An Unusual Pseudorandom Number Generator Program

H.T. GORDON
College of Natural Resources
University of California
Berkeley 94720

Many computer programs can generate a sequence of pseudorandom 8-bit binary numbers. Ideally, such a sequence requires that any number in the sequence have an equal probability of being followed by an other number (including itself). The ideal of perfect unpredictability of the next number may be unattainable, since this can always be predicted with complete certainty if both the generator algorithm and the numerical "keys" it will use are known. However, some algorithms can produce long sequences that satisfy statistical tests of randomness.

The goal of my work was to devise logic that would create the illusion of randomness, using a minimum of code and execution time. The starting point was my reading of the (13N + 1) algorithm by Daniel Greiser in the Nov 1977 *BYTE*, my first encounter with simple, fast algorithms of this kind. Since then, my conception of the problem has undergone many sea-changes. I have come to the point where I can no longer see how more could be done with less. It is time for fresher, keener minds to look at the problem!

Linear Core Algorithms

One requirement is that (in a long sequence) each of the 256 binary numbers occur at equal frequency. This is met by many simple algorithms of the form: $N_{i+1}=(4a+1)\ N+c$, where c must be an odd number. If $a=0$, the sequences are obviously non-random, and can easily be "deciphered" from a very short sequence. If $a=1$, the sequence has a much more random look. This (5N + 1) algorithm is the basis of my new subroutine SIMRND (650X-coded in the listings). My original thinking-out of this type of algorithm (in a MS sent to *BYTE* a year ago but not yet published) was in error, and caused me to use the unnecessarily complex (5N + $2B) algorithm, which works but codes less efficiently.

The SIMRND output is a sequence of the 256 different binary numbers which endlessly repeat. As with all linear algorithms, the "randomness" is in the high-order bits. If one writes the sequence as a 16 x 16 matrix, the low nibbles show a characteristic pattern: the first row has a sequence of all 16 possible low nibbles (with alternating odds and evens which are repeated exactly in each succeeding row (so each column has 16 identical low nibbles).

Evaluation of the Degree of Disorder in Short Numerical Sequences

If any number N_1 has an equal probability (1/256) of being followed by *any* other number N_2, all values of the *difference* $(N_2 - N_1)$ have a probability of 1/256. One way of analyzing a short sequence uses the concept of *heterogeneity*. The *zero-order* heterogeneity, H_0, is that of the sequence itself. It can be defined as the number of *different numbers* occurring in a sequence of 256. The maximum H_0 is of course 256. The *first-order* hereogeneity, H_1, requires a sequence of 257 numbers, from which the first-order sequence of 256 differences between successive numbers is calculated. The maximum H_1 is again 256. The concept and operation can be extended to H_2, H_3, etc. An interesting theoretical problem is to what extent a large value of H_0 is compatible with large values of H_1, H_2, etc. Another problem which I lack mathematical skill to solve is what values of H have the highest probability; although there exist an incredibly large number of different sequences with $H_0 = 256$, there are even more sequences with values of 255 or less, so it seems unlikely that a truly random sequence will have a maximum H_0.

To test the output of various algorithms, I wrote a KIM-1 program that generates sequences of increasing orders of heterogenity and calculates the approximate H values. I do not present the listing here because it is too specialized to be of general interest, but anyone who sends me an SASE can have a photocopy of it. The test results for 5 different linear algorithms are given in Table 1. (This also includes the effect of various "jumbling" routines, to be described in the next section of this note). All have an $H_0 = 256$. The simple (N + $7F) algorithm, of course, has $H_1 = 1$, the minimal value (it must remain so for all higher orders). The more complex algorithms are much more heterogeneous. The (5N + 1) and (13N + 1) have identical values, declining from 64 at H_1 to 1 at H_4, showing a fairly complex kind of order has been generated. The (9N + 1) is less heterongeneous, and the (17N + 1) still less. It is clear that the most useful algorithms are in fact of the type $(8n+5)N + 1$, where n is any number from 0 on up. Nevertheless, there is an underlying orderliness, quickly unraveled by the test program.

The Creation of Disorder by Jumbling Algorithms

In my (rapidly obsolescing!) BYTE MS, I introduced a second algorithm to "jumble" the output of the core algorithm. The underlying principle is that one can define *subsets* of the set of binary numbers, and *operations* that uniquely transform each number in a subset into another number in the same subset. The effect is to *relocate* these numbers in an order different from that in the original sequence.

My first try used the concept of *parity* to define 2 equal subsets: one in which bit 7 = bit 6, and one in which bit 7 ≠ bit 6. Every subset in which 2 (or any even number of) bits have the same (odd or even) parity has the useful property that the *complement* of every number is the number that, when itself complemented, will be the original number. An algorithm which recognizes numbers in which bit 7 ≠ bit 6, and complements all of them, will *exchange* pairs of complementary numbers within the original sequence. My original coding of this algorithm was very inefficient, using the 650X BIT instruction and the status of the N and V flags to identify numbers with bit 7 ≠ bit 6. In my new coding (the COMJUM module in the listings) this is done by a single CMP #$C0 instruction, that sets the N flag if the number in the accumulator has bit 7 ≠ bit 6. The significance of the "sign" flag in COMPARE instructions is seldom well explained in programming manuals, so it is used less often (and sometimes incorrectly) than the carry and zero flags in decision-making. For any given comparand, the N status subdivides the 256 possible bit-patterns in the accumulator into 2 equal sets. The comparand $C0 sets the N flag only if the accumulator contains a number in the range from $40 to $BF, which is the set of all numbers in which bit 7 ≠ bit 6.

