Max and Chaos

Random Numbers

In common usage, random numbers are numbers which cannot be predicted. In mathematical terms, a random series is one in which the probability of any number occurring at any point is 1/n, where n is the number of allowed values. Over a long set of numbers, you would expect to find equal quantities of each value. Random series are fairly common in nature (many noise waveforms are random, for instance) but they are surprisingly difficult to produce on computers.

In Max, random numbers are generated by the random object. In versions before 4.0, the random object would produce the same random number series every time you started Max. This is actually the usual behavior of random number generators. Each number is generated by applying a complex formula to the previous numbers in the series. Naturally it has to start somewhere, with a value called the seed. Any calculation done from the same seed will give the same results.

In version 4, the random object starts calculating with a number based on how long the computer has been on, so random is unpredictable as well as evenly distributed. To return to predictability, the random object accepts the seed message, which forces it to start calculation from the number you give it. The drunk object, which produces numbers that differ from the previous number by a random amount, also accepts seeding. For other ways of shaping the random output, see the essay Random Max.

Monte Carlo

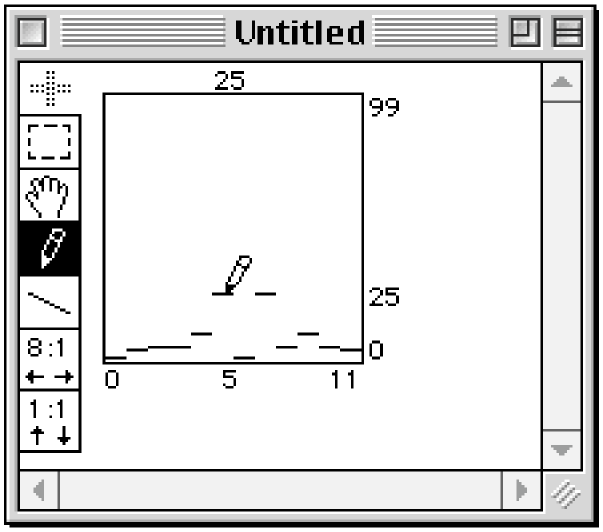
Sometimes you want to play the odds. In cards, we shuffle a deck and then examine it to see what order the cards are. As we get to the end of the deck, we begin to expect to see particular cards, just because we know we won't see the same one twice. This procedure is often applied to music, for example in the generation of tone rows. The urn object (Unique Random Number) is designed for this. Urn produces each possible number once, then bangs out the right. (Urn also can be seeded for predictable output.)

Probability

More likely than wanting each note exactly once, you just want a tendency for certain notes to follow others. Suppose we'd like to see mostly intervals of a fifth, some thirds, and very seldom hear tritones. We can make up a preference chart that looks like this:

interval	Out of 100
7	25
5	25
4	10
9	10
3	5
8	5
2	5
10	5
1	4
11	4
6	1
0	1

This sets up desired probabilities of each interval occurring. We can get values weighted by a probability out of the table object simply by entering the curve we want, and then banging the table. Here's what the table looks like:



When it is banged, the addresses (numbers across the bottom) pop out with the likelihood determined by the number at each address. Here's a sample run:

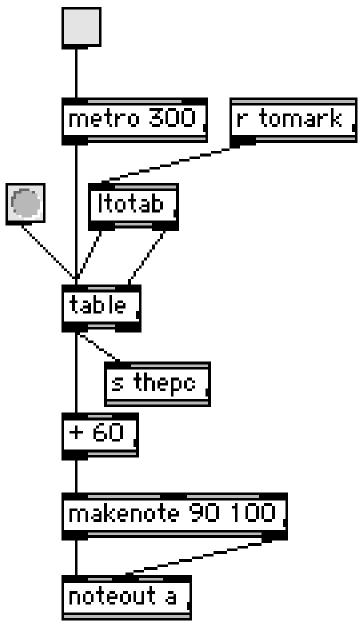
5 7 4 1 7 7 5 4 7 7 2 5 2 7 8 9 8 2 5 10 2 2 4 10 5 2 7 5 5 5 7 5 7 9 4 9 4 5 5 7 2 1 5 7 4 7 5

There are a lot of fives and sevens, and not a single six.

