CHAPTER 4

RANDOM SELECTION I: DIRECT RANDOMNESS

No treatment of decision making can be complete if it neglects randomness. Random decisions are made by consulting some process whose outcome is unpredictable, such as a roll of a die. Such decisions are ~~are~~ most appropriate in situations admitting many equally suitable (or equally odius) options.

4.1 TERMINOLOGY

We will refer to a specific instance of a process yielding a random outcome as a *random incident*. The collection of all possible outcomes is called the *range* of the process. Ranges fall into two categories:

1. In a *discrete* range, any two outcomes have a finite number of intermediate outcomes. For example, if we spend an hour fishing in a lake, we might catch two

fishes, or three fishes, but never two and a half fishes.

2. In a continuous range, any two outcomes have an infinite number of intermediate outcomes. Random durations typically have continuous ranges: the time required to catch a fish might be any portion of an hour, or longer.

We say that a random incident or an associated random decision is weighted, or biased, when some outcomes are more likely than others. The weight, or probability, of an outcome measures its relative likelihood, and the collection of weights for all outcomes comprises the distribution of outcomes.

We refer to an outcome produced by a specific random incident as a sample. A collection of samples is called a population. The mean of a population gives the average of all the samples.

An important distinction exists between random distributions and statistical distributions; populations do not need to be random. Where randomness deals with the unpredictable behavior of individuals, statistics are concerned with whole populations. It is quite possible for a population to obey a given statistical distribution and yet be wholly predictable; for example, the population:

0, 1, 0, 1, 0, 1, 0, 1, 0, 1

has just as many zeros as ones. The relationship between statistical distributions and their random counterparts is evident in the fact that if we paint these zeros and ones on ten balls, place the balls in an urn, mix them around, and draw a ball blindly, then the likelihoods of the ball showing a zero or a one would be equal.

An important facet of randomness is the independence of random incidents. Independence is expressed in the fact that the outcome of a random incident remains wholly unaffected by any previous incident. For example, the probability of obtaining heads from a toss of a coin is 1/2. It remains 1/2 no matter how many heads have been tossed in the past. Extrapolating from this fact, we see that not only is a head equally as likely as a tail, but any sequence of heads and tails is equally as likely as any other sequence.

To see why any sequence is equally likely, perform 80 "random experiments", each consisting of three successive coin tosses. Divide these experiments into two groups, I and II, according to whether the first toss yields a head (group I) or a tail (group II). Since each outcome is equally likely, each of the two groups will hold about 40 experiments. Now, divide group I and group II each into two subgroups according to whether the second toss yields a head (subgroups I-A and II-A) or a tail (subgroups I-B and II-B). Since each of these outcomes is equally likely, each of the four subgroups will hold about 20 experiments. Finally, divide each subgroup into two sub-subgroups according to whether the third toss yields a head (sub-subgroups I-A-1, I-B-1, II-A-1, and II-B-1) or a tail (sub-subgroups I-A-2, I-B-2, II-A-2, and II-B-2). Again, since each outcome is equally likely, each of the eight sub-subgroups will hold about 10 experiments. At this point, we have constructed one category for each possible combination of heads and tails, and it is clear that the probability that an experiment will fall into any one of these categories is 10/80 or 1/8. Table 4-1 summarizes this result:

Table 4-1: Outcomes and relative probabilities of three coin tosses.

4.2 A SURVEY OF RANDOM DISTRIBUTIONS

Category	Toss 1	Toss 2	Toss 3	Probability
I-A-1	head	head	head	1/8
I-A-2	head	head	tail	1/8
I-B-1	head	tail	head	1/8
I-B-2	head	tail	tail	1/8
II-A-1	tail	head	head	1/8
II-A-2	tail	head	tail	1/8
II-B-1	tail	tail	head	1/8
II-B-2	tail	tail	tail	1/8

Table 4-1

	Outcome										
	2	3	4	5	6	7	8	9	10	11	12
Weight	1	2	3	4	5	6	5	4	3	2	1
Density	.028	.056	.083	.111	.139	.167	.139	.111	.083	.056	.028

Table 4-2

4.2.1 Uniform Randomness

The least biased random distributions are *uniform* distributions. Outcomes of a random incident are uniformly distributed over a *discrete* range when each outcome is equally likely. For example, throwing one six-sided die produces uniform integers from 1 to 6.

Since over a *continuous* range, the likelihood of any specific outcome is negligible, our definition of continuous uniformity must be subtler. We say that a random incident is uniformly distributed over a continuous range when, given any two equally sized regions within the range, the likelihoods of an outcome falling within either region are equal. For example, suppose we have a random incident whose outcomes are uniformly distributed over a continuous range from 0 to 20. Then the likelihood of an outcome falling between 0 and 10 is the same as its likelihood of falling between 10 and 20. Similarly, the likelihood of an outcome falling between 2.5 and 3 is the same as its likelihood of falling between 4.5 and 5 or its likelihood of falling between 18 and 18.5.

Uniform randomness over continuous ranges does not occur directly in everyday experience. However, continuous uniform randomness is relatively easy to simulate using a computer. It also provides a convenient basis for deriving other types of randomness, (including discrete uniform randomness).

4.2.1.1 Continuous Uniform Samples - The most common method of generating uniform samples over a continuous range is the "linear congruence" method introduced by Derrick Henry Lehmer (1951; discussed in Knuth's <u>Seminumerical Algorithms</u>, pp. 9-24). This method is not truly random, but it produces a sequence of uniform samples which lacks, for most practical purposes, any discernable pattern.

The real-valued library function RANF implements a simplified version of Lehmer's method which is sufficient for our purposes. Each call to RANF returns a real number which is uniformly distributed between 0 and 1. The DATA statement (line 2) provides the value of SEED only for the first invocation of RANF; each new call transforms this value. RANF multiplies SEED by 877 (line 3), pares off all digits left of the decimal point (line 4), and returns the new value of SEED as the result (line 5):

-- Programming example 4-1: Listing of function RANF --

The variables FACTOR and SEED adhere to special criteria affecting the "period" of the random sequence, so these variables

Ex 4-1

```
function RANF()
data FACTOR/877.0/,SEED/0.2937859/
SEED = FACTOR * SEED
SEED = SEED - float(ifix(SEED))
RANF = SEED
return
end
```

Ex 4-2

```
do (50 times)
  TRASH = RANF()
repeat
```

should not be altered directly. A new sequence of samples may be obtained by "cycling" RANF an arbitrary number of times at the outset of the program. The following excerpt of code cycles RANF 50 times:

-- Programming example 4-2 --

Figure 4-1 graphs a population of 218 consecutive samples produced by RANF. This graph verifies that although Lehmer's method of generating samples is actually determinate, it is no simple matter to predict any sample from its predecessor.

Figure 4-1: Graph of 218 consecutive samples generated by RANF - The horizontal scale indicates successive calls to RANF, while the vertical scales gives the magnitude of each sample. Magnitudes are distributed uniformly over a continuous range from 0 to 1.

The best way to appreciate the distribution of a randomly generated population is to construct a histogram. This procedure involves 1) dividing the range of outcomes into equally sized regions and 2) graphing the number of samples in each region. Figure 4-2 presents four histograms of samples produced by RANF. The histograms are cumulative, that is, the first

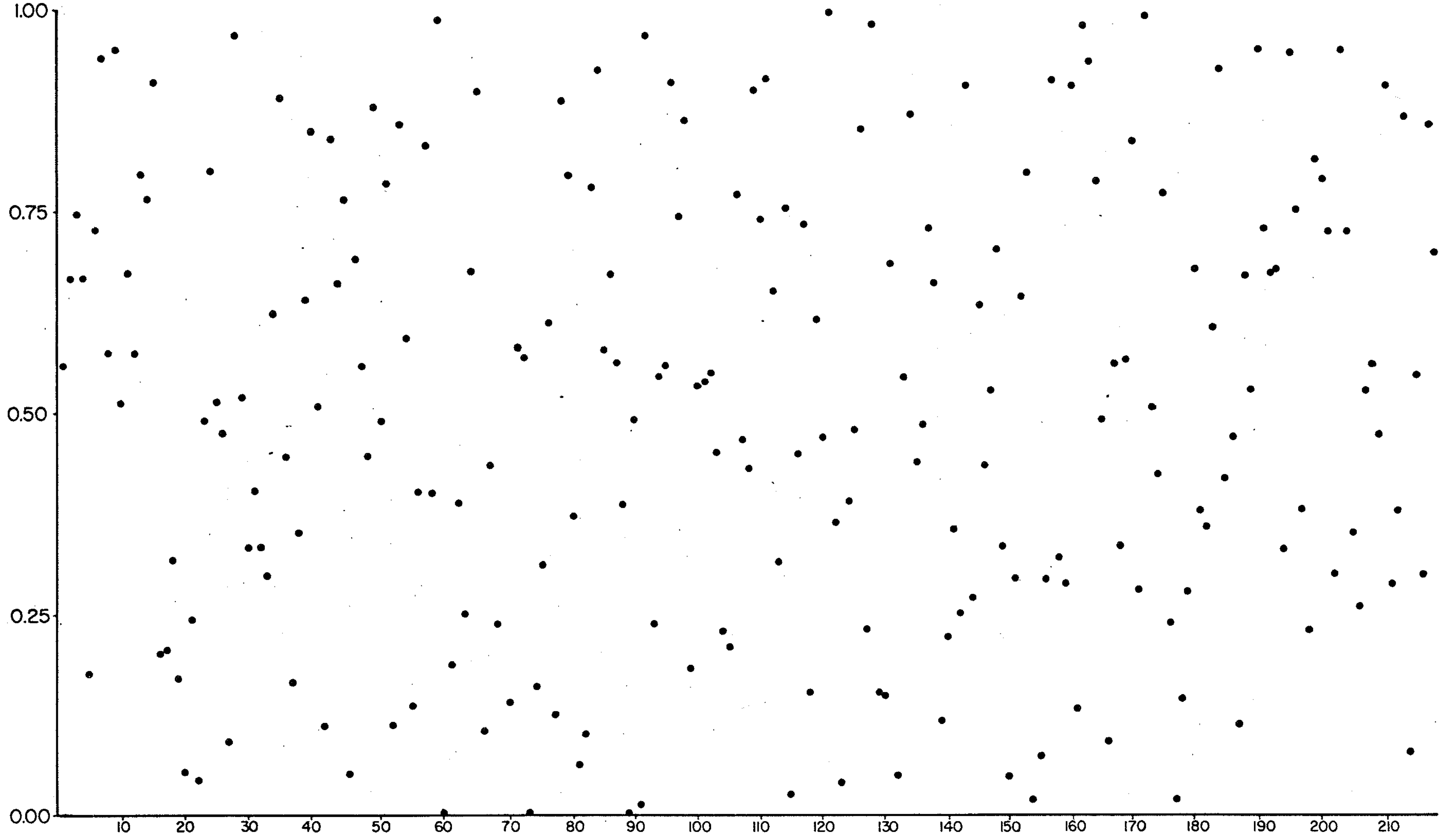

1.00
0.75
0.50
0.25
0.00
10
20
30
40
50
60
70
80
90
100
110
120
130
140
150
160
170
180
190
200
210

Fig 4-1

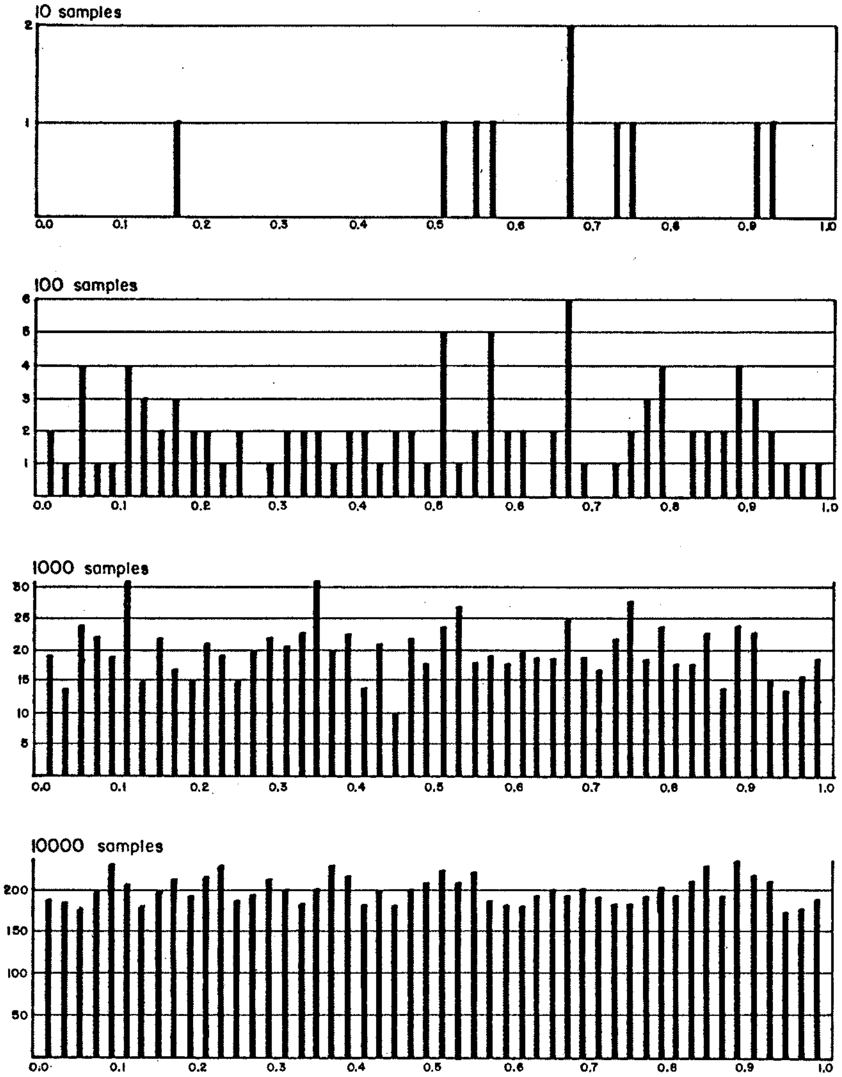

10 samples
2
1
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
100 samples
6
5
4
3
2
1
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
1000 samples
30
25
20
15
10
5
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
10000 samples
200
150
100
50
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0

Fig 4-2

histogram graphs 10 samples, the second histogram combines those 10 samples with 90 new samples for a total of 100, the third histogram combines those 100 with 900 new samples for a total of 1000, and so on. Notice that the population of 10 samples is certainly not uniformly distributed; we can only begin regarding the spread as even with populations on the order of 100 samples or greater.

Figure 4-2: Histograms of samples generated by RANF - Each bar tallies the number of samples falling within an interval of width 1/50.

4.2.1.2 Discrete Uniform Samples - The integer-valued ~~integer~~ library function IRND produces random integers over the range from 1 to NUM, with equal likelihood for each outcome. IRND first scales the output of function RANF by NUM. This procedure yields a sample which has an equal chance of falling into the region between 0 and 1 as it has of falling between 1 and 2, between 2 and 3, and so on up to the region between NUM ~~N~~-1 and NUM ~~N~~. Paring off the fractional part gives uniform integers from 0 to NUM ~~N~~-1, so IRND must add 1 to achieve the desired range.

Ex 4-3

```
      function IRND(NUM)
      IRND = ifix(RANF()*float(NUM)) + 1
      return
      end
```

Ex 4-4

```
      subroutine ALEA(RESULT,VALUE,NUM)
      dimension VALUE(1)
C     Generate scaled random number
      R = RANF() * float(NUM)
C     Look up corresponding entry in array VALUE
      RESULT = VALUE(ifix(R)+1)
      return
      end
```

Ex 4-5

```
      function SUCCES(P)
      logical SUCCES
      if (RANF().le.P) then
        SUCCES = .true.
      else
        SUCCES = .false.
      end if
      return
      end
```

-- Programming example 4-3: Listing of function IRND --

The library subroutine ALEA is based upon a feature from Koenig's PROJECT2 program (Koenig, 1970b) which returns one option out of a "supply" (Koenig's term for a discrete range) with equal likelihoods for each option. ALEA requres three arguments:

1. RESULT - ALEA selects one element of array VALUE and transfers its contents to this location (line 6). RESULT may be either an integer or a real.

2. VALUE - Repertory ("supply") of options. VALUE must be an array whose dimension in the calling program is NUM and whose type matches that of RESULT.

3. NUM - Number of options. NUM must be an integer.

-- Programming example 4-4: Listing of subroutine ALEA --

4.2.2 Weighted Discrete Randomness

We have all experienced weighted random incidents. The act of rolling two six-sided dice, for example, is six times more likely to produce a seven than it is to produce a two. This happens because there is only one way of producing a two--rolling two ones--while there are six ways of producing a seven. One may roll a one and a six, a two and a five, a three and a four, a four and a three, a five and a one, or a six and a one. Table 4-2 gives the distribution of weights for two dice:

Table 4-2: Weights and densities for numbers rolled using two six-sided dice.

Often we might wish to relate the number of times a specific outcome appears to the total size of the population. If n represents the number of specific occurances while N represents the total size, then the proportion n/N becomes a very useful kind of weight which we call a <u>density</u>. A distribution of densities always sums ~~up [illegible]~~ to unity. Densities make it possible to compare distributions of different-sized populations. The <u>density function</u> of a distribution gives relationships between each outcome in the range and its density. Table 4-2 also gives densities for each possible number rolled by two six-sided dice, as derived from the weights (note 1).

4.2.2.1 The *Bernoulli* Distribution - This archetypal distribution honors the mathematician Jakob Bernoulli (1654-1705). It is also called the *point* distribution. The Bernoulli distribution models the simplest random incident: a single *trial* with two possible outcomes, success and failure, and a fixed probability p of success. Tossing a coin is the simplest example; here p = 1/2. Equation 4-1 expresses the density function when -1 (.true.) represents a success and 0 (.false.) represents a failure:

$$\begin{aligned} f(-1) &= p \\ f(\ 0) &= 1 - p \end{aligned} \qquad \text{(Equation 4-1)}$$

The ~~logical~~ logical-valued library function SUCCES returns Bernoulli-distributed successes and failures with probability of success P. Figure 4-3 presents histograms of samples produced by SUCCES.

