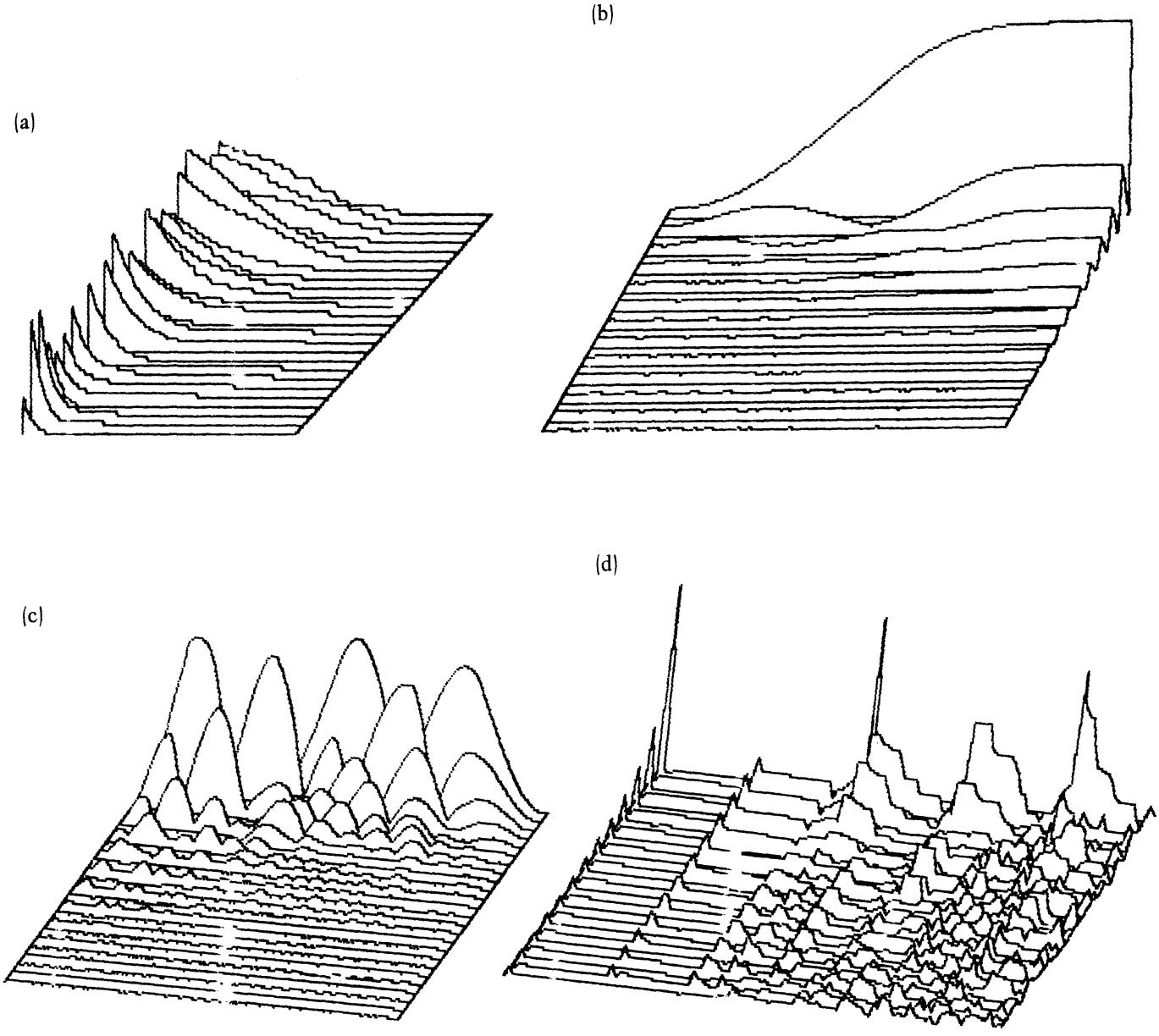
```
  init;
  FOR i:=0 TO bound1
    tab[i] := clip( f0 ( . . . , i) );
  ENDFOR;
  FOR i := bound1+1 to bound2
    tab[i] := clip( f1 ( . . . , i) );
  ENDFOR;
    (. . .)
  FOR i:= boundn+1 to 3*maxsamp
    tab[i] := clip( fn( . . . , i) );
  ENDFOR;

{ To spare space, the procedure doesn't include the code for a
  symmetric rule, which will be easily deduced from the generic
  code. }

END; (makerule1)

FUNCTION ff (a1,a2,c2,d2: REAL; b1,b2: INTEGER);
{ A simple way of building a transition rule which acts in different
  way according the parity of the sum of the cells }

  IF i:= 2*int(i/2)     {true when i is even}
  THEN ff := f(a1,b1,i);
  ELSE ff := g(a2,b2,c2,d2,i);  {or any two functions you like}
  ENDIF;

END; (ff)
```