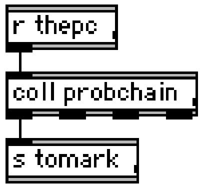
The histo object makes an interesting companion to the table used for probabilities. It keeps track of data coming in, and counts the occurrence of values. This data can be transferred to the table so the proportion of pitches generated sort of matches the proportion of pitches received.

Markov

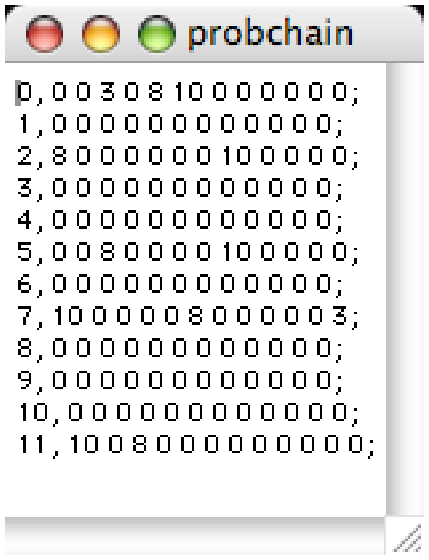
The next step is to make the probabilities depend on the current note. To do that, we can use a coll with 12 different probability tables. When a note is chosen, it pops a new set of probabilities out of the coll and they are loaded into the table. The LtoTab object makes that easy. A system where the most recent value determines the probabilities for the next value is called a Markov chain of the first order. If the last two values determine the probabilities, it's a second order Markov chain. That can be done with pitches, but requires 144 tables. Generally you would use high order Markov processes in operations with fewer choices, such as choosing direction or rhythm. Here's a structure that provides the basic Markov process:



I've used send and receive objects to make it easy to connect a graphic front end. (That front end is presented in User Interface Design in Max). In this instance we'll use a coll:



You can see that when the table provides a number (thepc) the coll will pop out a list the reprograms the table (via ltotab) for the next note. Here's an example of what may be in the table:



```
0, 0 0 3 0 8 10 0 0 0 0 0 0;
1, 0 0 0 0 0 0 0 0 0 0 0 0;
2, 8 0 0 0 0 0 0 10 0 0 0 0;
3, 0 0 0 0 0 0 0 0 0 0 0 0;
4, 0 0 0 0 0 0 0 0 0 0 0 0;
5, 0 0 8 0 0 0 0 10 0 0 0 0;
6, 0 0 0 0 0 0 0 0 0 0 0 0;
7, 10 0 0 0 0 8 0 0 0 0 0 3;
8, 0 0 0 0 0 0 0 0 0 0 0 0;
9, 0 0 0 0 0 0 0 0 0 0 0 0;
10, 0 0 0 0 0 0 0 0 0 0 0 0;
11, 10 0 8 0 0 0 0 0 0 0 0 0;
```

After editing one of these for a while, you will understand why I'm interested in graphical user interface. But with careful attention, you can produce effective markov sequences.

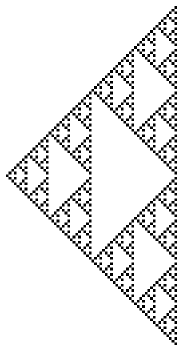
Prob

The prob object also does first order Markov processes. You program it by feeding in a series of lists of type A B P. This sets the chances of B following A as P. (Actually P/T, where T is the total of all Ps you have assigned a particular A. If the prob object happens to get into a state for which there is no programmed choice, it will bang the right outlet.

Second Order Markov process.
The coll and table approach can easily be expanded to second order markov processes. You simply compute the index in the coll from the product of the two previous notes. This does require 144 lines, so some kind of programming machinery is highly recommended.