-- Programming example 4-5: Listing of function SUCCESS --

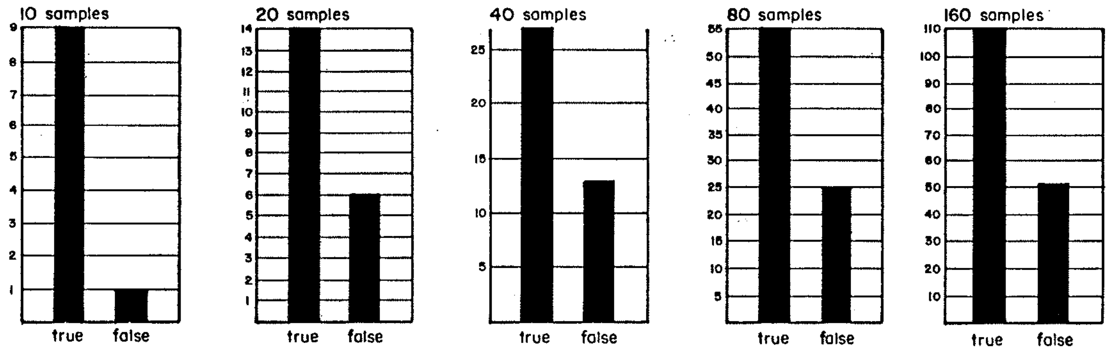

10 samples
9
8
7
6
5
4
3
2
1
true
false
20 samples
14
13
12
11
10
9
8
7
6
5
4
3
2
1
true
false
40 samples
25
20
15
10
5
true
false
80 samples
55
50
45
40
35
30
25
20
15
10
5
true
false
160 samples
110
100
90
80
70
60
50
40
30
20
10
true
false

Fig. 4-3

Figure 4-3: Histograms of samples produced by SUCCES - Each bar tallies the number of successes (.true.) and failures (.false.) generated by SUCCES with argument P=2/3.

4.2.2.2 The Binomial Distribution - If we initiate n consecutive Bernoulli trials, each with probability p of success (for example, if we toss a coin n times) then the number of successes follows a binomial distribution. Possible outcomes range from 0 successes to n successes; Equation 4-2 gives the density for an outcome of i successes (note 2).

$$f(i) = \frac{n!}{i!(n-i)!} p^{i} (1-p)^{n-i} \qquad \text{(Equation 4-2)}$$

Graphs of binomial densities for various values of p appear in Figure 4-4. The mean of a binomial distribution, which predicts the average outcome over a binomial population of samples, is np.

Figure 4-4: Graphs of binomial densities - The horizontal scale gives possible outcomes, expressed as

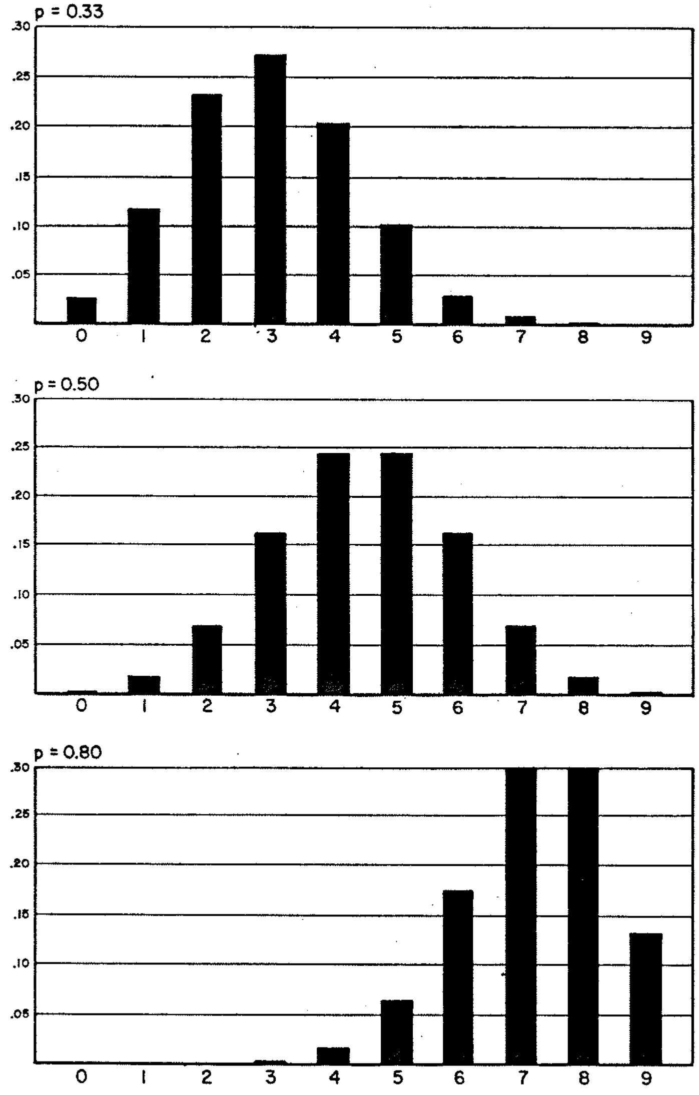

p = 0.33
.30
.25
.20
.15
.10
.05
0 1 2 3 4 5 6 7 8 9
p = 0.50
.30
.25
.20
.15
.10
.05
0 1 2 3 4 5 6 7 8 9
p = 0.80
.30
.25
.20
.15
.10
.05
0 1 2 3 4 5 6 7 8 9

Fig 4-4

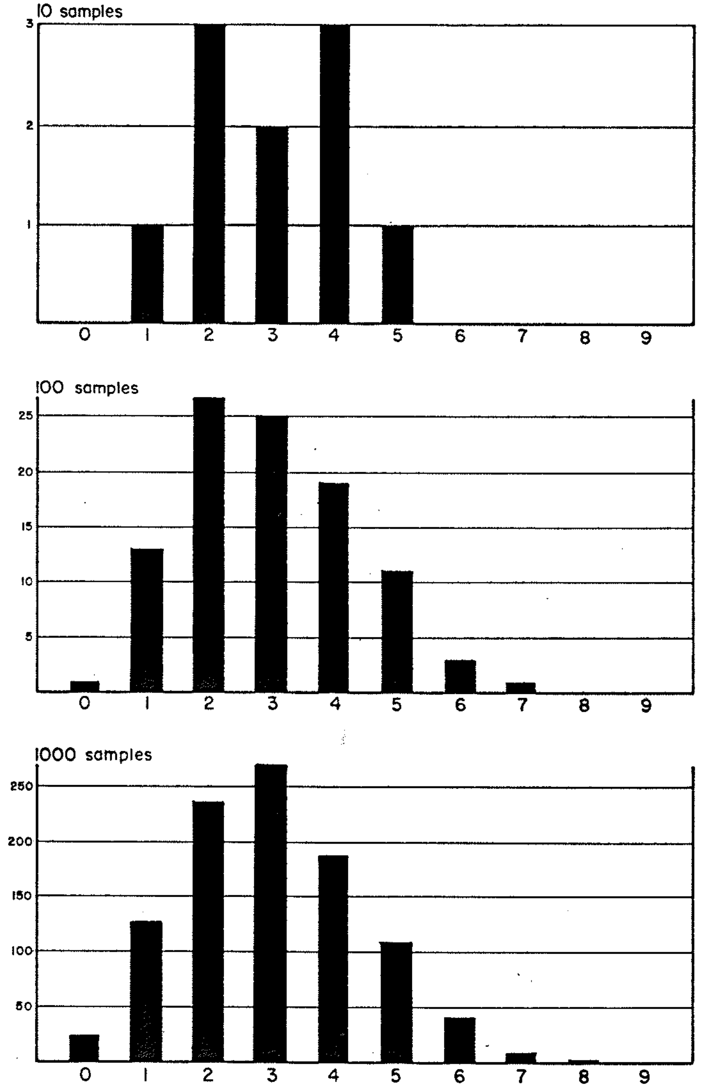

10 samples
3
2
1
0 1 2 3 4 5 6 7 8 9
100 samples
25
20
15
10
5
0 1 2 3 4 5 6 7 8 9
1000 samples
250
200
150
100
50
0 1 2 3 4 5 6 7 8 9

Fig 4-5

numbers of successes obtained during sequence of 9 Bernoulli trials. The vertical scale gives relative likelihoods for each outcome.

The ~~integer~~ integer-valued library function IBINOM produces binomially distributed samples assuming N Bernoulli trials, each with probability P of success for each trial. Figure 4-5 presents histograms of samples produced by IBINOM.

-- Programming example 4-6: Listing of function IBINOM --

Figure 4-5: Histograms of samples generated by IBINOM - The horizontal scale in each graph gives possible outcomes, expressed as numbers of successes obtained during sequence of 9 Bernoulli trials with probability 1/3 of success. The vertical scale tallies the number of times each outcome occurs within the population.

4.2.2.3 The Geometric Distribution - If we initiate consecutive Bernoulli trials with a fixed probability of success p until one trial fails (for example, if we toss a coin repeatedly until it comes up tails), the number of successes

Ex 4-6

```
function IBINOM(N,P)
logical SUCCES
IBINOM = 0
do (N times)
  if (SUCCES(P)) IBINOM = IBINOM + 1
repeat
return
end
```

Ex 4-7

```
function IGEOM(AVG)
logical SUCCES
P = AVG/(AVG+1.0)
IGEOM = 0
do
  if (.not.SUCCES(P)) return
  IGEOM = IGEOM + 1
repeat
end
```

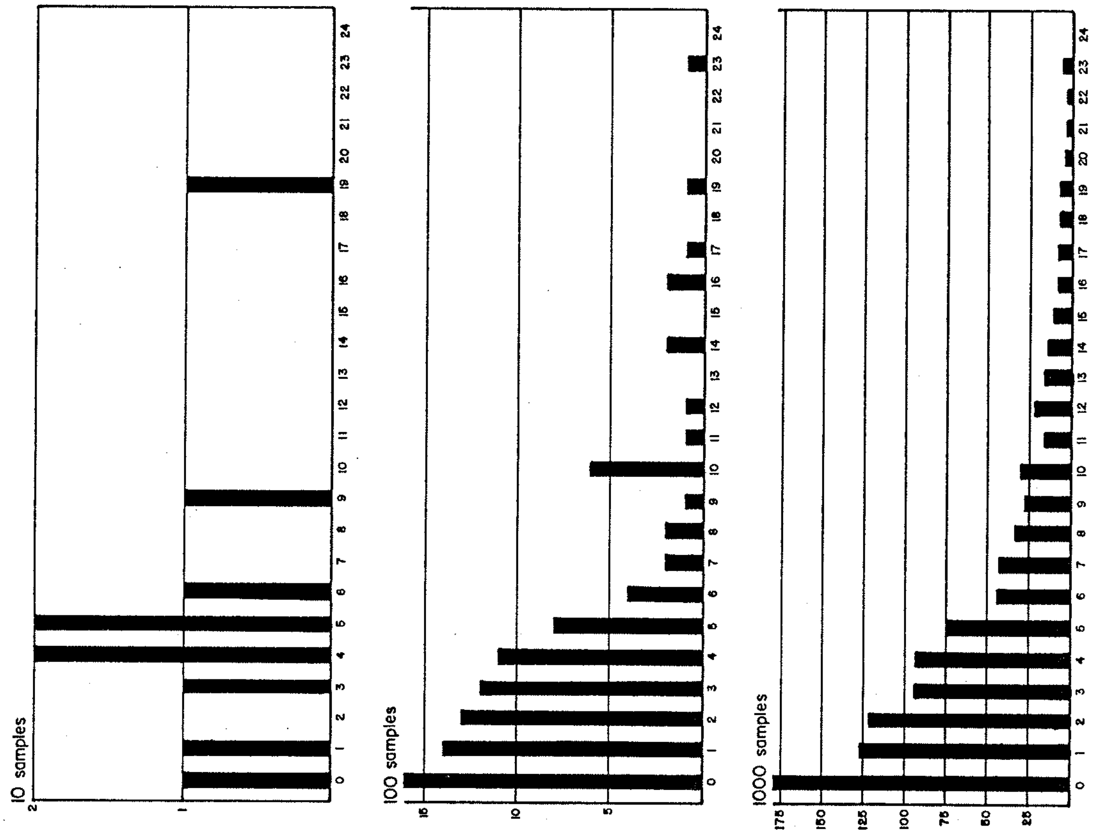

10 samples
2
1
100 samples
15
10
5
1000 samples
175
150
125
100
75
50
25
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24

Fig 4-7

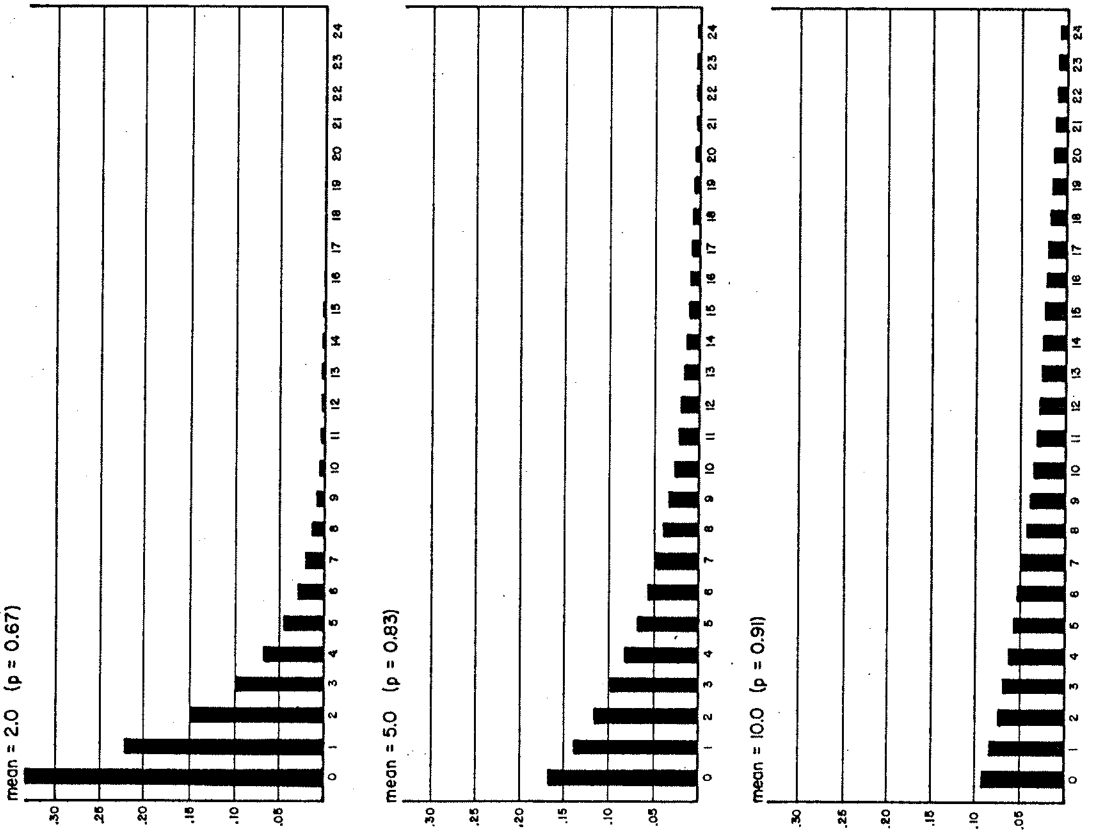

mean = 2.0 (p = 0.67)
mean = 5.0 (p = 0.83)
mean = 10.0 (p = 0.91)
.30
.25
.20
.15
.10
.05
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24

Fig 4-6

follows a geometric distribution. Possible outcomes range from 0 upwards, though large values become increasingly improbable. Equation 4-3 gives the density of i successes.

$$f(i) = (1-p)\,p^{i} \qquad \text{(Equation 4-3)}$$

Graphs of this density function for various values of p appear in Figure 4-6. The <u>mean</u> of a geometric distribution, which predicts the average outcome over a geometric population of samples, is p/(1-p).

Figure 4-6: Graphs of geometric densities - The horizontal scale gives possible outcomes, expressed as numbers of successes occuring before one failure during an open-ended sequence of Bernoulli trials. The vertical scale gives relative likelihoods for each outcome.

The ~~integer~~ integer-valued library function IGEOM produces geometrically distributed samples with mean AVG. It begins by using AVG to determine the probability of success P for a single Bernoulli trial (line 1). The loop (lines 5-8) counts successive trials up to the first failure. Figure 4-7 presents histograms of samples produced by IGEOM.

-- Programming example 4-7: Listing of function IGEOM --

Figure 4-7: Histograms of samples generated by IBINOM with AVG=5.0 - The horizontal scale gives possible outcomes, expressed as numbers of successes occuring before one failure during an open-ended sequence of Bernoulli trials. The vertical scale tallies the number of times each outcome occurs within a population.

4.2.2.4 The Poisson Distribution - This distribution was formulated by Simeon-Denis Poisson (1781-1840). It models occurances of rare events over a period of time, for examples: the number of beta particles detected from a radioactive source in a second, the number of fish caught from a large lake in an hour, the number of lightbulbs failing in a month. We shall defer a specific motivation of the Poisson distribution until we encounter the "exponential" distribution later in this chapter. Like geometrically distributed samples, outcomes of the Poisson distribution range from 0 upwards. Equation 4-4 gives the density of i events for the Poisson distribution. The parameter u gives the average number of events in a period.