Musical Results

How do automata really sound? It is difficult to give anything but a very partial answer at this early stage of experimentation, but the first results are rather interesting. Figure 6 shows a three-dimensional, spectral, surface diagram of the harmonic evolution for some example sounds. The LASy algorithm has shown its ability to generate a large range of sounds with diverse timbral evolutions. In particular, this algorithm is well suited for the generation of fast transients (especially in the attack stage of the sound), an area where traditional synthesis techniques show some weakness. This leads us to expect this technique to be valuable for the simulation of acoustical instruments. No special work has been done in this direction, but some of the sounds generated to date may already fit as crude approximations of wind and stringed instruments (and, of course, good imitations of plucked strings).

Let us try to establish an embryonic classification of the generated tones. One family includes the sounds with the simplest behavior. For example, the use of function *f* of Fig. 5 in a symmetrical transition rule will result in a monotonous evolution of the harmonic content of the sound. The number and the amplitude of the partials will follow either an increasing or a decreasing curve. In both cases, the evolution leads to a steady-state

Fig. 6. Evolution of the amplitude of the 25 first harmonics for some automaton-generated tones. Time goes from left to right, the order of harmonics from back to front, and the vertical axis shows the amplitude. Due to the use of 8-bit resolution and to some rounding errors, the graphics are somewhat approximate. Karplus-Strong algorithm simulation with a "sum-of-three-neighboring-cells" automaton (a). Use of a "growing" transition rule. The sound begins with a smooth attack and a timbre with only a few significant sinusoidal components. It reaches a stable state corresponding to a square wave (b). In this example, the harmonic evolution is less elementary. It is worth noting that the sound presents its proper and rather classical envelope (c). A never-ending and complex sound. Due to the relations between harmonic components, the result sounds less noisy than it appears on the graphics (d).



situation (silence or harmonic-rich sound) after a finite number of generations (examples of such simple automata are the ones that simulate the Karplus-Strong algorithm and its variations). Increasing the complexity level of the transition rule, we obtain sounds that present a never-ending but still simple evolution (in one common case, the evolution will be semicyclical and presents a succession of similar but not identical phases). In the last category are automata that exhibit totally unpredictable behavior and generate complex, ever-changing (and often quite strange) sounds.

This classification should be related to results obtained from the study of cellular automata. Automata have been grouped in different classes according to the level of complexity of their evolution. It has been established that automata of the last class are computationally irreducible. What this means is that there is no shortcut to compute the future state of the automaton directly. The only way of knowing what an automaton will generate at a given time is to launch it and wait while all intermediate generations are computed.

The musical implications of such properties appear worthy of investigation. Present synthesis techniques are all computationally reducible. For example, mathematical rules allow us to calculate the spectral composition of a frequency-modulation sound at any instant, given its parameters. This characteristic of conventional synthesis techniques is a great advantage since it gives the composer complete control over the results. On the other hand, the sounds become easily recognizable and foreseeable, and it may be argued that they proclaim too loudly their mathematical origin and lack the degree of unpredictability the human mind expects from an artistic process. Of course, everybody has a personal opinion on the subject and may or may not agree with such views. Leaving ethical or aesthetical choices out of the discussion, the availability of a different, unpredictable synthesis algorithm should be taken into consideration as it presents a great opportunity for exploring new horizons. One positive feature of the LASy technique lies in the availability within a unique algorithm of both predictable and unpredictable sounds, allowing great liberty of experimentation within the same working environment.

Musical Applications

A large family of automata (such as the never-ending and unpredictable ones) generates sounds with a rich harmonic structure in continuous evolution. These complex sound textures could be used as raw material to be worked upon in a musical composition. Other automata are best suited to "note-oriented" music. Some of the sounds generated by LASy show the typical Attack–Decay–Sustain–Release (ADSR)-like evolution of a musical event and may be used "as is" without further envelope control. Other sounds are more specialized; the components of the spectrum follow either an increasing or a decreasing curve. In such cases, we may use two (or more) transition rules in succession. A first transition rule operates upon the initial waveform after a note-on signal, and when the corresponding note-off signal is acknowledged, a new transition rule is substituted "on the fly" for the first one. There are many possible variations of this simple scheme, leading to a more flexible and programmable instrument. A real automaton-oriented instrument would provide other facilities, such as the inclusion of real-time controllers and the introduction of a programmable level of randomness in the elements of the synthesis algorithm. The possibility of modifying the tables that determine the sound in real-time gives a large range of ways to control the musical result.