Fractals

Fractal geometry is the study of objects that have a property known as self-similarity – They are made up of smaller copies of the overall shape. There are many such forms, and although they are visually appealing, it is often difficult to translate their properties into musical context. One that does work is called the Sierpinski triangle:

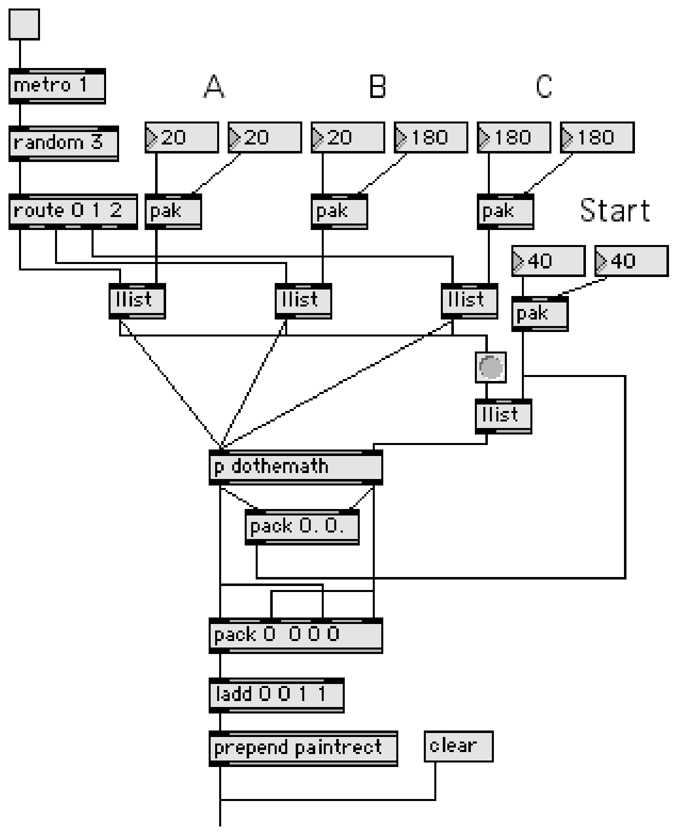


It is traditionally made by starting with a solid triangle and removing smaller and smaller triangles. This is a rather boring musical process, but there's another way to generate them that is interesting.

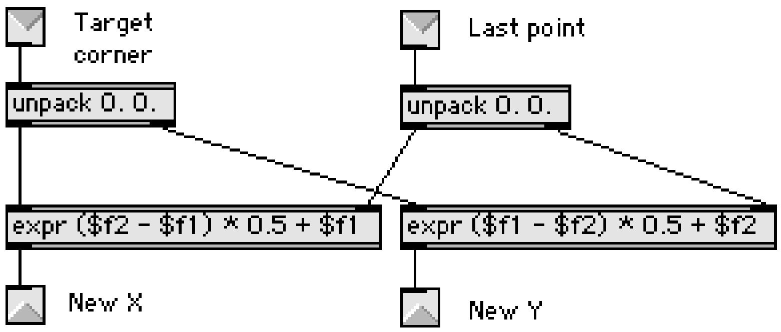
The process goes like this:
1. Plot the three corners of the triangle.
2. Pick an arbitrary starting point (not one of the above)
3. Randomly choose one of the corners.
4. Plot a new point halfway before the previous point and the corner.
5. Repeat steps 3 and 4 a couple of thousand times.

To do this in Max, we'll draw in the LCD object. For every point we plot, we'll draw a square one pixel on a side. The command to do this is paintrect 0 0 1 1 . The numbers represent left edge, top edge, right edge, bottom edge, in that order, measured from the upper left of the space. To put a square at a desired location, we add the horizontal value (X) to left and right, and the vertical value (Y) to the top and bottom. So paintrect 20 20 21 21 will put a square at X = 20, Y = 20.