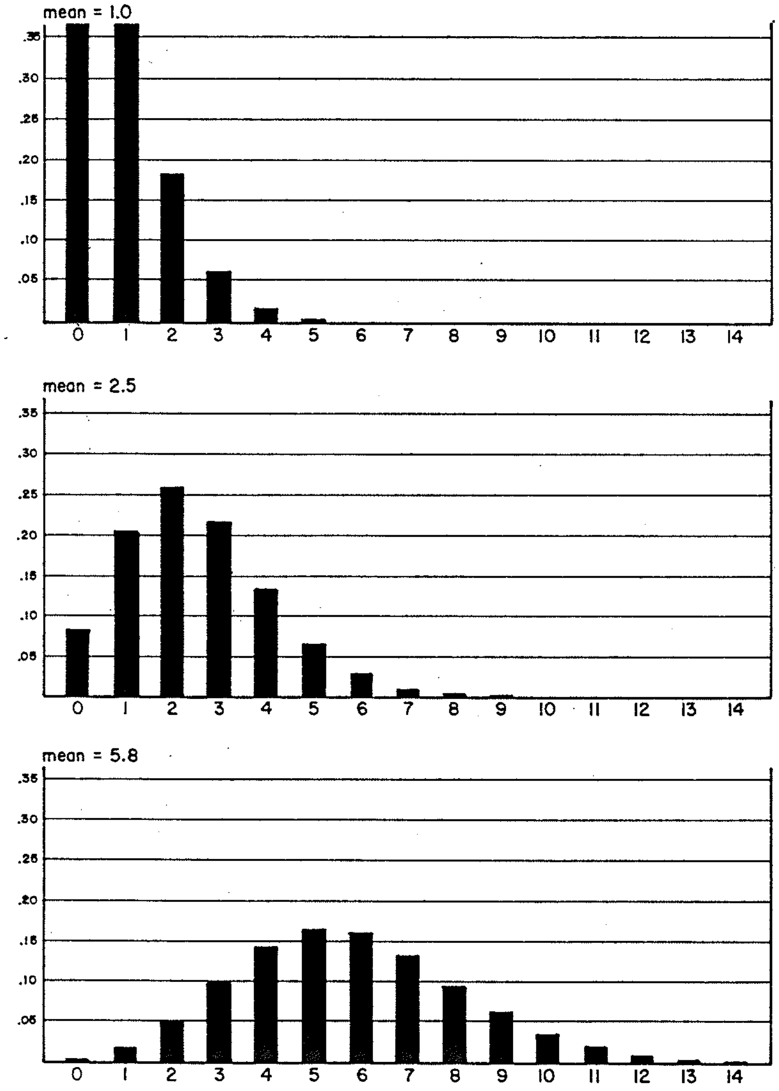

mean = 1.0
mean = 2.5
mean = 5.8
.35
.30
.25
.20
.15
.10
.05
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14

Fig 4-8

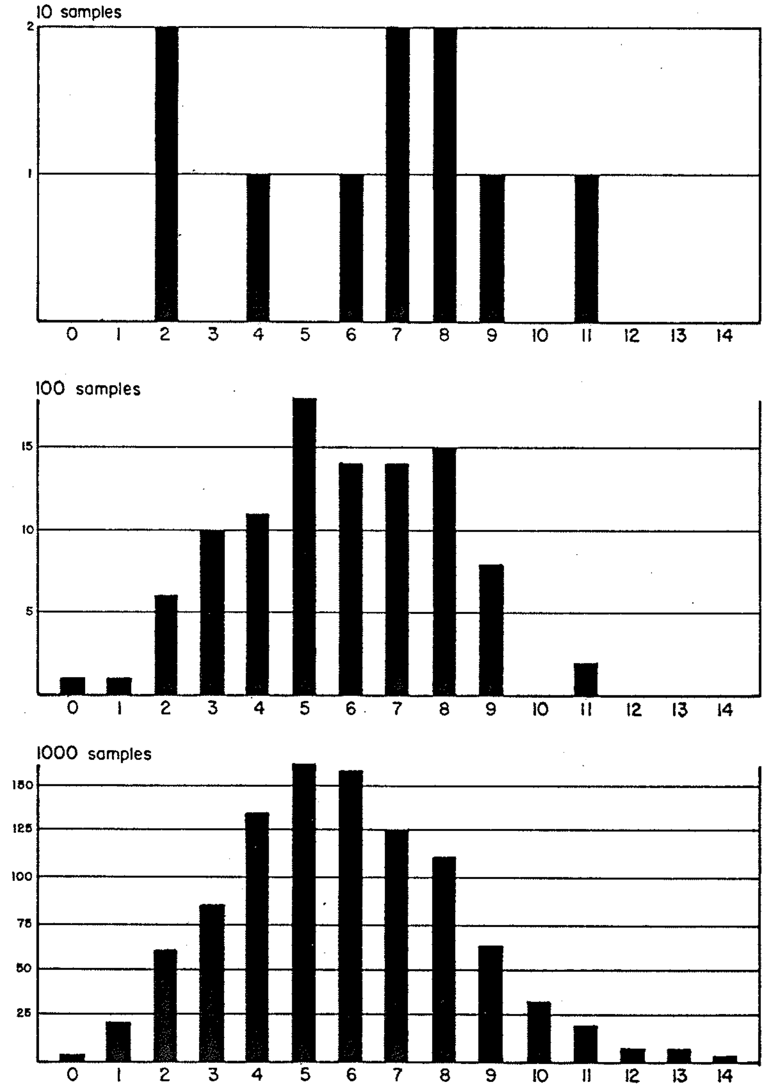

10 samples
2
1
100 samples
15
10
5
1000 samples
150
125
100
75
50
25
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14

Fig 4-9

$$f(i) = \frac{e^{-u} u^{i}}{i!} \qquad \text{(Equation 4-4)}$$

Graphs of this density function for different values of u appear in Figure 4-8.

Figure 4-8: Graphs of Poisson densities - The horizontal scale gives possible outcomes, while the vertical scale gives relative likelihoods for each outcome.

The ~~integer~~ integer-valued library function IPOISS generates Poisson-distributed samples with mean AVG. [~~The algorithm appears on page 117 of Knuth's Seminumerical Algorithms.~~] Figure 4-9 presents histograms of samples produced by IPOISS.

-- Programming example 4-8: Listing of function IPOISS --

Figure 4-9: Histograms of samples generated by IPOISS with AVG=5.8 - The horizontal scale gives possible outcomes, while the vertical scale tallies the number of times each outcome occurs within a population.

Ex 4-8

```
      function IPOISS(AVG)
      P = exp(-AVG)
      Q = 1.0
      IPOISS = 0
      do
        Q = Q * RANF()
        if (Q.lt.P) return
        IPOISS = IPOISS + 1
      repeat
      end
```

Ex 4-9

```
      subroutine SELECT(RESULT,VALUE,WEIGHT,SUM,NUM)
      dimension VALUE(1),WEIGHT(1)
C     Generate scaled random number
      R = SUM * RANF()
C     Locate weighted region aligning with this number
      do (I=1,NUM)
        W = WEIGHT(I)
        if (R.lt.W) exit
        R = R - W
      repeat
      if (I.gt.NUM) stop 'Bad weights for SELECT.'
C     Look up corresponding entry in array VALUE
      RESULT = VALUE(I)
      return
      end
```

← 4.2.2.5 Stored Discrete Distributions - In automated composition it is often necessary to select random options directly with respect to the individual weights. Sometimes these weights may be specified by the composer; in other cases, they may derive from a random process which does lend itself to algorithmic coding.

The library subroutine SELECT returns one option from a repertory relative to a stored distribution of weights. It requires 5 arguments:

1. RESULT - SELECT chooses one element of array VALUE and transfers its contents to this location (line 13). RESULT may be either an integer or a real.

2. VALUE - Repertory of options. VALUE must be an array whose dimension in the calling program is NUM and whose type matches that of RESULT.

3. WEIGHT - Relative weights for each option. WEIGHT must be a real array whose dimension in the calling program is NUM. Each weight must be zero or greater, and at least one weight must be positive.

4. SUM - total of all NUM weights in array WEIGHT. SUM

must be real.

5. NUM - Number of options. NUM must be an integer.

Figure 4-10 illustrates how SELECT chooses one of 15 options relative to a stored distribution of weights. The random value R (line 4) is distributed unformly between 0 and SUM. Sizes of the NUM regions of this range associated each option are determined by the corresponding weights. Consider two arbitrary options, i and j: If option i has twice the weight of option j, the region of the total range associated with option i is twice as large, so R is twice as likely to fall within the ith region as the jth region.

-- Programming example 4-9: Listing of subroutine SELECT --

Figure 4-10: Mechanics of subroutine SELECT - Each column represents one iteration of the loop (lines 6-10), and the number at the top of each column indicates the value of I for that iteration. The black stripe represents the variable R.

If the weights remain constant, there are two steps one can take to speed up selection of an option. The first step is to

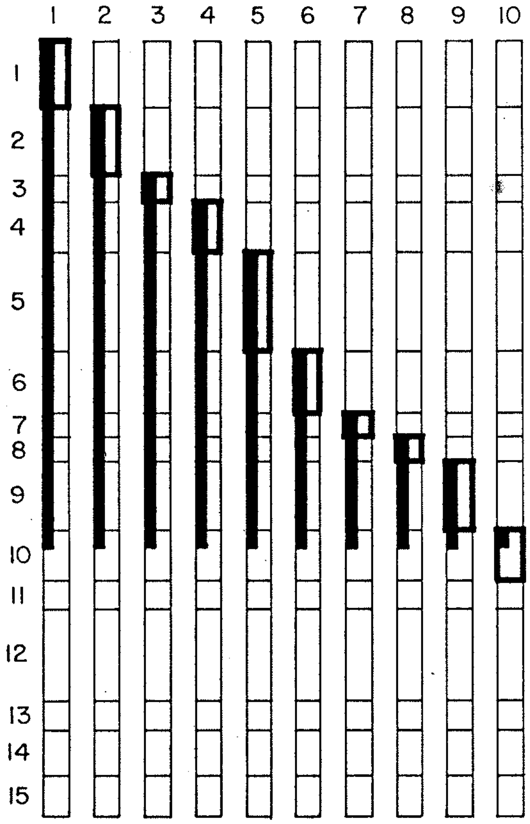


Fig 4-10

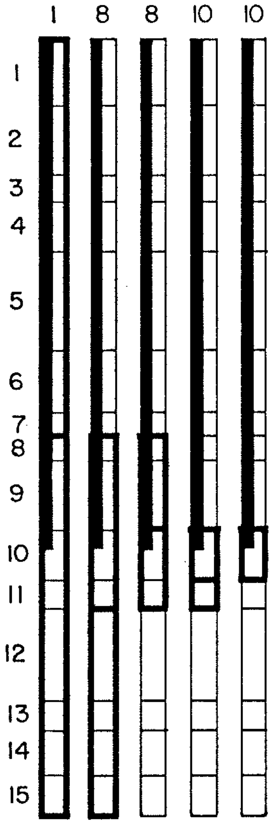


Fig 4-11

Ex 4-10

```
C     subroutine QUICK(RESULT,VALUE,BOUND,NUM)
      dimension VALUE(1),BOUND(1)
C     Generate scaled random number
      R = BOUND(NUM) * RANF()
C     Locate weighted region aligning with this number
      I1 = 1
      I2 = NUM
      do
        if (I1.ge.I2) exit
        I = (I1 + I2) / 2
        B = BOUND(I)
        if (R.le.B) then
          I2 = I
        else
          I1 = I + 1
        end if
      repeat
C     Look up corresponding entry in array VALUE
      RESULT = VALUE(I1)
      end
```

compute boundaries for each region of the range from 0 to SUM. This step obviates the need to pare down R with each iteration. The fact that these boundaries proceed in a strictly increasing sequence makes possible the second step, a binary search for the region holding R. (Knuth discusses binary searches on page 407 of Sorting and Searching). The library subroutine QUICK implements these two steps. Figure 4-11 illustrates how QUICK works for the same distribution of weights illustrated in Figure 4-10.

-- Programming example 4-10: Listing of subroutine QUICK --

Figure 4-11: Mechanics of subroutine QUICK - Each column represents one iteration of the loop (lines 8-17); the number at the top of each column indicates the value of I1 for that iteration. The black stripe represents the variable R.

4.2.3 Continuous Weighted Randomness

The usual procedure for generating non-unform random numbers over a continuous range is not to simulate the random process

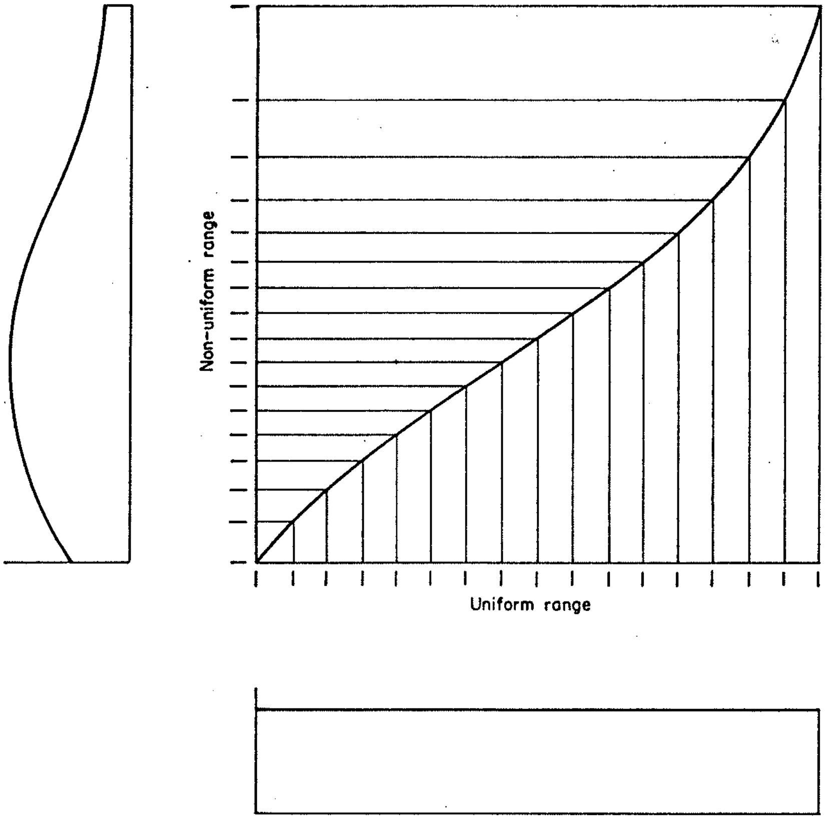


Fig 4-12

directly (as with the library functions IBINOM and IGEOM) but rather first to generate a uniform random number and then to subject this number to a statistical transformation which "maps" each point in the uniform range into a corresponding point in the non-uniform range. Figure 4-12 illustrates how the relatively steeper portions of a statistical transformation act to spread uniform samples over correspondingly wider regions of the non-uniform range.

Figure 4-12: Mechanics of a statistical transformation.

4.2.3.1 The Exponential Distribution - The exponential distribution is the continuous analog of the geometric distribution. Exponential distributions model lifetimes of physical entities such as atomic nuclei and lightbulbs. We might say that they result from consecutive Bernoulli trials, each occuring over an unmeasurably short period. If a trial succeeds, the entity survives, otherwise it ~~dies~~ expires. The probability of survival approaches unity; however with so many trials, the entity is bound to ~~die~~ expire sometime.

An inherent assumption of an exponential distribution is that the probability of failure is independent of past history. For example, nuclear physicists regard the probability of a nucleus throwing out a beta particle in a given moment to be unchanged whether the nucleus has existed previously for five minutes or five centuries. Such a nucleus grows no less stable with time; the drop in probability as lifetimes increase results solely from the fact that it has to survive through one moment to exist during the next.

Exponentially distributed samples range upwards from 0. The musical attraction of the exponential distribution is the balance it acheives between short and long durations: The cumulative amount of time occupied by long durations equals the cumulative amount occupied by short durations. Equation 4-5 gives the density at any positive value x.

$$f(x) = u\ e^{-ux} \qquad \text{(Equation 4-5)}$$

The <u>mean</u>, u, gives the expected lifetime of the entity under consideration. The mean is also called <u>halflife</u> because it predicts the amount of time required for half of a large population of identical entities to die away.

The basic algorithm for producing random samples adhering to a pure exponential distribution involves generating a random sample X uniformly between 0 and 1 and then subjecting X to the statistical transformation given by Equation 4-6 to obtain an exponentially distributed sample Y:

$$Y = -u \log_e(X) \qquad \text{(Equation 4-6)}$$

~~(Equation 4-6~~ ~~is taken from~~ ~~appears on page 114 of Knuth's Seminumerical~~ footnote
~~Algorithms.)~~ Figure 4-13 illustrates how this statistical transformation produces exponentially distributed samples from uniform ones. Notice that because the transformation slopes downward, it has the effect of mapping large uniform samples into small exponential samples, and vice versa. Figure 4-14 presents histograms of exponentially distributed samples obtained using this method.

Figure 4-13: Generating exponentially distributed samples.

Figure 4-14: Histograms of exponentially distributed samples - Each bar tallies the number of samples falling within a region of width 1/10 for a mean u=2.0.