Digital Signal Processing using LASy

The LASy algorithm can easily be modified to allow digital signal processing (DSP) of some input sample stream. For example, let n be the width of the neighborhood of the chosen automaton. We get the last n samples of the input signal, apply the automaton transition rule to them, and feed the DAC with the result (or repeat the process). We have not yet carried out such an experiment, but it seems like an interesting path.

Future Work

The first steps in the exploration of linear automata synthesis have been made rather randomly, without precise guidelines. Special tools are needed to allow the pursuit of the research in a more rational way, so the next step will be the implementation of a dedicated working environment, allowing both scientific experimentation and practical creation of timbres. This LASy workstation should provide the integration of both a sound-generation module and an automaton-creation module, allowing for immediate aural feedback. Additional modules should provide useful features, such as real-time, graphical display of the output waveform and of the spectrum of the generated sound. The building tools should allow several levels of access to automata creation. The entry level should be graphics-oriented to allow musicians to create their own sounds without any familiarity with automata theory and numbers in general. It is important to keep the structure open, so as to allow easy integration of new types of automata and different building algorithms for the transition rules.

Currently, I am implementing such an automaton workstation at the L.I.M. (Laboratorio d'Informatica Musicale) in Milan, Italy. An Apple Macintosh equipped with a Digidesign Sound Accelerator board that includes a 16-bit DAC and a Motorola 56001 DSP chip is being used. The DSP is only used for ancillary tasks at the moment (such as real-time fast Fourier transforms), since the Macintosh microprocessor is more than fast enough to keep up with monophonic, real-time synthesis and the associated graphical display.

Conclusion

Composers and researchers will need to make some changes in their working habits when working with the LASy algorithm. The first steps are disorientating because intuition or mathematical methods cannot be relied upon. Using the right tools, it is possible to acquire sufficient familiarity with the algorithm to control it in a proficient way. From personal experience, I would say that roughly 50 percent of the new automata I build generate sounds with the characteristics I expected from them, while about 10 percent of the automata give unexpected but interesting results and are often the point of departure from which new paths may be explored. (The other ones should only be forgotten.) Such nice surprises are the kinds of rewards possible in the exploration of the new world of LASy-generated sounds.

Acknowledgments

I am truly grateful to the staff of the L.I.M., and especially to Goffredo Haus, for their support. I would also like to thank James Dashow for his advice and encouragement. This work has been realized with the financial support of the C.N.R. (the Italian National Research Council), in the framework of the Music Goal, Sub-project 7 "Sistemi di Supporto al Lavoro Intellettuale," Finalized Project II "Sistemi Informatici e Calcolo Parallelo."

References

Battista, T., and M. Giri. 1988. "Composizione Tramite Automi Cellulari." *Atti del VII Colloquio di Informatica Musicale.* Rome, Italy: Edizioni Arti Grafiche Ambrosini, pp. 181–182.

Burks, A., ed. 1970. *Essays on Cellular Automata.* Champaign/Urbana, Illinois: University of Illinois Press.

Chareyron, J. 1988a. "Sound Synthesis and Processing by Means of Linear Cellular Automata." *Proceedings of the 1988 International Computer Music Conference.* San Francisco: Computer Music Association.

Chareyron, J. 1988b. "Wavetable come Automa Cellulare: una Nuova Tecnica di Sintesi." *Atti del VII Colloquio di Informatica Musicale.* Rome, Italy: Edizioni Arti Grafiche Ambrosini, pp. 174–177.

Edgar, R., and J. Ryan. 1986. "LINA" *Exhibition of the 1986 International Computer Music Conference.* San Francisco: Computer Music Association.

Farmer, D., T. Toffoli, and S. Wolfram, eds. 1984. *Cellular Automata.* North-Holland Physics Publishing.

Gardner, M. 1970. "The Fantastic Combinations of John Conway's New Solitaire Game 'Life.'" *Scientific American* 223(4): 120–123.

Jaffe, D., and J. Smith. 1983. "Extensions of the Karplus-Strong Plucked-String Algorithm." *Computer Music Journal* 7(2): 56–69. Reprinted in C. Roads, ed. 1989. *The Music Machine*. Cambridge, Massachusetts: MIT Press. pp. 467–480.

Karplus, K., and A. Strong. 1983. "Digital Synthesis of Plucked-String and Drum Timbres." *Computer Music Journal* 7(2): 43–55. Reprinted in C. Roads, ed. 1989. *The Music Machine*. Cambridge, Massachusetts: MIT Press. pp. 481–494.