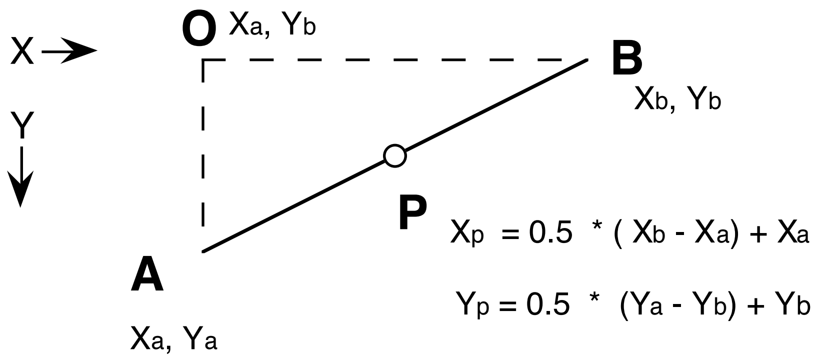
Here's a patcher to do it:



The bottom lines go to the LCD. You can see how the three corners and the starting point are entered. The metro will make the plotting fast enough to watch the process. Here's how new points are calculated:

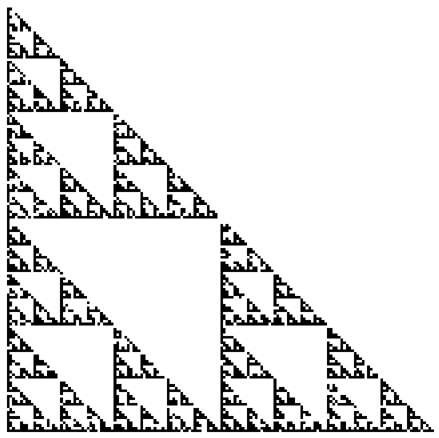


Remember, everything in the LCD is placed by counting pixels from the upper left of the display area. Therefore, a line from A to B can be thought of as the hypotenuse of a right triangle with the right angle (call it point O) at the x of A and the y of B.

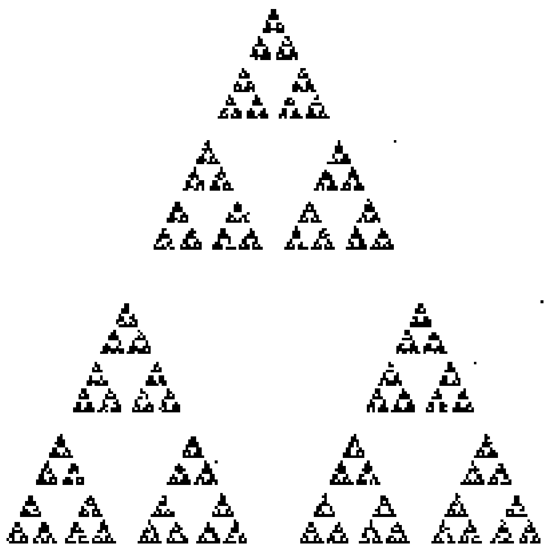


A new point P on the line gives a similar triangle, so the sides will all be in proportion. If P is halfway between A and B, the X value for P can be found with the formula above: half the distance OB, plus the X value at O. Y is similar, but slightly different to get the sign right. If A or B wind up above O or to the left, the math still works because you will have negative distances to add to Xa and Yb. For different proportions, the fraction in Xp becomes the proportion and the fraction in Yp is the compliment - so to find points a quarter of the way along the line, the fractions become 0.25 and 0.75.

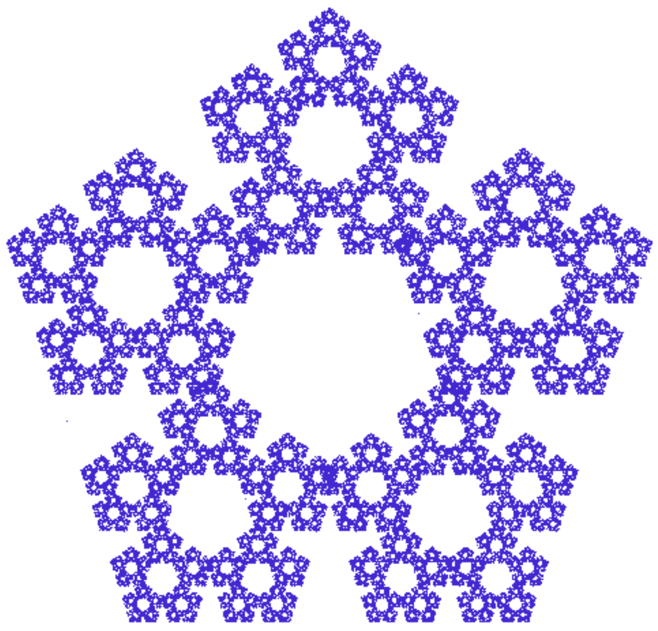
Anyway, with the patcher working right, the figure drawn looks something like this:



What's going on? Well, we are finding points halfway between the last point and random corners of the triangle. So if you start outside the triangle, the points will be inexorably pulled into the triangle, and once inside, can't get out. If you start from a point inside the triangle, you will never land in the central hole, because all points in there would have to be derived from a point outside the triangle. If you look at the process backwards for a moment, you'll see what I mean. If there can be no points in the central hole, well, anywhere halfway between the hole and the corners is excluded as well. You can see where I'm going with this. With other divisions of the line, you get related but different pictures, like this one:



The triangle begins to break apart because the exclusion zone is larger. These figures can also be drawn with more points:



What is the area of something like this? Well, in the ideal version, unlimited by printing resolution, every pentagon is missing a pentagon of 1/6th its area, in an infinitely diminishing series. That means its area approaches zero. It's really a very complicated line. What is the length of the line? Following all infinitely small twists and turns, the length approaches infinity. Since it has no area, it is not a two dimensional object, but it has length, so it is more than one dimension. It's dimension is some fraction between one and two, which is called a <u>fractal</u>.

Sonification

How do we get music out of these things? One approach is to tap into the drawing process. For each point, an x and y value is calculated as it is drawn. We can remap these to notes by adjusting the range of possible values to fit a desired scale. On the pentagon shown, x values could run from 0 to 300. There are several ways to adjust, each with different results. They are all done with an expression:

- Expr $f1/2.36 will convert the x dimension directly to pitches. Some will be very low and some will be very high. What you actually hear will depend on the synthesizer used.
- Expr $f1 / 4.41 + 24 will map the x value from a low C of 24 to a high C at 92. Incidentally, division is a slower process than multiplication, so it is slightly better to do expr $f1 * 0.22667 + 24
- Expr abs($f1 - 150) * 0.453 + 24 will create a mapping symmetrical around the center. This makes sense for this type of figure.

This particular figure is made up of thousands of dots. To make it visually interesting you'd probably want to draw it pretty fast, most likely too fast to actually play. You can slow down the notes by playing every 10th value or so, generating rhythm somewhere else and playing the most recent x, or using the y values and an if then to select notes to play.

Another approach is to poll specific locations in the LCD to see if a pixel has been drawn there. The message [getpixel x y] will return a message [pixel r g b x y] with the rgb color values in it. If r g and b are all 255 (white), there's no dot. All you need to do to make this work is use a counter to choose locations to test. If you just take the x values, the motion will tend to be scalewise, but it's not hard to break it up into chords or other patterns.

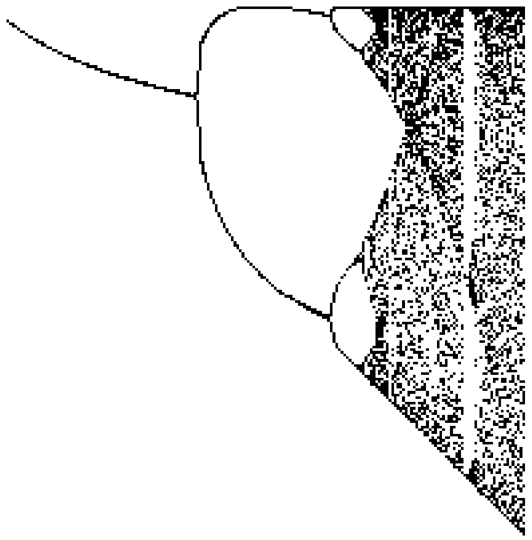
What does it sound like? Well, fractal patterns sound somewhat random, but at the same time vaguely familiar, as patterns seem to reappear from time to time.