John Myhill (1979) has developed a method for implementing a continuum of random distributions ranging from complete certainty that a sample will be located at the average value u to a true

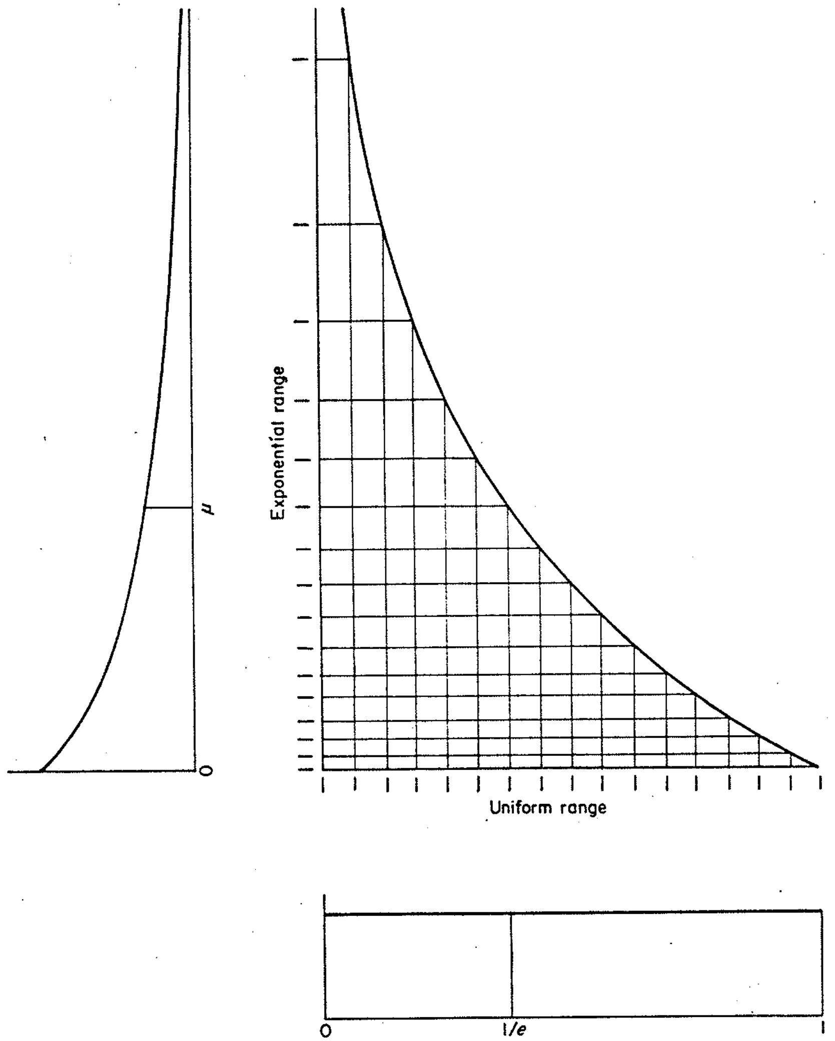


Fig 4-13

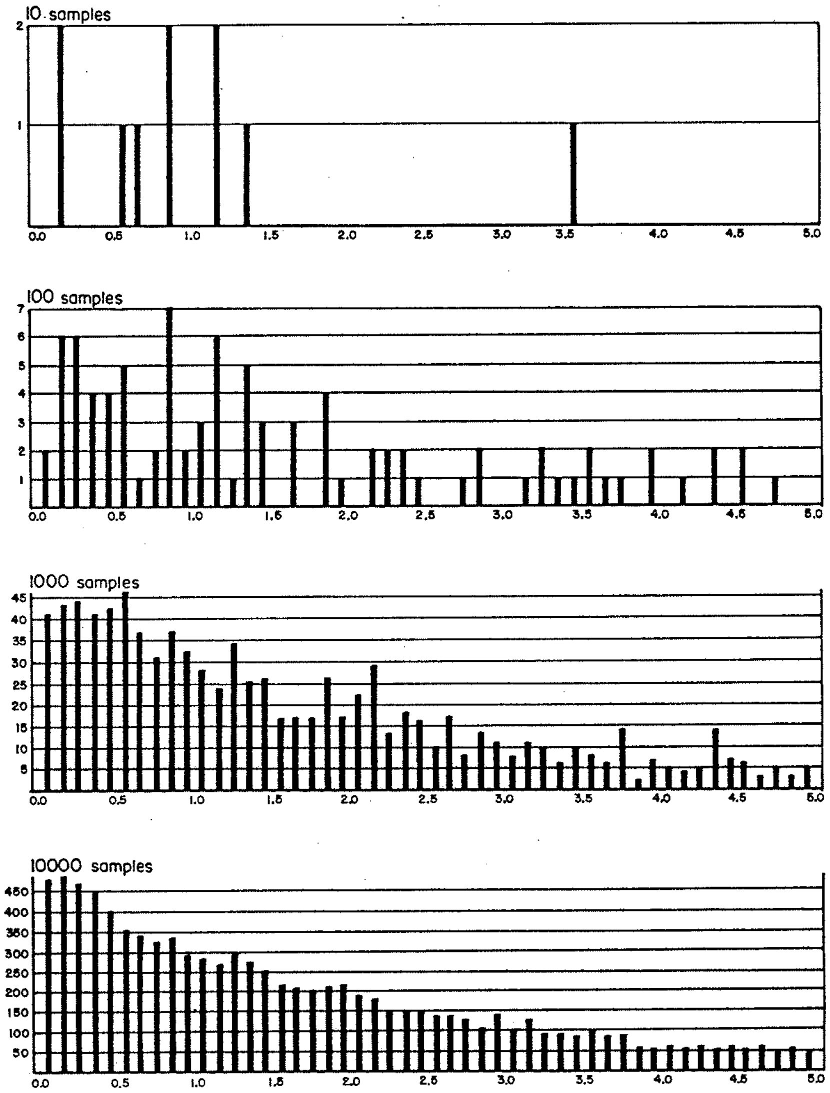


Fig 4-14

exponential distribution. Where a pure exponential distribution admits samples which can be arbitrarily close to zero or arbitrarily large, Myhill's method limits samples to within an explicit minimum (larger than zero) and a finite maximum. These limits depend upon two arguments supplied by the user, the average duration and a ratio relating the maximum and minimum samples. For example, suppose we wish to generate a sequence of random durations. If we indicate an average duration of 2.0 sixteenths and maximum duration 4.0 times as large as the minimum, then the actual durations produced by Myhill's method will range from 0.92 sixteenths to 3.70 sixteenths (3.70/0.92=4.0).

The real-valued library function RANX implements Myhill's procedures. RANX requires two arguments:

1. AVG - average value. AVG must be a real number whose value is positive.

2. PROPOR - maximum sample divided by minimum sample. PROPOR must be a real number whose value is 1.0 or greater.

-- Programming example 4-11: function RANX --

Ex 4-11

```
function RANX(AVG,PROPOR)
if (AVG.le.0. .or. PROPOR.lt.1.0) then
  stop 'Bad argument to RANX.'
else if (PROPOR.lt.1.01) then
  RANX = AVG
else if (PROPOR.lt.1000.0) then
  R2 = PROPOR ** (-1.0/(PROPOR-1.0))
  R1 = R2 ** PROPOR
  R = (R2-R1)*RANF() + R1
  RANX = -AVG * alog(R)
else
  RANX = -AVG * alog(RANF())
end if
return
end
```

Ex 4-12

```
function WARP(R)
if (R.le.1.0) stop 'Bad argument to WARP.'
WARP = alog((R-1.0)*RANF()+1.0)/alog(R)
return
end
```

The mathematics behind lines 8 and 9 of function RANX are beyond the scope of this book. However, the quantity (R2-R1) is significant because it measures how closely the output of RANX approximates a pure exponential distribution. Figure 4-15 illustrates how this quantity varies with PROPOR. Notice that Figure 4-15 indicates ratios logarithmically, that is, each mark indicates a doubling of ratios (note 4). Ratios less than 2.0 produce less than 25% approximations; when used to control rhythmic ~~durations~~ periods, results are 'drunkenly' ~~periodic~~ regular. Ratios ranging from 4.0 to 16.0 produce approximations ranging from 50% to 75%; results are quite syncopated but still noticeably less irregular than true exponential rhythms. Ratios above 128.0 produce approximations above 95%; results are indistinguishable from exponential results.

Figure 4-15: Approximation of RANX to a pure exponential distribution.

The exponential distribution relates to the <u>Poisson</u> distribution as follows: Consider a sequence of random durations, each distributed exponentially with an average length of a seconds. A sequence of lightbulbs is an appropriate example. It may be shown using some rather complicated mathematics (see, for instance, Karlin and Taylor, 1975, pages 22

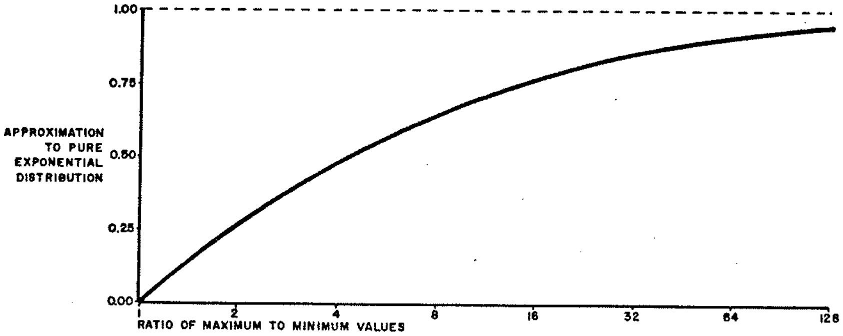

1.00
0.75
0.50
0.25
0.00
APPROXIMATION TO PURE EXPONENTIAL DISTRIBUTION
1
2
4
8
16
32
64
128
RATIO OF MAXIMUM TO MINIMUM VALUES

Fig 4-15

to 26) that the number of new durations beginning in a period t seconds long follows a Poisson distribution with mean t/a. Specifically, this mean is $1/a$ when $t=1$. The reason why the Poisson parameter is the inverse of the exponential parameter should be clear: if a typical duration lasts 1/5 of a second, we could then reasonably expect 5 such durations to fill the unit.

4.2.3.2 A '<u>Warped</u>' Distribution - Another useful distribution increases proportionally in density as values move along the continuous range from 0 to 1. Equation 4-7 gives its density function; the "warping factor" r has the effect of making the distribution r times as dense at 1 as it is at 0:

$$f(x) = \frac{1}{r - 1} \log_e(r)\, r^x \qquad \text{(Equation 4-7)}$$

Equation 4-8 provides the means for transforming a uniform distribution into a 'warped' distribution. Y represents a warped sample corresponding to the uniform sample X, given a warping factor r:

$$Y = \log_{r}((r-1)*X+1) \qquad \text{(Equation 4-8)}$$

Figure 4-16 illustrates how Equation 4-8 transforms uniform samples into 'warped' ones.

Figure 4-16: Generating proptionally distributed samples.

The real-valued library function WARP returns samples 'warped' by the factor R. R must be a real number whose value exceeds unity. Figure 4-17 presents histograms of samples produced by WARP.

-- Programming example 4-12: function WARP --

Figure 4-17: Histograms of proportionally distributed samples - Each bar tallies the number of samples falling within a region of width 1/50 for a warping factor r=8.0.

A useful musical application of this proportional distribution is the following: suppose one has a process (random or otherwise) which generates registers uniformly distributed over the range of the piano. The perceptual result might be that the lowest registers will sound muddy, while the highest will

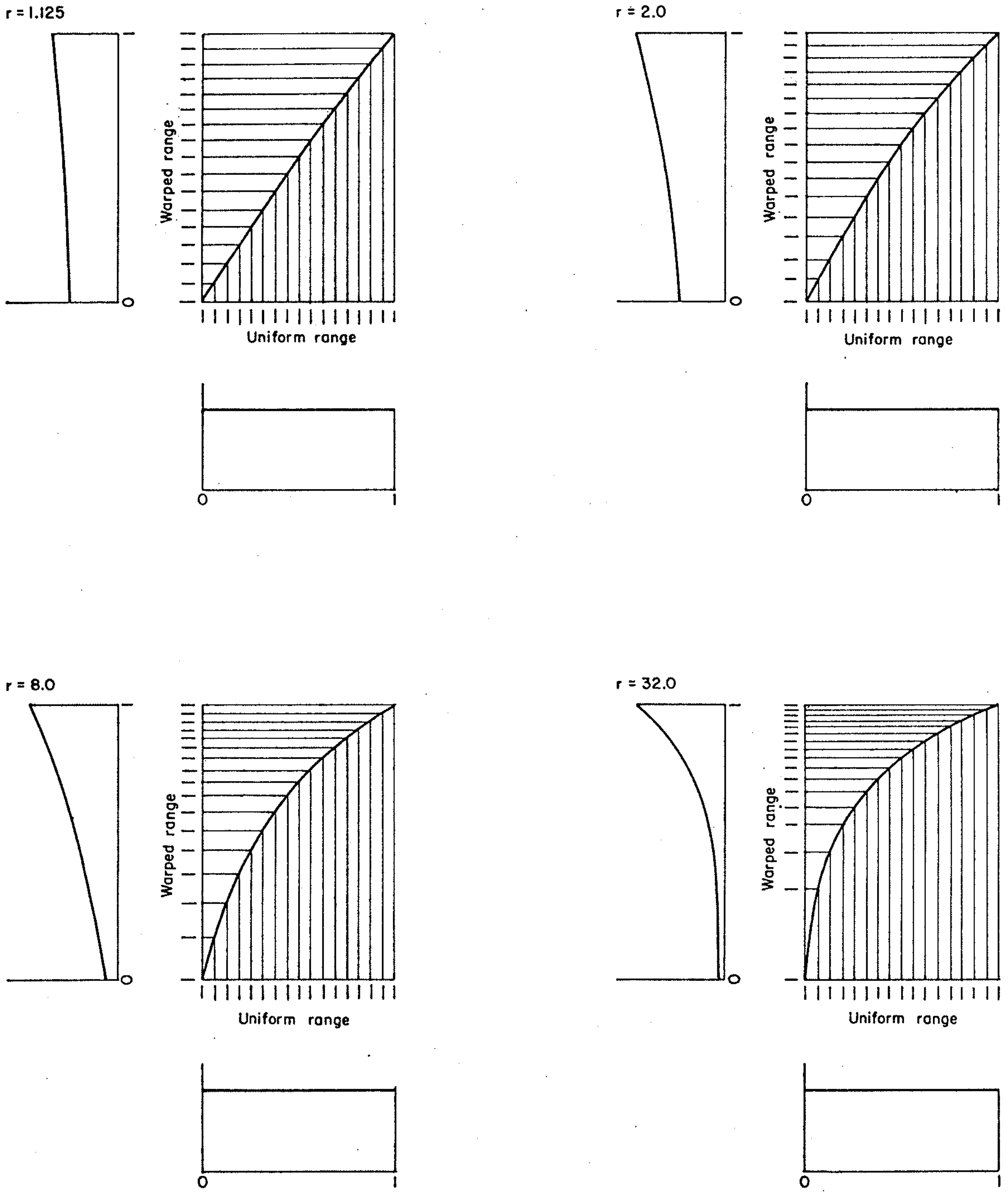


Fig 4-16

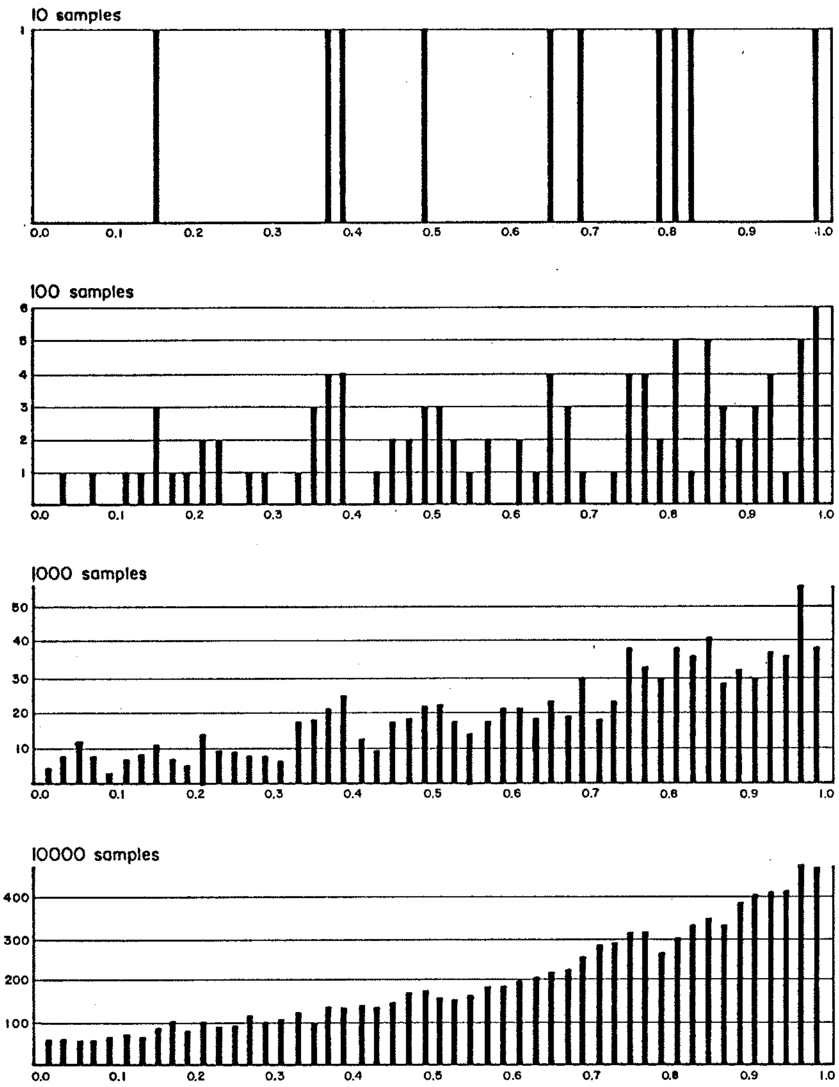

10 samples
1
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
100 samples
1
2
3
4
5
6
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
1000 samples
10
20
30
40
50
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
10000 samples
100
200
300
400
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0

Fig 4-17

sound sparse. By selecting a suitable value of r, however, one could apply Equation 4-8 to transform these registers in order to obtain a more satisfactory balance.

4.2.3.3 The Gaussian Distribution - This distribution was actually discovered by Abraham De Moive (1667-1754), though Carl Friedrich Gauss (1777-1855) did study it extensively. It is the classic "bell curve" of statistics, and is also called the normal distribution. Gaussian distributions are characteristic of scatterings of projectiles aimed at a target. The range stretches from negative infinity to positive infinity, and Equation 4-9 gives the density at any value x. The mean, u, gives the central point around which the distributed samples cluster. The deviation, d, gives the magnitude of scatter.

$$f(x) = \frac{1}{d*sqrt(2*pi)} e^{-[(x-u)/d]^2} \qquad \text{(Equation 4-9)}$$

Graphs of this density function for various values of d appear in Figure 4-18. While the range is theoretically unbounded, samples far from u become extremely improbable.