Appendix 1: Analysis of the LASy Synthesis Algorithm

Assume the chosen automaton is of the classical type in which the future state of a cell is determined by the sum of its current state and those of its two neighbors. Since the automaton is a digital filter that operates upon the initial wavetable, we may write the following difference equation:

$$y(n) = x(n) + f(y(n - p - 1) + y(n - p) + y(n - p + 1)), \quad (1)$$

where p is the length of the wavetable; $x(n) = 0$ for $n < 0$ and $n = p$ and the values of $x(n)$ are the initial numbers with which we have fed the wavetable for $0 = n < p$; and f is the function corresponding to the transition rule. The $3*2^b$-5 values of the function f are stored in the lookup table used to compute the new state of the automaton.

To be able to carry on the analysis with the usual tools of DSP, it is necessary for the function f to be linear: i.e.,

$$f(x) = ax,$$

where a is a real with a modulus $< 1/3$ (this last condition is necessary to obtain a result compatible with the resolution of the DAC).

It is worth noting that these functions represent only an infinitesimal part of the whole world of possible automata. (Yes, linear automata make generally nonlinear filters.) Since in these particular cases the results of the spectral evolution are simple and the generated sounds musically interesting, let us go on with the analysis.

The difference equation corresponding to this family of automata is now

$$y(n) = x(n) + ay(n - p - 1) + ay(n - p) + ay(n - p + 1) \quad (2)$$

This synthesis algorithm may be seen as a variation of the basic Karplus-Strong algorithm. When $a = 1/3$, the two techniques combine a delay loop with an average calculation to compute the next sample. There are two differences—the Karplus-Strong algorithm uses a two-value average and a p-length delay when the LASy algorithm uses a three-value algorithm and a $p - 1$ length delay. Let us go on and write the transfer function corresponding to Eq. (2).

$$H(z) = \frac{1}{1 - az^{-p-1} - az^{-p} - az^{-p+1}}$$

$$= \frac{1}{1 - az^{-p+1}(1 + z^{-1} + z^{-2})}$$

$$= \frac{1}{1 - H_1(z) \cdot H_2(z)}$$

Let us examine in more detail the spectral evolution of the generated signal. It makes things easier to split the process into its two elements—delay and three points average (with further product). The frequency response of the delay is:

$$H_1(e^{j\omega T}) = e^{-j(p-1)\omega T},$$

where T is the sampling period. The corresponding gain is:

$$G_1(\omega) = |e^{-j(p-1)\omega T}| = 1.$$

For the average-like part, things are a little bit more complex. The frequency response is given by:

$$H_2(e^{j\omega T}) = a(1 + e^{-j\omega T} + e^{-2j\omega T})$$

$$= ae^{-j\omega T}(e^{j\omega T} + 1 + e^{-j\omega T})$$

$$= ae^{-j\omega T}(2\cos \omega T + 1)$$

and the gain by:

$$G_2(\omega) = |H_2(e^{j\omega T})|$$

$$= |ae^{-j\omega T}(2\cos \omega T + 1)|$$

$$= |a| \cdot |2\cos \omega T + 1|.$$

Since the modulus of a is $< 1/3$, the resulting gain is always < 1. The minor loss corresponds to the fundamental frequency, and the greater loss is seen when

$$\begin{aligned} 2\cos \omega T + 1 &= 0 \Leftrightarrow \cos \omega T \\ &= -1/2 \Leftrightarrow \omega T = 2\pi/3 \\ &\quad \text{(because } 0 \leq \omega T \leq \pi\text{)} \\ 2\pi fT &= 2\pi/3 \Leftrightarrow fT \\ &= 1/3 \Leftrightarrow f = 1/3T \end{aligned}$$

i.e., for a frequency which is the third of the sampling frequency.

These results are less straightforward than those of the Karplus-Strong algorithm, but a common scheme may be identified. As new generations are computed there is a decay of the harmonics that is greater for the high-numbered ones than for those near to the fundamental. An aural test confirms this likeness: starting with a random wavetable, the generated sound is a nice plucked string simulation.