Chaos

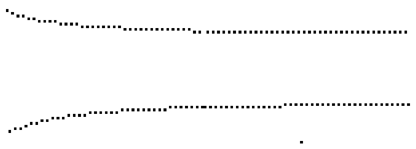
There are other ways to generate complex patterns. Chaos is the study of systems that are determinate (not random), but dependent on initial states in a complicated way. For instance the formula

$$f(x) = 1-cx^2$$

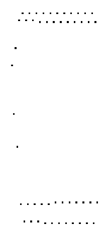
is evaluated by repeating the calculation using the last answer as x. Of course you need to start with something, so we pick a seed value for x. c is a constant whose choice also profoundly affects the outcome. Here is a graph of what happens as the constant is increased from 0 to 2 in steps of 0.01, with 100 iterations on each step.



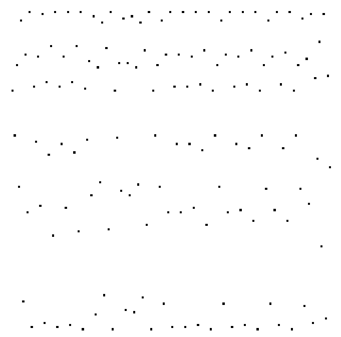
As long as c is smaller than about 0.75, the result follows a curve from near 1.0 down to 0.66 or so (exact numbers depend on initial x). At a magic value of c, the results start jumping back and forth, always landing near one of the two forks of the curve. At another magic value, the curves fork again, and results cluster around four values. Eventually, as c nears 2.0, the results are jumping in wildly unpredictable patterns. The values the formula favors for any given c are called attractors. Look at the pattern you get with a c of 0.76 and initial x of 0.5:



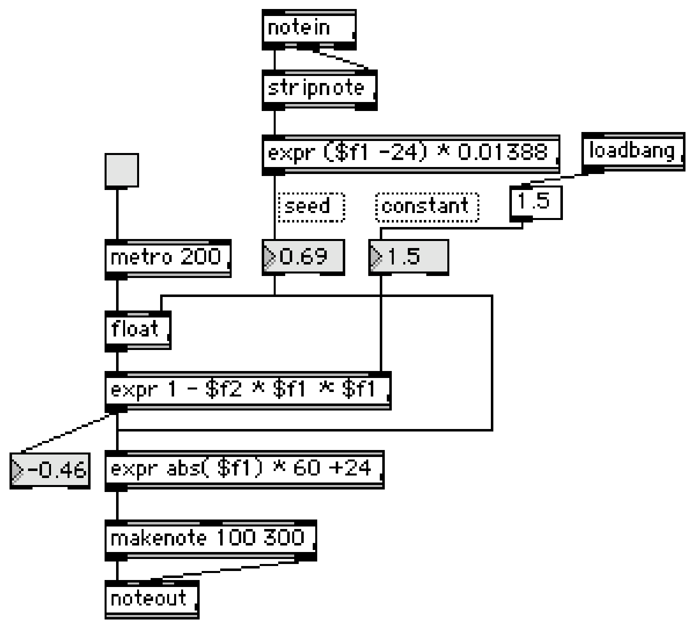
This is plotted as result vs. iteration number. The values basically alternate, converging on the attractors (which are 0.52 and 0.79). Here's what happens with c of 1.26:



There are now 4 attractors: –0.23, -0.10, 93, 99. But notice, in getting to the attractors the output hits some other values. When c is 1.5, the chaos begins to show:



If these values are mapped to notes, the results are pretty interesting. Here's a patcher for it:



The main calculation takes place in [expr 1-$f2 * $f1*$f1]. The other two expressions scale the input and output to appropriate values. Deriving the seed from note in gives a nice performance feature. Each key produces a unique pattern, some of which go on for quite a while before settling on the attractors. See what happens when you change the constants.