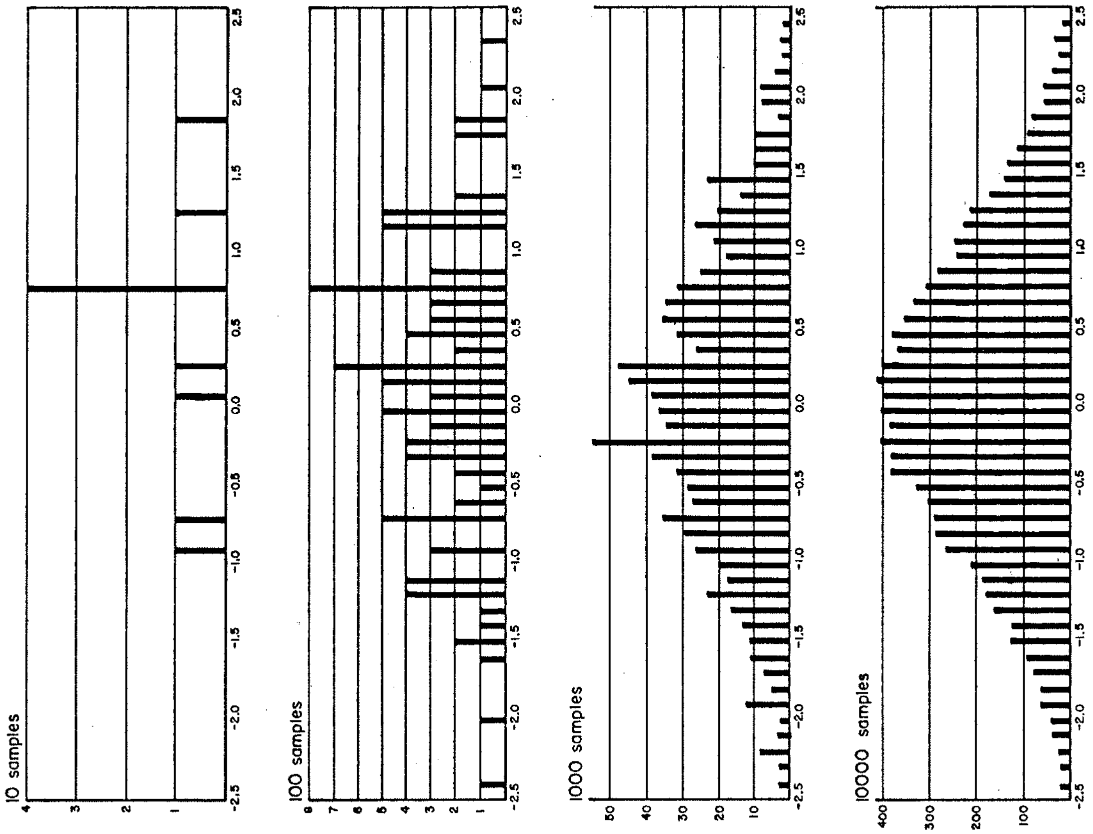


Fig 4-19

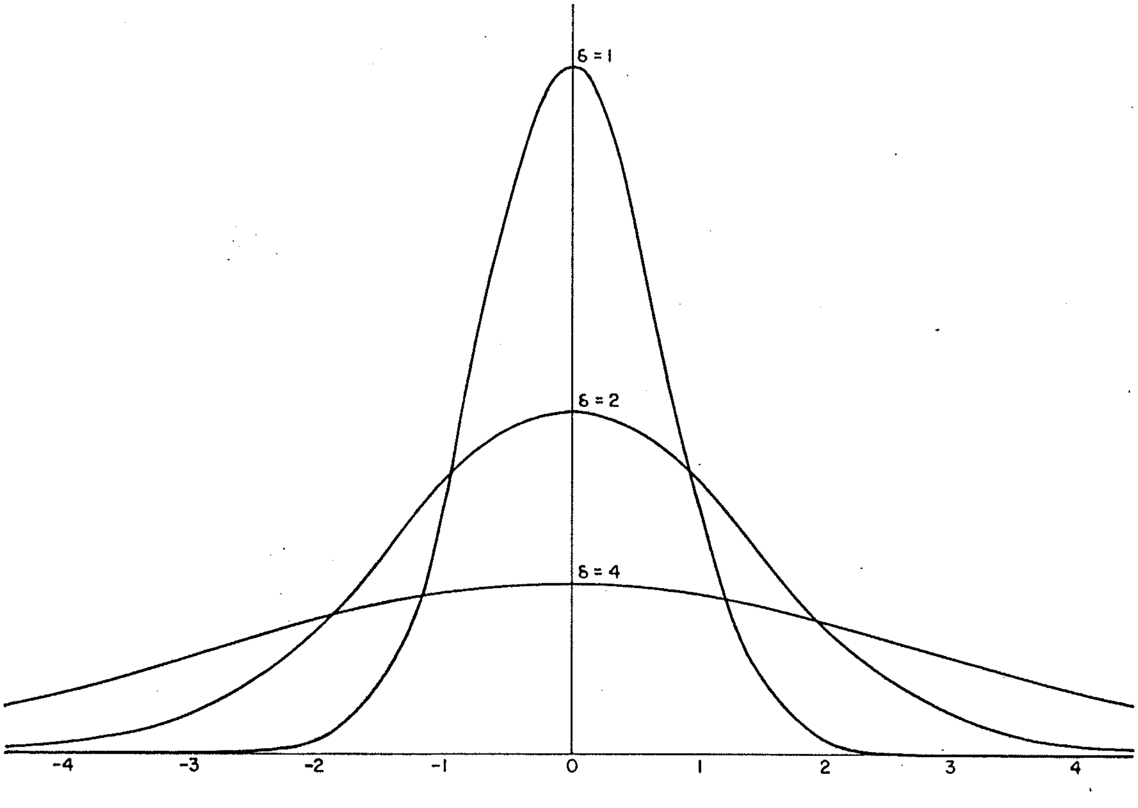


Fig A-18

Figure 4-18: Gaussian densities.

The Gaussian distribution relates to binomial distribution in the same way that the exponential distribution relates to the geometric distribution. Specifically, if we take n Bernoulli trials with probability of success p=1/2 and choose n large enough that the distinction between k out of n and k+1 out of n successful trials is negligible, then the distribution of k will approximate a Gaussian distribution with mean n/2 and deviation (1/2)*sqrt(n). The greater number of trials, the better the approximation, though as few as 5 (!) trials provides a good approximation.

There are several ways to produce Gaussian numbers. The simplest to program is the "polar" method devised by Box, Muller, and Marsaglia (Knuth, 1969, page 103). ~~([illegible]). Knuth gives the algorithm on page 103 of Seminumerical Algorithms.~~ The polar method produces two independent samples with each invocation. The real-valued library function GAUSS produces normally distributed numbers with mean AVG and deviation DEV. It invokes the polar method on every other call. Figure 4-19 presents histograms of samples produced by GAUSS.

-- Programming example 4-13: function GAUSS --

Fg 4-13

```
function GAUSS(AVG,DEV)
logical FLAG
data FLAG /.true./
if (FLAG) then
  do
    X = 2.0 * RANF() - 1.0
    Y = 2.0 * RANF() - 1.0
    S = X*X + Y*Y
    if (S.lt.1.0) exit
  repeat
  S = sqrt(-2.0*alog(S)/S)
  GAUSS = DEV * X * S + AVG
  FLAG = .false.
else
  GAUSS = DEV * Y * S + AVG
  FLAG = .true.
end if
return
end
```

Figure 4-19: Histograms of samples generated by GAUSS - Each bar tallies the number of samples falling within a region of width 1/10 for a mean u=0.0 and a deviation d=1.0.

4.2.3.4 Stored Continuous Distributions - Consider an arbitrary random incident over a continuous range whose density at each value x is given by f(x). What does f(x) mean? It does not give the probability that x will occur. Instead, f(x) gives us the probability that the random incident will produce an outcome near x. Looking more precisely at this interpretation, suppose x1 and x2 are two closely spaced values. The value midway between x1 and x2 will then be (x1+x2)/2. Equation 4-10 estimates the probability p(x1,x2) that the random outcome will lie in the region between x1 and x2.

$$p(x1,x2) \quad = \quad f[(x1+x2)/2] * (x2-x1) \qquad\qquad \text{(Equation 4-10)}$$

Equivalently stated, we can approximate p(x1,x2) by multiplying the density at the midpoint of the region by the region's width.

The accuracy of Equation 4-10 increases as the gap between x1 and x2 narrows.

This insight enables us to generate random numbers conforming to any continuous distribution, provided we know its density function. The process involves 1) dividing the range into small regions, 2) applying Equation 4-10 to determine a weight for each region, 3) using the methods for generating discrete random samples from stored distributions to select a region, and 4) selecting a sample uniformly from this region.

The library subroutine CURVE returns continuous random samples over a range divided into NUM regions. Similar in form to subroutine SELECT (Heading 4.4.1.5), it requres five arguments:

1. RESULT - CURVE selects a real sample between VALUE(0) and VALUE(~~4~~NUM) and returns the result in this location (line 16). RESULT must be real.

2. VALUE - ~~selected~~ user-supplied values along the range. VALUE must be a real array whose indices run from 0 to NUM. VALUE(0) must be low enough so that the densities at all smaller values are negligible. Similarly, VALUE(NUM) must be high enough so that the densities at all higher values are negligible. Intermediate values will usually be

Ex 4-14

```
      subroutine CURVE(RESULT,VALUE,WEIGHT,SUM,NUM)
      dimension VALUE(0:1),WEIGHT(1)
C     Generate scaled random number
      R = SUM * RANF()
C     Locate weighted region aligning with this number
      do (I=1,NUM)
        W = WEIGHT(I)
        if (R.lt.W) exit
        R = R - W
      repeat
      if (I.gt.NUM) stop 'Bad weights for CURVE.'
C     Interpolate between corresponding entries of array VALUE
      V = VALUE(I-1)
      RESULT = V + (VALUE(I)-V)*R/W
      return
      end
```

spaced evenly between VALUE(0) and VALUE(NUM).

3. WEIGHT - Array element WEIGHT(i) holds the probability of an sample falling between VALUE(i-1) and VALUE(i), as established by Equation 4-10. WEIGHT must be a real array of dimension NUM.

4. SUM - total of all NUM weights in array WEIGHT.

5. NUM - number of regions in the continuous range.

-- Programming example 4-14: subroutine CURVE --

4.3 RANDOM MUSICAL COMPOSITION

Randomness probably occurs much more frequently in creative decision-making than most composers and aestheticians would care to admit. However, the first documented method of composition relying explicitly on random decisions is the "Musical Dice Game" attributed to Mozart (a page appears in Cowell, 1952). This method enables its users "to compose, without the least knowledge of music, country dances, by throwing a certain number with two dice."

Random composition became fashionable in art music during the 1950's with John Cage. Cage threw the I Ching to decide which sounds should follow which in his Music of Changes for piano (1952; described in Cage, 1952, also Cowell, 1952). Cage saw an aesthetic appeal in music which truly eliminated the biases inherent in systematic procedures or in personal whims (note 5).

4.3.1 Lejaren Hiller and Leonard Isaacson: Illiac Suite

In order to compose the Illiac Suite for string quartet (1957), Lejaren Hiller and Leonard Isaacson programmed the Illiac computer to make uniform random choices and then to test these choices according to stylistic rules. Whenever a choice failed to meet all rules, it was discarded and a new choice was initiated (note 6). ~~They~~ Hiller and Isaacson justified the idea of 'filtering' random choices through rules by describing composition as "the extraction of order from chaos". An excellent overview of the four "experiments" which make up this work~~,~~ is available in Hiller and Isaacson's own summary, reproduced here as Table 4-3.

Illiac Suite Experiments Summarized

Experiment One: Monody, two-part, and four-part writing

A limited selection of first-species counterpoint rules used for controlling the musical output

(*a*) Monody: *cantus firmi* 3 to 12 notes in length
(*b*) Two-part *cantus firmus* settings 3 to 12 notes in length
(*c*) Four-part *cantus firmus* settings 3 to 12 notes in length

Experiment Two: Four-part first-species counterpoint

Counterpoint rules were added successively to random white-note music as follows:

(*a*) Random white-note music
(*b*) Skip-stepwise rule; no more than one successive repeat
(*c*) Opening C chord; *cantus firmus* begins and ends on C; cadence on C; B–F tritone only in VII$_6$ chord; tritone resolves to C–E
(*d*) Octave-range rule
(*e*) Consonant harmonies only except for $^{6}_{4}$ chords
(*f*) Dissonant melodic intervals (seconds, sevenths, tritones) forbidden
(*g*) No parallel unisons, octaves, fifths
(*h*) No parallel fourths, no $^{6}_{4}$ chords, no repeat of climax in highest voice

Experiment Three: Experimental music

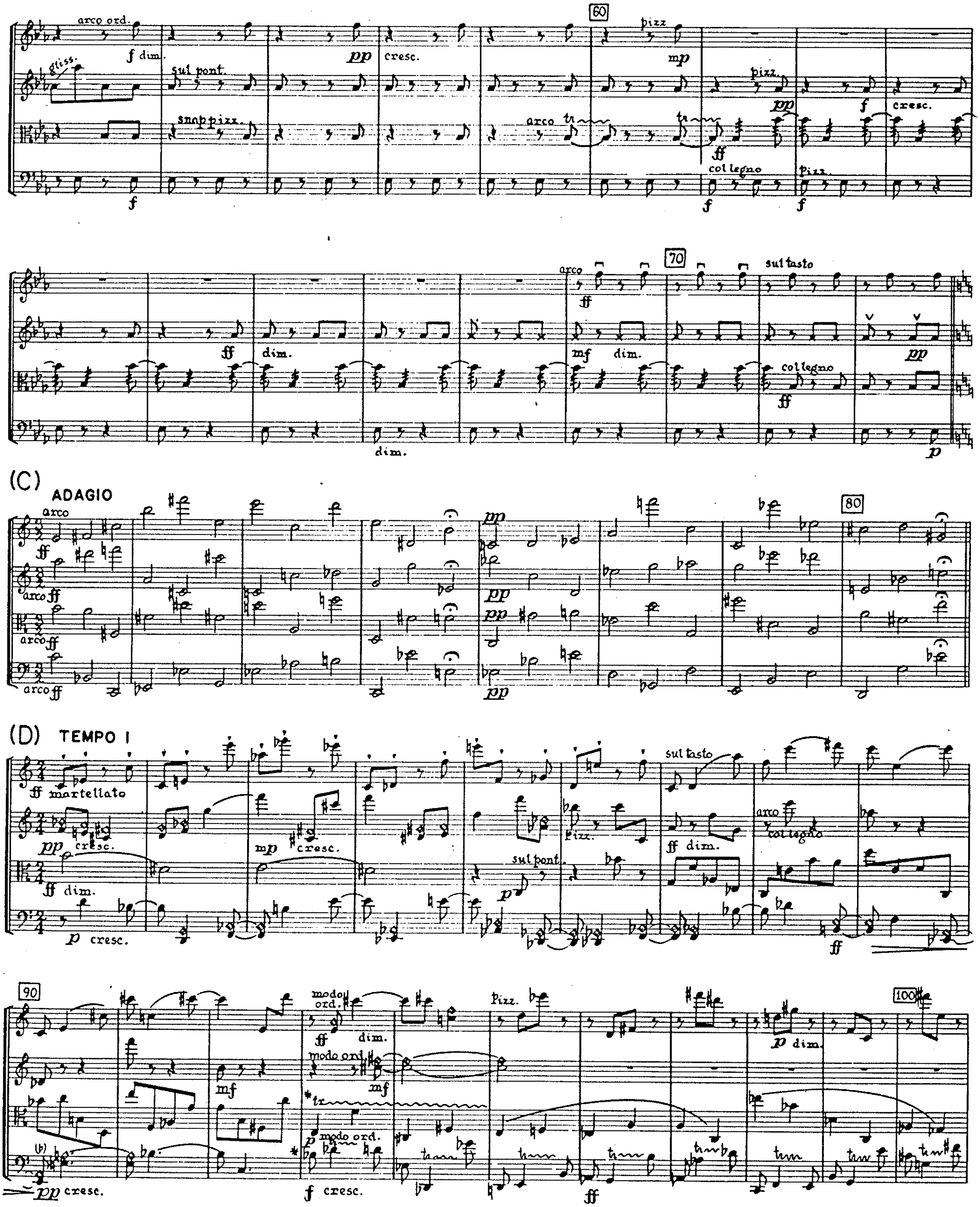
Rhythm, dynamics, playing instructions, and simple chromatic writing

(*a*) Basic rhythm, dynamics, and playing-instructions code
(*b*) Random chromatic music
(*c*) Random chromatic music combined with modified rhythm, dynamics, and playing-instructions code
(*d*) Chromatic music controlled by an octave-range rule, a tritone-resolution rule, and a skip-stepwise rule
(*e*) Controlled chromatic music combined with modified rhythm, dynamics, and playing-instructions code
(*f*) Interval rows, tone rows, and restricted tone rows

Experiment Four: Markoff chain music

(*a*) Variation of zeroth-order harmonic probability function from complete tonal restriction to "average" distribution
(*b*) Variation of zeroth-order harmonic probability function from random to "average" distribution
(*c*) Zeroth-order harmonic and proximity probability functions and functions combined additively
(*d*) First-order harmonic and proximity probability functions and functions combined additively
(*e*) Zeroth-order harmonic and proximity functions on strong and weak beats, respectively, and vice-versa
(*f*) First-order harmonic and proximity functions on strong and weak beats, respectively, and vice-versa
(*g*) *i*th-order harmonic function on strong beats, first-order proximity function on weak beats; extended cadence; simple closed form

Table 4-3



* WHOLE-TONE SHAKE

Fig 4-20

Table 4-3: Illiac Suite Experiments Summarized - Reproduced from Lejaren Hiller and Leonard Isaacson, Experimental Music (New York: McGraw-Hill, 1959), page 83. Copyright 1959 McGraw-Hill.

Figure 4-20: Lejaren Hiller and Leonard Isaacson, Illiac Suite (New York: New Music Edition, 1957), Experiment 3, measures 1-54 - Copyright 1957 New Music Edition.