Now let us look at the phase delay (in samples) of the two components of the filter. For the delay we have:

$$\begin{aligned} P_1 &= -\frac{\arg(H_1(e^{j\omega T})}{\omega T} = \frac{\arg(e^{-j(p-1)\omega T})}{\omega T} \\ &= -\frac{(1-p)\omega T}{\omega T} = \mathrm{p} - 1 \end{aligned}$$

where $\arg(z)$ is the argument, or angle, of the complex number z. For the average-like part, we have:

$$\begin{aligned} P_2 &= -\frac{\arg(H_2(e^{j\omega T})|)}{\omega T} \\ &= -\frac{\arg\,[ae^{-j\omega T}(2\cos \omega T + 1)]}{\omega T} \\ &= -\frac{(-\omega T)}{\omega T} = 1. \end{aligned}$$

The total loop length is p samples, which corresponds to a "period" of pT. There is no real period in the generated tone, since the wavetable is modified at every generation. But if the changes are smooth (and they are for the musically interesting automata), the signal is heard as a pitched sound. The corresponding frequency is $1/pT = f_s/p$ (where f_s is the sampling frequency), a rather predictable result since this is the frequency of a periodic signal generated from a fixed wavetable of the same length.

Note that things are a little more complex in the original form of the Karplus-Strong algorithm, where the total phase delay is $p + 1/2$ samples. This derives from the use of the two-sample average, which may be simulated using an automaton with an asymmetrical neighborhood (the cell and its right neighbor). An interesting variation of the basic Karplus-Strong algorithm may be obtained by giving a 1-2-1 weighting to the three cells of the neighborhood. Note that this version of the algorithm prescribes a very small increase in computing activity (one extra shifting operation per sample) with regard to the "standard" algorithm. The difference equation now becomes:

$$\begin{aligned} y(n) = x(n) + f(&y(n-p-1) \\ &+ 2{*}y(n-p) + y(n-p+1)). \end{aligned}$$

With a particular linear function f:, we get:

$$\begin{aligned} y(n) = x(n) + [&y(n-p-1) \\ &+ 2{*}y(n-p) + y(n-p+1)]/4. \end{aligned}$$

So the transfer function of the average part of the algorithm is

$$H_2(z) = 1 + 2z^{-1} + z^{-2}$$

and the corresponding frequency response is described by

$$\begin{aligned} H_2(e^{j\omega T}) &= 1 + 2e^{-j\omega T} + e^{-2j\omega T} \\ &= e^{-j\omega T}(e^{j\omega T} + 2 + e^{-j\omega T}) \\ &= e^{-j\omega T}(2\cos \omega T + 2) \\ &= 2e^{-j\omega T} \cdot 2\cos^2(\omega T/2) \\ &= 4e^{-j\omega T} \cos^2(\omega T/2) \end{aligned}$$

with a gain given by

$$\begin{aligned} G_2(\omega) &= |H_2(e^{j\omega T})| \\ &= |4e^{-i\omega T}\cos^2(\omega T/2)| \\ &= 4\cos^2(\omega T/2) \\ &= 4\cos^2(\pi f T). \end{aligned}$$

Assuming that f is lesser than the Nyquist limit, $\pi f T$ is bound by 0 and $\pi/2$, and we see that the amplitude loss is now constantly growing with the frequency of the harmonics. There is no change in the phase delay, which is p samples as it would be with any automaton working on a symmetrical neighborhood.

Appendix 2: LASy and Distortion

The action of the transition rule upon an initial waveform may be seen as a form of distortion. This is evident if we consider a trivial type of automaton where the neighborhood of one cell is confined to the cell itself. The two algorithms (nonlinear distortion and LASy) give the same results from the same initial waveform. If f is the distorting function and $iw(n)$ an element of the initial waveform, the corresponding element of the distorted waveform will be

$$dw(n) = f[iw(n)].$$

There are, however, important differences between the two synthesis techniques if we consider a more general automaton. First, in LASy we use the value of the current cell *and those of its neighbors* to compute the corresponding value of the distorted waveform. Second, the LASy algorithm generates an ever-changing waveform, whereas nonlinear distortion leads to a stable waveform (variations in the spectral composition of the sound must be achieved through modification in time of the distorting function). In the trivial example of a "no-neighbors neighborhood" automaton, call f the corresponding distorting function. The waveform at the n^{th} generation is the result of distorting the initial waveform with a function f^n (whereby f^n represents the result of composing the function f with itself n times). In most cases, the distortion tends to add complexity to the harmonic spectrum of the waveform and the repeated composition emphasizes this phenomenon. It seems easy to predict that we will end with brighter and brighter—or more probably noisier and noisier—tones. In real automata, the use of the neighboring cells' values during the computation generally has a "smoothing" effect and leads to the erosion of higher harmonics. This phenomenon is clear in the particular case of the Karplus-Strong algorithm. We must balance these two contradictory forces to generate a sound with the needed characteristics. Although schematic (a large dose of unpredictability is present in the more complex automata), these observations must be kept in mind in practice. The ability to finetune the action of both elements and reach some kind of equilibrium is fundamental to obtain musically usable sounds.