4.3.2 Iannis Xenakis's Stochastic Music Program

Iannis Xenakis has preferred the term "stochastic" to "random" ("stochos" is Greek for "chance"). Xenakis introduced continuous statistical distributions for the first time into musical composition with the Gaussian "temperatures" of massed glissandi in Pithoprakta (1956; described 1971, p. 12) and later employed Poisson densities to arrange aggregates of sound in Achorripsis for 21 instruments (1957; described 1971, p. 22). For Achorripsis, Xenakis divided his score into a grid and used Poisson's formula (Equation 4-4) to construct tables detailing how many "events" could occur in each square of the grid. The statistical properties of the work match what would have resulted had the placement and content of each event been determined by random processes, yet Xenakis arranged his material entirely by hand.

The aggregate effect of these statistical scores concerned Xenakis more than specific relationships between elements, and in his 1962 ST program he began consigning specific decisions to the random number generator of a computer. The following synopsis of the this landmark program draws both from Xenakis's own description of the program (1971) and John Myhill's excellent critique (1978). The design of the ST program centers around two nested loops, an "outer" loop for sections and an "inner" loop for notes. Prior to initiating these loops, the ST program accepts directives from the composer (note 7) detailing the following information:

1. Length of composition.

2. Minimum and maximum "objective" densities of notes per second in any section - From this information the ST program derives a logarithmic scale of "subjective" densities which we present here in a simplified -- though equivalent -- formulation: given any "subjective" density X in the range from zero to one,

the ST program converts this "subjective" value into an "objective" density Y using the transformation:

$$Y = A * (B/A)^{X}$$

Where A and B are the minimum and maximum "objective" densities, respectively. Notice that X=0 returns Y=A while X=1 returns Y=B.

3. Average length of sections.

4. Definition of the orchestra - The composer organizes his ensemble into a collection of "timbre classes", each consisting of one or more instruments. One and the same instrument used to produce different types of sounds (such as sustained notes, tremelos, or glissandi) must be included under multiple "timbre classes". Once this scheme of organization has been established, subsidiary information must also be provided:

   a. For a given number of points equally spaced from zero to one along the scale of "subjective" densities (item 2 above), the composer must indicate the relative percentages of "timbre classes" which the ST program is to employ at these given densities.

b. For each instrument in each timbre class, the composer must indicate a relative percentage of notes in the timbre class which should employ the specified instrument. He must also indicate lowest and highest pitches for the instrument (not necessarily corresponding to the instrument's full range) along with a maximum duration.

With each iteration of the "outer" composing loop, the ST program first determines attributes of a section and then initiates iterations of the "inner" composing loop in order to fill out this section with notes. The "outer" loop selects three attributes for sections:

1. Durations of sections follow an exponential distribution around the average sectional duration supplied by the composer (item 1 of the previous list).

2. Densities of sections follow a uniform distribution along the "subjective" scale of densities from 0 to 1 (item 2 of the previous list). However, "a certain

concern for continuity" leads Xenakis to inhibit more drastic contrasts using a strategy which John Myhill explains in the following way. Suppose the "subjective" density of the most recent section was X'. Then in order to select a "subjective" density X for the current section, the ST program first initiates a Bernoulli trial with probability 1/2 of success (that is, it flips a coin) to determine whether X will be smaller or larger than X'. If X will be smaller than X', the program selects two random values x1 and x2 uniformly between 0 and X', then sets:

$$X = X' - |x1-x2| .$$

Otherwise if X will be larger than X', the program selects two random values x1 and x2 uniformly between X' and 1, then sets:

$$X = X' + |x1-x2| .$$

According to Myhill, the average distance between X and X' using this strategy is 1/6. By contrast, the average distance between two independent random samples distributed uniformly between 0 and 1 is 1/3 (note 8).

The "subjective" density is then converted into an "objective" value using the formula described above.

3. Once it has determined a "subjective" density, the ST program then uses this value to determine what proporations of "timbre classes" should occur at this density. It does this by consulting the percentages supplied by the composer for the two nearest points along the density scale (item 4a of the previous list) and then interpolating (Chapter 8) between these percentages to create a table of weights effective throughout the current section.

When these sectional attributes have been established, the ST program begins composing notes. Each note is described by six attributes:

1. The starting time of a note is calculated relative to periods between consecutive attacks. Periods are distributed exponentially around a mean value which is the reciprocal of the "objective" density established for the current section (item 2 of the second list above).

2. The instrument upon which a note is played is determined by first randomly selecting a "timbre class" using the table of weights established for a section (item 2 of the second list above) and next randomly selecting an instrument within this "timbre class" using the secondary table provided by the composer for each "timbre class" (item 4b of the first list above).

3. To determine the pitch of a note, the ST program employs a strategy similar to the one used to select "subjective density". In this instance, the program consults the most recent pitch played by the given instrument, decides whether to move upward or downward from this pitch, and then uses the lowest and highest pitches indicated for the instrument by the composer (item 4b of the first list above) to determine how far to move.

4. If the "timbre class" admits glissandi, then the ST program selects a glissando speed by a Gaussian distribution. The mean of this distribution is always zero (that is, there will on the average be just as many downward-moving glissandi as upward-moving ones); the deviation may increase with density, decrease with density, or vary independently of density, but in any

case the deviation holds constant throughout a section.

5. Durations also follow Gaussian distributions. Given the density of notes per second for the current section (item 2 in the second list above) and the relative likelihoods of each instrument in each "timbre class" at this current density (obtained from the tables provided in items 4a and 4b of the first list above), the ST program performs some rather elaborate calcuations in order to determine two quantities: 1) the average distance Z between attacks *by the current instrument* in the current section and 2) the distance Zmax between attacks the same instument in this instrument's *slowest* section. The program reconciles these two quantities with the longest duration G playable on the instrument (item 4b of the first list above) using the formula:

$$Z' = G * \frac{\ln(Z)}{\ln(Zmax)}$$

Notice that the formula returns Z'=0 when Z=1, returns Z'=G when Z=Zmax, and acts to foreshorten Z' for intermediate values of Z. For the formula to be meaningful, it is necessary to express Z and Zmax relative to the shortest possible note duration. The absolute minimum note length allowed by the ST program is 1/10 of a second (quintuplet eighths at 𝅗𝅥 =60), which would require scaling both Z and Zmax by a factor as large as 10 (note 9). The ST program generates a random duration using a Gaussian distribution with mean Z'/2 and deviation Z'*sqrt(2)/4; durations are limited on the lower end by the shortest note duration and on the higher end by Z'.

6. A final attribute selected for each note is an "intensity form". Xenakis admits five basic forms in all variants employing the dynamics ppp, p, f, and ff:

   a. Steady dynamics (e.g., steadily ppp),

   b. Crescendi (e.g., ppp-cresc-f, p-cresc-ff),

   c. Diminuendi (e.g., ff-dimin-f, ff-dimin-ppp),

   d. Swells (e.g., pp-cresc-f-dimin-pp, p-cresc-ff-dimin-p), and

ST/4-1

1 violon, 1 v/c, 1 c B, 1 piano

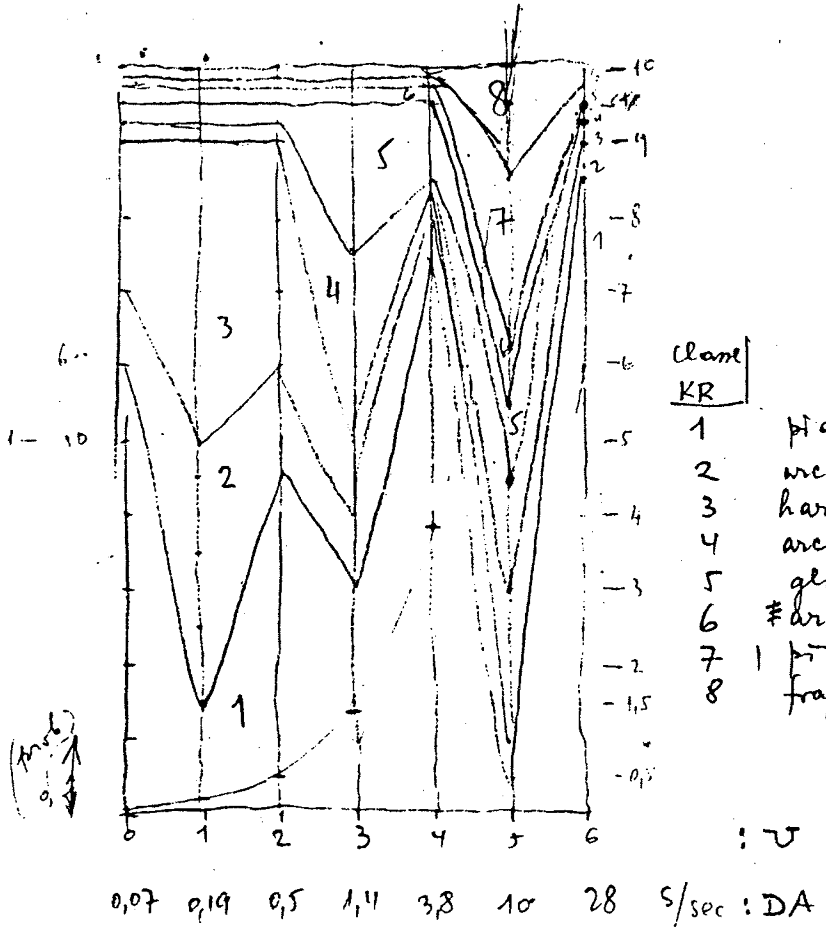
1 violon	4	s/sec
1 V/C	4	s/sec
1 C/B	3	s/sec
1 pian	15	s/sec
DA_{max}	26	s/sec

$V3 = 0{,}07$ KTE = 7

$\max DA = 0{,}07 \cdot e^{6} = 28{,}3$ s/sec

$\max DA_{cordes} = 0{,}07 \cdot e^{5} = 10{,}4$ s/sec

$\max DA_{piano} = 0{,}07 \cdot e^{5{,}5} = 17{,}2$ s/sec



Classe KR	
1	piano
2	arco ponticello
3	harmoniques
4	arco norm
5	gliss
6	# arco ponticello
7	1 pizz
8	frapp col legno

$V3 \cdot e^{v} = DA$

Density = 5.3)

JW9 Density = 0.7

JW10 Density = 3.0

JW11 Density = 0.4

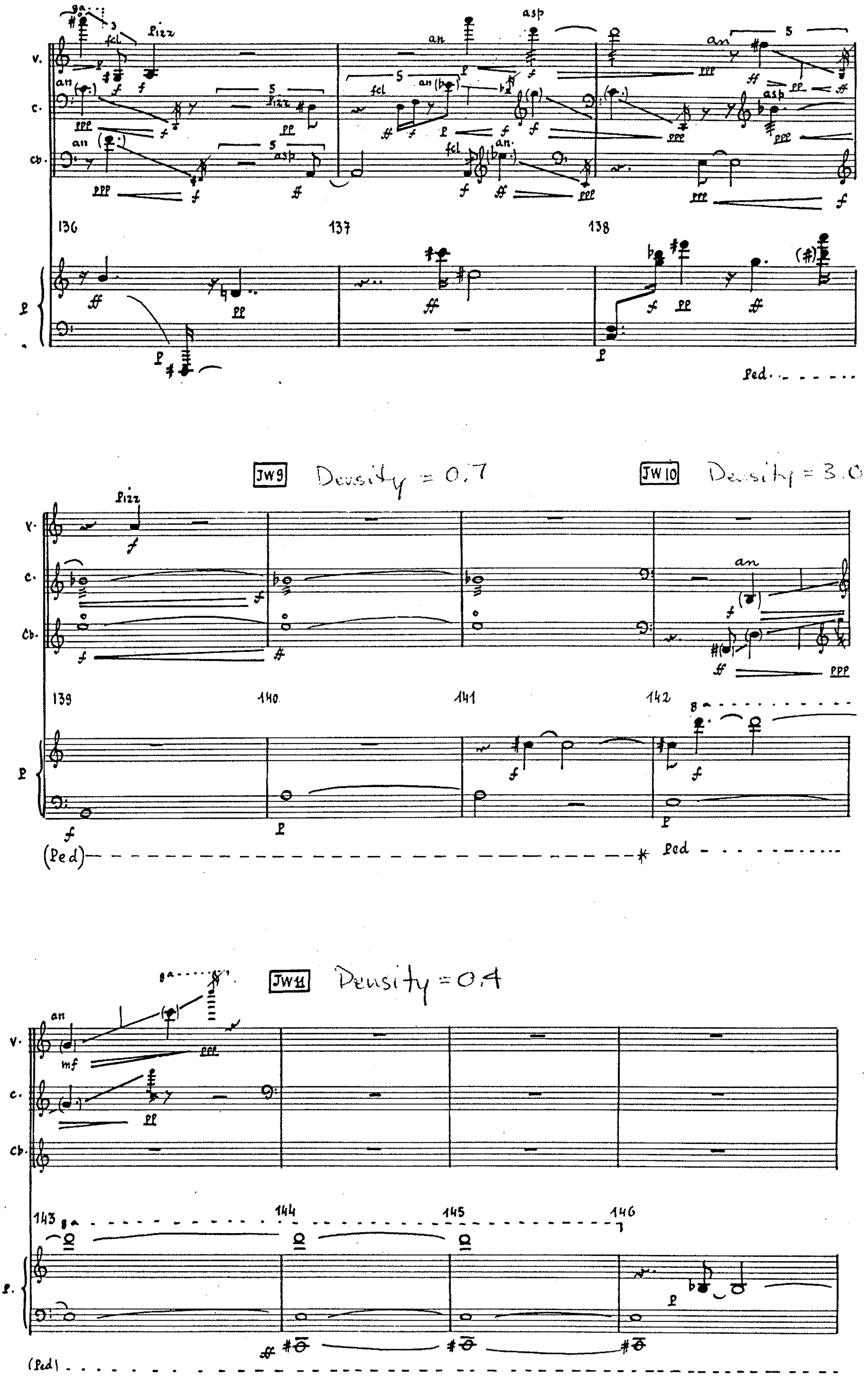


Fig 4-21

e. Inverted swells (e.g., ff-dimin-f-cresc-ff, ff-dimin-ppp-cresc-ff).

The total number of variants is 44. The ST program chooses uniformly from among these variants.

Upon completing the ST program in 1962, Xenakis used it to compose several works for varied ensembles including ST/10-1,080262 for 10 instruments, ST/48-1,240162 for large orchestra (48 instruments), and Morisma-Amorisma (ST/4-1,030762) for violin, 'cello, contrabass, and piano. A relatively active excerpt from this last composition is reproduced here as Figure 4-21.

Figure 4-21: Iannis Xenakis, Morsima-Amorsima (1962), measures 115-123. The ~~indications~~ abbreviations in the string parts are: asp - arco sul pont.; an - arco posit. norm.; fcl - frappe col legno. Copyright 1967 Boosey and Hawkes.

Each "JW" mark ~~indicates~~ signifies the first complete measure of a section. Indicated densities ~~are values~~ are those selected by the ST program.

4.3.3 James Tenney: Stochastic String Quartet

James Tenney was programming the computers at Bell Telephone

4-42a

STOCHASTIC STRING QUARTET

James Tenney
1/20/63

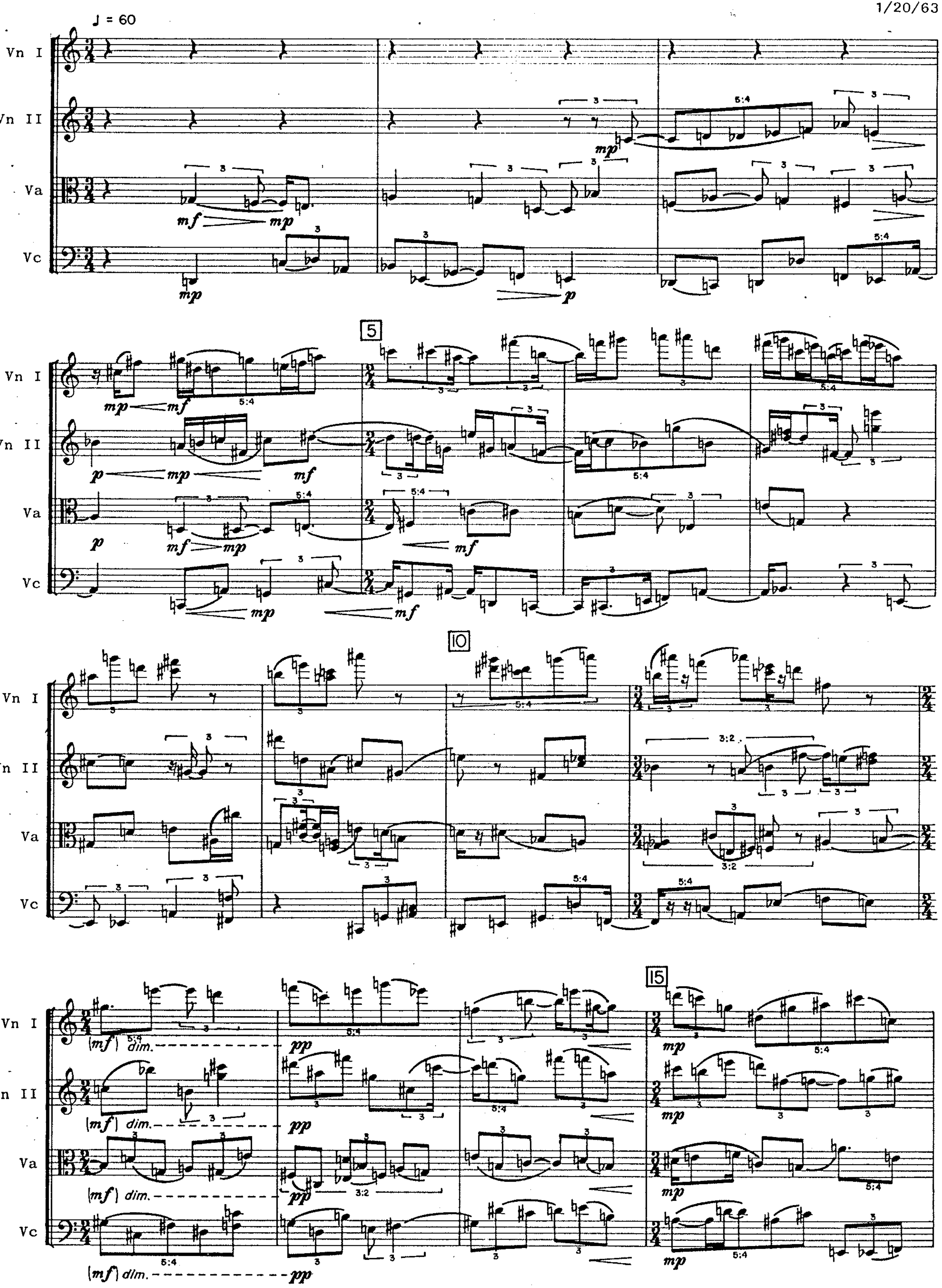


Fig 4-22

Laboratories to create statistical compositions roughly at the same time as Xenakis was writing his ST program (Tenney, 1969). Though unaware of Xenakis's program, Tenney was well aware of Xenakis's manually composed scores such as Pithoprakta and Achorripsis. Tenney's programs differ from ST in that Tenney usually retains direct control over the large-scale "direction" of a piece. Where the ST program holds such musical attributes as densities and proportions of "timbre classes" fixed over each section, Tenney also often allows one or more musical attributes to evolve gradually. (Such gradual evolutions will be discussed in greater detail in chapter 8). Where the ST program implements only three levels of musical structure -- composition, section, and note -- Tenney implements several intervening levels.

Figure 4-22: James Tenney: Stochastic String Quartet, measures 1-15 - Copyright 1963 James Tenney.

Most of Tenney's composing programs generated files of numeric data which ~~was~~ were then realized ~~directly~~ by digital synthesis. One exception which admits direct examination of the score is a short piece called the Stochastic String Quartet (1963). This piece divides into three sections, which at the indicated tempi last 52, 102, and 37 seconds. Intervening between the section and its component notes is an intermediate structural unit which Tenney calls a "clang", while the long-term "direction" is determined by parametric graphs detailing how various musical attributes evolve over time. The attributes under Tenney's control include "clang durations", average periods between attacks, ranges of pitches, dynamics, and several aspects of timbre: vibrato, tremelo, "spectrum" (sul tasto, ord., sul pont.), and "envelope" (pizz., arco-staccato, arco-legato, arco-marcato, arco-sforzando). Each graph supplies a mean value to one or more random automata in order to select attributes for a specific "clang"; in turn, the program feeds these "clang" attributes into further random automata in order to select attributes for a note.

An especially significant feature of the Quartet is Tenney's elaborate procedure for composing rhythms. Not content to approximate his rhythms as displacements relative to a simple meter such as 4/4 (what has become the usual approach) Tenney exploits bracketed rhythmic proportions directly as a compositional resource. His program employs a scheme of triply nested loops:

> Main loop: For each "clang", the program chooses several attributes including a "clang duration" consisting of approximately two to five quarter notes and a mean period between attacks.

First nested loop: For each instrumental part (violin I, violin II, viola, 'cello), the program chooses a range of pitches, and a number of "gruppetto" units. This second number need not match number of quarters in the "clang"; for example, if the "clang" lasts for 3 quarters and the number of "gruppetto" units is 5, then the result will be a bracketed rhythm in the proportion of 5:3.

Second nested loop: For each "gruppetto" unit in the current part, the program selects a number of rhythmic divisions.

Third nested loop: At the innermost level, the program steps through the divisions of each "gruppetto" unit. With each step it compares the period of whatever musical event (single note or double stop) is being played by the current instrument to the minimum period established for the entire "clang". If the event's period is still to short, the program increments it; otherwise, the program initiates a new event.

The likelihood of an event being a double stop versus a single note is controlled by a graph which ranges between complete certainty of a single note and complete certainty of a double stop. Tenney employs Hiller and Isaacson's procedure to subject pitches to a non-repetition rule: pitches are selected at random, but any pitch which has recently been used by the current instrument is discarded in favor of a new random attempt.

4.3.4 Herbert Brun: Non-Sequitur VI

Figure 4-23: Herbert Brun: Non-Sequitur VI (1966), measures 1-8 - Copyright 1966 Herbert Brun.

Herbert Brun used random selection in Non-Sequitur VI. (Description forthcoming.)

4.3.5 Barry Truax's POD Program

Figure 4-24: Graphic depiction of input data to Barry Truax's POD program.

In its original (1973) version, Barry Truax's POD program implements procedures similar to those used manually by Xenakis

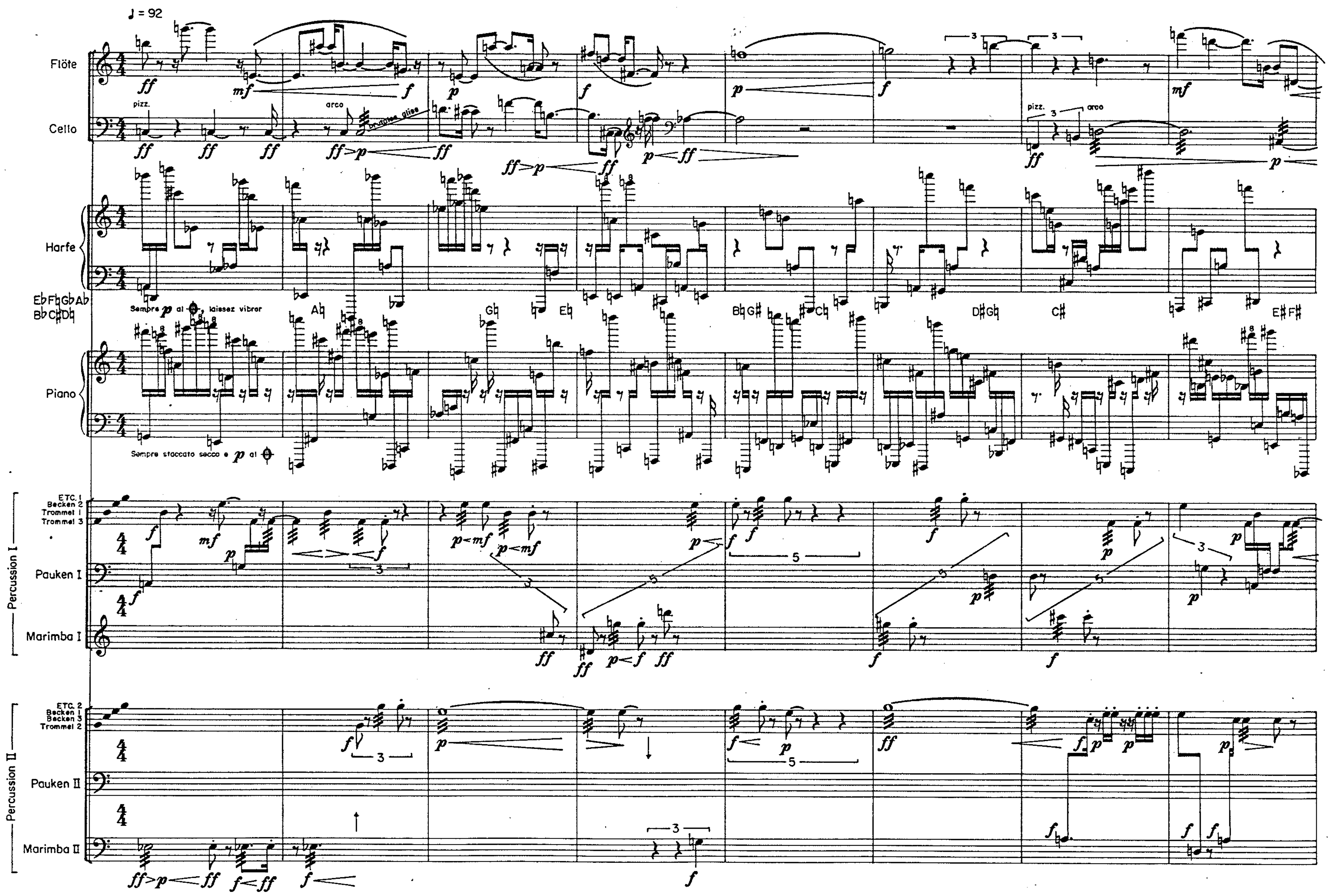

♩ = 92
Flöte
Cello
Harfe
Eb F♮ Gb Ab
Bb C♮ D♮
Piano
Sempre p al ⊕, laissez vibrer
Sempre staccato secco e p al ⊕
Percussion I
ETC. 1
Becken 2
Trommel 1
Trommel 3
Pauken I
Marimba I
Percussion II
ETC. 2
Becken 1
Becken 3
Trommel 2
Pauken II
Marimba II
pizz.
arco
glissando gliss
A♮
G♮
E♮
B♮ G♯
C♮
D♯ G♮
C♯
E♯ F♯

Fig 4-23

4-46a

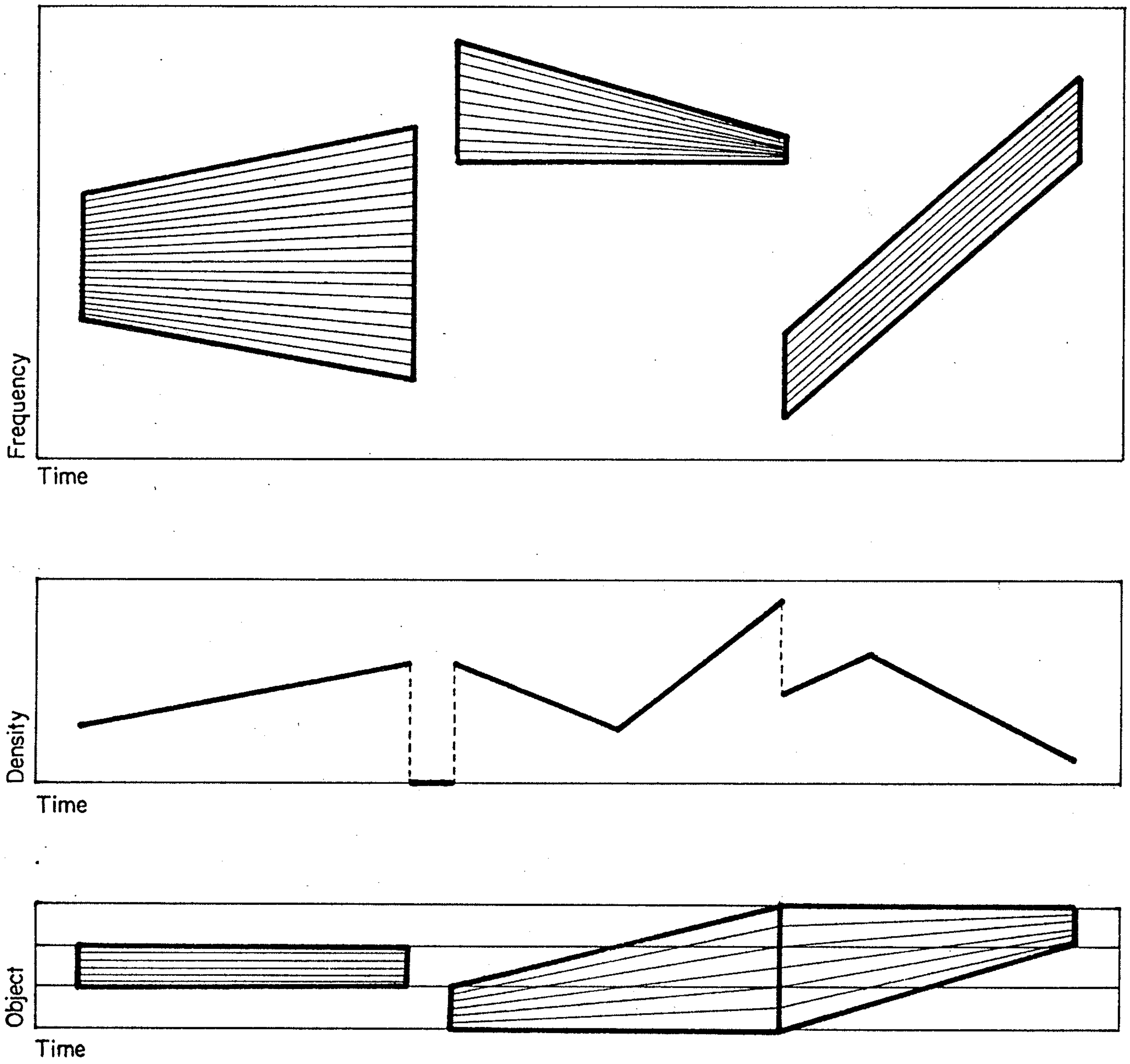


Fig 4-24

to compose Achorripsis. "POD" in fact stands for "POisson Distribution". The user of POD describes a musical composition using three graphs controlling 1) densities of sonic events in time, 2) lowest and highest frequencies, and 3) an "object" number which usually determines the timbre of a sound. Figure 4-24 illustrates the form of the graphs of densities and frequencies graphs, which are composed from elementary segments. Each segment in the density graph is described by an initial density, a final density, and a duration; likewise, segments of the frequency graph, which Truax calls the "frequency-time mask", are described by initial and final ranges of frequencies along with a duration. Segments of the density graph need not necessarily coincide with segments of the frequency graph. Once the three graphs have been completely described, POD implements the following procedures:

1. POD divides the time-axis into 1-second units and interpolates (Chapter 8) along the density graph to determine the number of events in each unit.

2. From the frequency graph, POD constructs a frequency-time "trapesoid" lasting one second. Upon this trapesoid, POD superimposes a grid in which each column lasts 0.01 seconds and in which the number of

elements in a column is fixed so that each element covers only a very small fraction of the total range of frequencies. POD then employs Poisson's formula (Equation 4-4) to distribute sonic events around the grid.

3. For each sonic event, POD selects an "object number". Truax's method of selecting objects adapts the TENDENCY feature of Gottfried Michael Koenig's PROJECT2 program, the mechanism of which is described in Chapter 8.

The results of these computations can be either performed in real time or stored as score files. Though Truax's procedures have strong antecedents in earlier programs by Xenakis and Koenig, POD provides the flexibility of an interactive editor for creating and modifying the three parametric graphs. Trux developed his his original system for personal use, using it to produce several compostions including Trigon (1975) and Nautilus (1976). In his 1978 version of POD, Truax expands the system's generality by allowing users to edit and merge score files in a "second level" of processing. This version is responsible for compositions by Truax such as Androgeny (1978) and Arras (1980); this version has gained some use by composers other than Truax himself. ~~A new~~ The newest version, ~~called~~ PODX, incorporates an even greater degree of sophistication, incorporating "contitional editing" features along with routines for spatial placement. Included among Truax's compositions produced using PODX are Wave Edge (1983).

4.4 DEMONSTRATION 2: RANDOM SELECTION

Demonstration 2 illustrates the use of random decision-making to compose a piece of music. Like its predecessor, Demonstration 2 has a three-tiered structure in which phrases occupy an intermediate level between the piece as a whole and the local details. From this point on, however, all resemblance ceases. Where the compositional directives for Demonstration 1 completely describe the musical results, direct compositional control over Demonstration 2 is limited to specifying a repertory (or range) of options for each musical attribute involved and prescribing ~~designating~~ a relative weight for each option. All further decision-making at both intermediate and local levels of structure is delegated to the random-number generator.

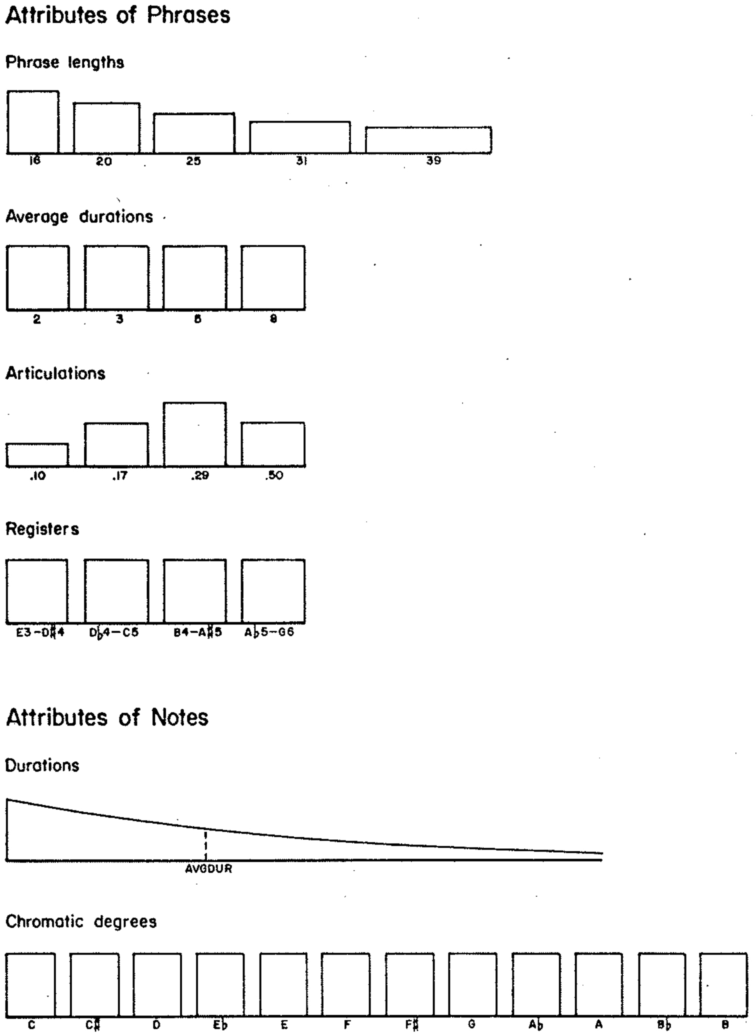

Attributes of Phrases
Phrase lengths
16
20
25
31
39
Average durations
2
3
5
8
Articulations
.10
.17
.29
.50
Registers
E3-D♯4
D♭4-C5
B4-A♯5
A♭5-G6
Attributes of Notes
Durations
AVGDUR
Chromatic degrees
C
C♯
D
E♭
E
F
F♯
G
A♭
A
B♭
B

Fig 4-25

4.4.1 Compositional Directives

Figure 4-25: Compositional Data for Demonstration 2.

We may regard each phrase of Demonstration 2 as a string of consecutive rhythmic 'units', where an individual unit can be either a note or a rest. Phrases are distinguished by four randomly selected attributes: 1) length of phrase, 2) average duration of notes (equivalently, average tempo), 3) articulation, and 4) register; probability distributions affecting how each of these is selected appear depicted in Figure 4-25.

Articulations express the probability that a rhythmic unit within the current phrase will be a rest instead of a note. If the unit is a rest, then its duration will center around an average half as large as the average duration characterizing notes. Two consecutive notes with no intervening rest are slurred; two consective notes separated by a rest of null duration are tongued.

For the purposes of Demonstration 2, "register" indicates a <u>gamut</u> of twelve adjacent semitones which holds fixed though a phrase. If a unit is a pitch, then the program first selects one of the twelve chromatic degrees and next places this degree within the appropriate register.

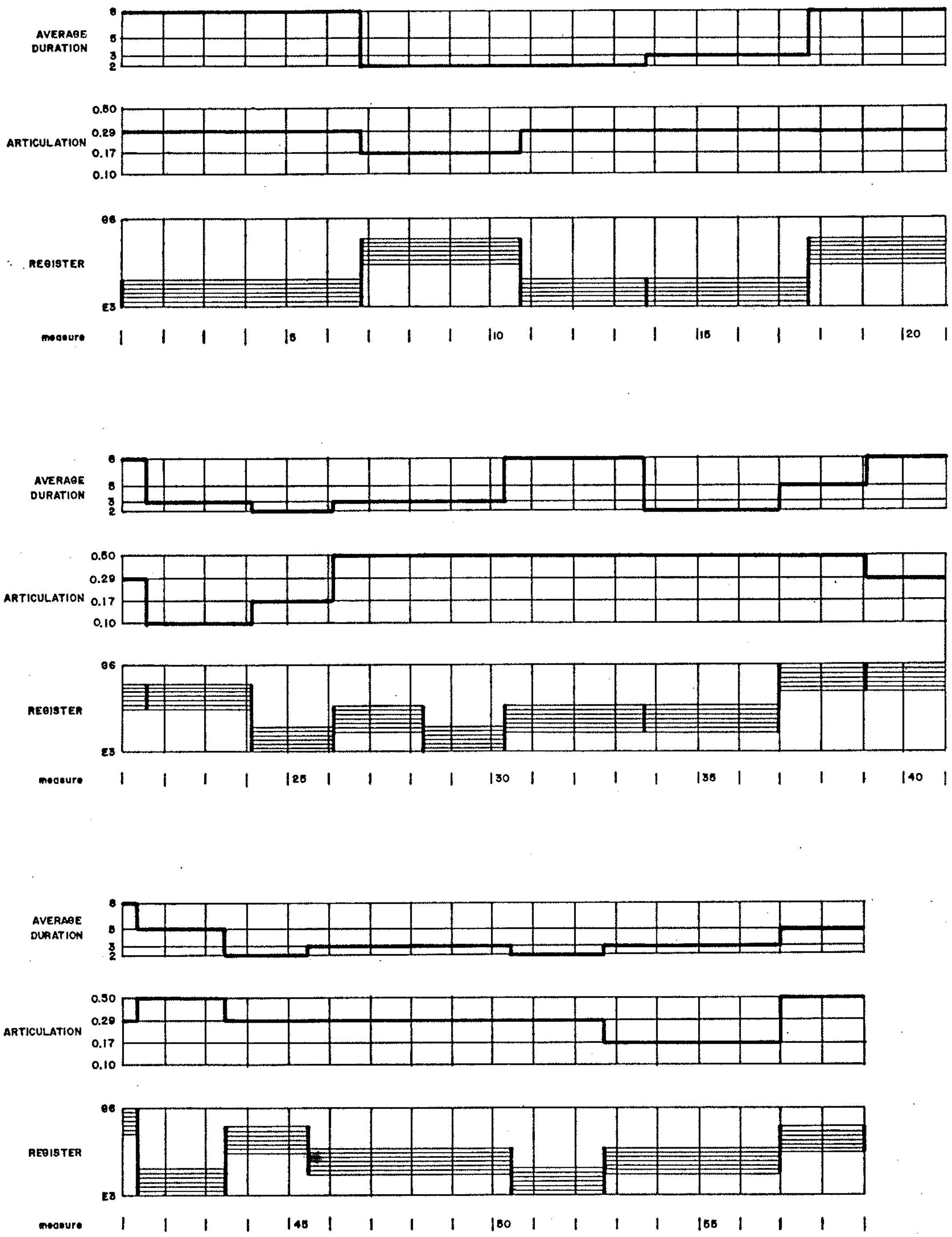

AVERAGE DURATION
8
5
3
2
ARTICULATION
0.50
0.29
0.17
0.10
REGISTER
G6
E3
measure
5
10
15
20
25
30
35
40
45
50
55

Fig 4-26

4-50b

Demonstration 2

Clarinet

Charles AMES

STRICTLY ♩ = 80

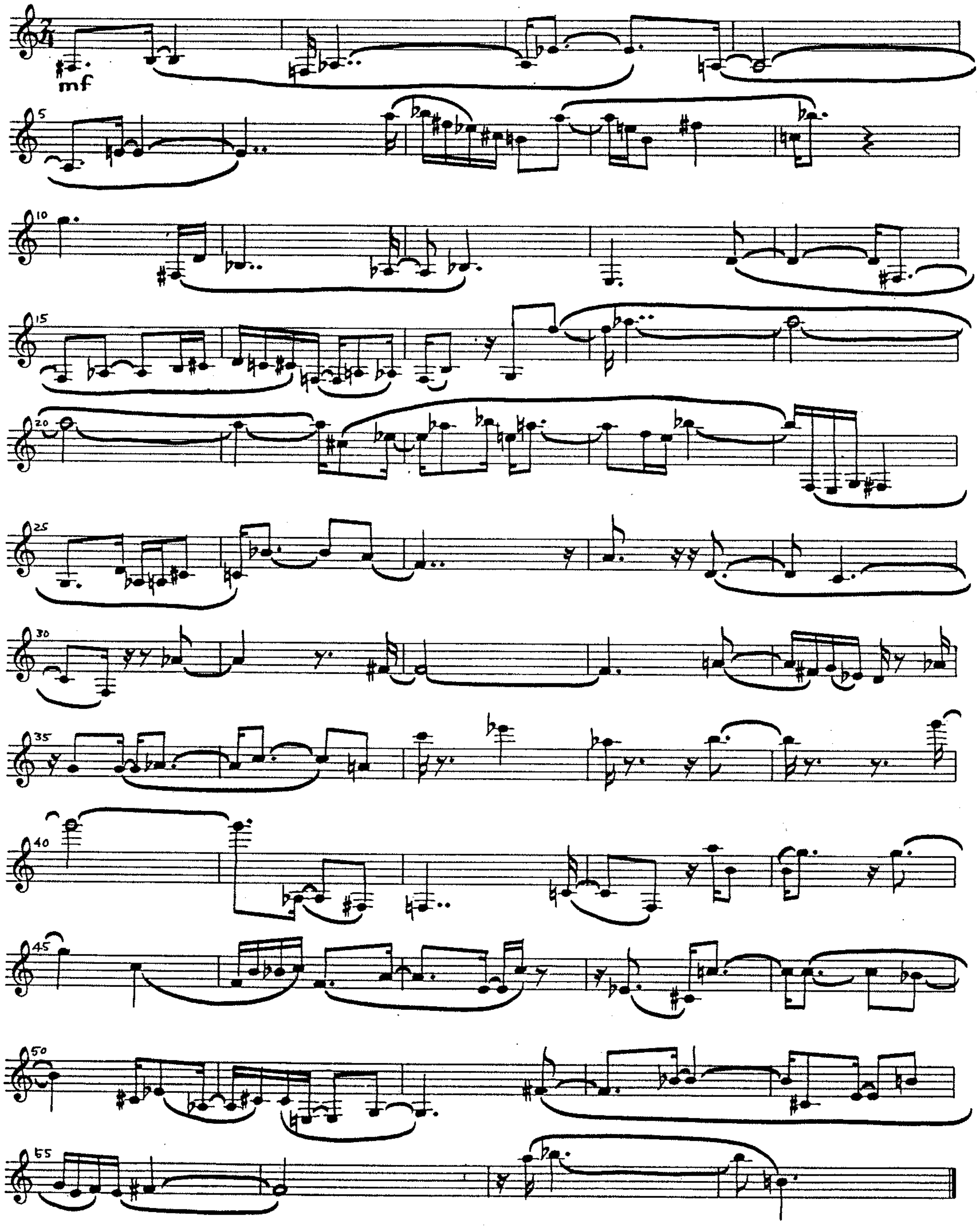


Fig 4-27

In addition to depicting compositional data for phrases, Figure 4-25 also depicts the exponential distribution used to select durations for notes (rests are half as long) and the uniform distribution used to select degrees. Attributes selected for phrases are depicted graphically in Figure 4-26, while the complete product is transcribed in Figure 4-27.

Figure 4-26: Profile of Demonstration 2.

Figure 4-27: Transcription of Demonstration 2.

4.4.2 Implementation

-- Programming example 4-15: program DEMO2 --

Following the model of Iannis Xenakis's ST program, program DEMO2 reflects the three-tiered musical structure described above as a design of nested loops. The bulk of DEMO2 proper (lines 27-41) constitutes the "outer" composing loop; this loop establishes the attributes for phrases depicted in Figure 4-26.

```
      program DEMO2
C
C     Demonstration of random selection
C
      parameter (MPHR=5,MAVG=4,MART=4,MREG=4)
      integer VALREG(MREG),VALPHR(MPHR)
      real VALAVG(MAVG),VALART(MART)
      real WGTPHR(MPHR),WGTART(MART)
      data VALPHR/  16, 20, 25, 31, 39/,
     :     WGTPHR/.625, .5, .4,.32,.26/,
     :     SUMPHR/2.105/
      data VALAVG/2.,3.,5.,8./
      data VALART/.1,.17,.29,.5/
     :     WGTART/1.,2.,3.,2./
     :     SUMART/8./
      data VALREG/40.,49.,59.,68/
C
C     Initialize
C
      open (2,file='DEMO2.DAT',status='NEW')
      ITIME = 0
      MTIME = 8 * 60
C
C     Outer composing loop
C
      do
C       Select duration of phrase
        call SELECT(IPHR,VALPHR,WGTPHR,SUMPHR,MPHR)
        KTIME = ITIME + IPHR
C       Test for end of composition
        if (KTIME.gt.MTIME) exit
C       Select average duration for notes in phrase
        call ALEA(AVGDUR,VALAVG,MAVG)
C       Select probability of rest
        call SELECT(ARTIC,VALART,WGTART,SUMART,MART)
C       Select register
        call ALEA(IREG,VALREG,MREG)
C       Compose phrase
        write (2,10) IPHR,AVGDUR,ARTIC,IREG
   10   format ('Phrase duration:',I4,'  Average duration:',F7.1,/,
     :           'Articulation:',F7.3,'  Lowest pitch:',I3)
        call PHRASE(KTIME,ITIME,AVGDUR,ARTIC,IREG)
      repeat
      close (2)
      stop
      end

      subroutine PHRASE(KTIME,ITIME,AVGDUR,ARTIC,IREG)
C
C     Inner composing loop
C
      logical SUCCES
      data REMAIN/0./, IDEG/0/
      do
C       Select note or rest (no two consecutive rests)
        if ( IDEG.eq.0 .or. .not.SUCCES(ARTIC) ) then
C         Select duration of note
          DUR = RANX(AVGDUR,1000.0) + REMAIN
          IDUR = max0(1,ifix(DUR+0.5))
          REMAIN = DUR - float(IDUR)
C         Select degree
          IDEG = IRND(12)
        else
C         Select duration of rest
          DUR = RANX(AVGDUR/2.0,1000.0) + REMAIN
          IDUR = ifix(DUR)
          REMAIN = DUR - float(IDUR)
C         Null degree indicates rest
          IDEG = 0
        end if
C       Write note or rest
        call WNOTE(ITIME,IDUR,IDEG,IREG)
C       Test for end of phrase
        if (ITIME.ge.KTIME) exit
      repeat
      return
      end
```

Ex 4-15

At the end of each iteration of the "outer" composing loop, DEMO2 calls subroutine PHRASE. This call invokes an entire cycle of iterations by an "inner" composing loop (lines 5-26 of PHRASE), each of which iterations composes a note or rest.

The symbols of DEMO2 proper adhere to four mnemonic 'roots' corresponding to attributes of phrases:

1. PHR - length of phrase.

2. AVG - average duration of notes (equivalently, average tempo); the average duration of rests is half as large.

3. ART - articulation, expressed as the probability that a rhythmic unit will serve as a rest.

4. REG - register, expressed as the lowest pitch in a twelve-semitone gamut.

The number of options available to each attribute is given by a parameter starting with the letter M. The values reside in arrays beginning with VAL, their associated weights reside in arrays beginning with WGT, and the sum of these weights is held by a variable beginning with SUM.

The methods used to implement the different random decisions vary with the distributions depicted in Figure 4-25. DEMO2 asks the library subroutine ALEA to select average durations (line 34) and registers (line 38) since these decisions provide equal likelihoods for each option. PHRASE selects chromatic degrees directly by asking the library function IRND to select a uniform random integer between 1 and 12 (line 13). So long as no two rests occur consecutively, PHRASE settles the question of whether a rhythmic unit should serve as a note or as a rest by asking the library function SUCCES to conduct a Bernoulli trial (line 7). DEMO2 asks the library subroutine SELECT to provide durations of phrases and probabilities of rests according to stored distributions (lines 29 and 36). Durations for individual notes and rests follow exponential distributions provided by the library function RANX (lines 9 and 16 of PHRASE).

Remember that exponentially distributed samples occur over a continuous range. The variable REMAIN (lines 9-18 of PHRASE) enables PHRASE to reconcile exponentially distributed durations with the discrete rhythmic neumes of musical notation. REMAIN resembles an "accumulator": Each time PHRASE computes IDUR from DUR, it stores the fractional part in REMAIN. (In line 11, the lower bound of 1 precludes generation of notes without durations. The increment of 0.5 causes PHRASE to approximate DUR as the next larger integer when its fractional part exceeds 1/2. In such cases, the amount added to REMAIN is negative.) The next time it computes DUR, PHRASE adds REMAIN to the new random value. Fractional parts of consecutive random values accumulate until their sum exceeds unity; when this happens an additional sixteenth bleeds off into IDUR.

The tasks of subroutine WNOTE in DEMO2 differ from those of its counterpart in DEMO1 in that pitch information takes the form of a chromatic degree (1 to 12) and a register. Registers are expressed in semitones above C0 and indicate the lower bound of the octave which is to hold the final pitch. The integer library function IOCT performs the necessary conversion from registers to octaves:

-- Programming example 4-16: function IOCT --

4.5 NOTES

1. Mathematicians regard a <u>function</u> as a "rule of association"; while it is generally most economic to distill functions into formulae, tables such as Table 4-2 suffice.

2. Where not explicitly indicated otherwise, the motivations for the random distributions described under heading 4.2 are drawn

Ex 4-16

```
function IOCT(IDEG,IREG)
IOCT = (IREG-IDEG)/12 + 1
return
end
```

from George Roussas's A First Course in Mathematical Statistics (1973). Similarly, the algorithms for generating random samples conforming to these various distributions are drawn from Chapter 3 of Donald Knuth's Seminumerical Algorithms (1969).

3. The exclamation points in Equation 4-2 indicate taking *factorials*. The factorial of 0 is 1; for an arbitrary integer M, we define M! as follows:

$$M! = 1 * 2 * 3 * \ldots * (M-1) * M$$

For examples, 1! = 1, 2! = 2, 3! = 6, 4! = 24, and so on.

4. Myhill's 1979 article describes his method with reference to a "degree of non-periodicity" which is in fact equal to the natural logarithm of the argument PROPOR described here. Function RANX incorporates Myhill's own formula for deriving the values R1 and R2 directly from PROPOR.

5. Pierre Boulez's term "aleatoric music" (Boulez, 1957) is often used to indicate scores such as Morton Feldman's Projection I for solo 'cello (1950) and Boulez's own Third Piano Sonata (1957) which leave decisions to the discretion of the performer. This term ("alea" means "dice") is misleading; performers are

not random.

6. Hiller and Isaacson's strategy has the disadvantage that rejecting a choice does not preclude it from being reconsidered in a later attempt. A much more direct solution to the problem would have been to assemble a schedule of all available choices (possibly in random order) and then to step through this schedule systematically until a suitable choice had been discovered. This alternate solution is much faster and has the additional capability of determining when <u>no</u> available choice is acceptable. It is discussed more exhaustively in chapters 5, 7, 9, and 14.

7. Though many composers have employed Xenakis's program, only Xenakis's own compositions have acheived critical acknowledgement.

8. Xenakis's method of inhibiting large leaps between consecutive random samples was first described by the mathematician Emile Borel (1871-1956).

9. Xenakis unfortunately neglects such a conversion, which results in his formula producing negative durations when Z falls short of unity.

4.6 RECOMMENDED READING

Arkin, Herbert, and Raymond Colton. Statistical Methods (New York: Barnes and Noble, 1934. Fifth edition, 1970.

Huff, Darrell. How to Lie with Statistics (New York: W.W. Norton, 1954).

Cowell, Henry. "Current Chronicle: New York", Musical Quarterly, volume 38, number 1 (January 1952), p. 123.

Myhill, John. "Some simplifications and improvements in the stochastic music program", Proceedings of the 1978 International Computer Music Conference (Computer Music Association, 1979).

Xenakis, Iannis. Formalized Music (Bloomington: Indiana University Press, 1971).