When I first used the "jumbling" concept, I had not thought it out properly, and so did not realize that *the core algorithm itself* defines 2 equal subsets, since the N flag is set only by the 128 numbers with bit 7 = 1. If one uses an EOR #$7F instruction, to avoid altering bit 7, one can "exchange" number-pairs in one of these 2 subsets in which the 7 low-order bits are complementary. This is the simple logic of the SIMJUM module in the listings. Its effect on the heterogeneity of the output of various core algorithms is shown in Table 1, and is *identical* with the effect of the COMJUM module. Used with the ultrasimple (N+$7F) algorithm, it gives a remarkably high value of H_1. However, the output of the jumbled algorithm is not all that disorderly, as shown by the rapid decline in higher-order H values. Most of the high H_1 signifies a more complex order. Even if one goes to H_{32} the value does not decline to 1 (and even shows a very gradual, erratic rise).

With the more complex algorithms, SIMJUM creates a similar large increase in disorder (insofar as H values define it), most of it persistent in higher-order H analysis. This is obviously not *true* disorder, since a simple "dejumbling" of the output would allow my H test program to resolve it. However, it may be a good *simulation* of true disorder.

Since jumbling based solely on the status of bit 7 had such an interesting effect, I also explored bit 6 by the JUMSIX algorithm (cf. listings). As shown in Table 1, jumbling on the bit 6 status has a very small effect on the (N+$7F) algorithm (however, it has a much larger one on the (N+$3F) algorithm, with an H_1 of 65). With the more complex algorithms, JUMSIX creates less heterogeneity than SIMJUM, especially on the ones on which SIMJUM is most effective.

Having at last recognized that each bit defines a pair of equal subsets which are (in principle) independent, I was able to see that the CMP #$C0 of the CONJUM logic actually defines *four* subsets. One way of defining these and causing some exchange of complementary pairs in all four of them is shown as module TETJUM in the listings. The operands of the 3 EOR instructions can be altered, but the effects are minor. However, the effect on heterogeneity is not markedly different from that of SIMJUM.

A quite different way of defining and altering the original sequence is module ROLJUM (cf. listings). Again there is no greater heterogeneity than with SIMJUM. The jumbling information in bits 7 and 6 seems not to be independent.

The H_1 value of 187 differences (in Table 1 for the (5N+1) algorithm with SIMJUM) is not the possible maximum. The RISJUM module (cf. listings) gives an H_1 of 205 differences with (5N+1), but successive values decline to 64, 63, 16, 16, 4, 4, and 2 (the stable value), again proving that a high H_1 may simply indicate very complex order. If the SIMJUM module is used first, to introduce "stable disorder", then followed by RISJUM, there is usually a small enhancement of the H values. Note that the SIMJUM EOR operand, however, must be changed from $7F to $7E, so as not to alter either bit 7 or bit 0; the reason is that RISJUM is jumbling on the status of bit 0 (moved into the carry flag by the LSR A instruction). It is vital (to retain an H_0 of 256) that the bit or bits on which jumbling is based *not be changed* by the operation. The reason why RISJUM alone cannot introduce significant stable disorder is that bit 0 is the least disorderly bit in the zero-order sequence, alternating between 0 and 1. However, I now doubt that *any* multiple-jumbling will be worth the cost.

A friend pointed out to me that the sequence: N, N+1, N-1, N+2, N-2 . . . . . N+$80 will contain all 256 different numbers and 255 of the possible differences (except 0); since the 257th number will be equal to the 256th, H_1 must be 256. I generated and tested this sequence and found successive H values of 256, 64, 64, 16, 16, 4, 4, and 1. The extreme orderliness of this sequence is revealed by adding SIMJUM to the generator. H_0 is 256, but H_1 crashes into 3 (and eventually to a stable value of 2). A very high H_1 value is compatible with (and may well *indicate*) extreme orderliness! So the *higher* orders of H seem to be a better criterion.

I have emphasized the concept of orders of heterogeneity because it can (and will) be used to resolve the underlying order in pseudorandom sequences. Dejumbling tools will be needed as well. Whatever has been raveled by rigid logic can be unraveled (though it may take a vastly more complex program to do so). I have dealt only with the simplest logic which can be unlocked with keys of very few bits. With more elaborate logic and keys, an infinite variety of pseudorandom sequences can be generated. While a deciphering program could soon recognize the pseudorandomness, recreating the generator logic might be extremely difficult!

Creating Sequences Longer than 256 Numbers

In my original design, I used 2 more zero-page "working registers" (in addition to the RND seed location): one worked as a "same-sequence-of-256" counter, incremented at every call to the subroutine, while the other was an "addend", incremented only when the counter became zero (i.e., at the start of each new sequence). The content of the "addend" location

was added to the output of the core algorithm before jumbling of the accumulator content, to give the final output number. This added 9 program bytes and 15 microseconds execution time per number generated. Since the primitive version was both code-efficient and time-inefficient, this seemed trivial. The new SIMRND-SIMJUM coding needs only 14 program bytes and executes in 31 microseconds, so that the simpler INCRND module (cf. listings) seems preferable since it adds only 6 program bytes and 8 microseconds. It also lengthens the non-repeating sequence to 65K. Both the slower "addend" operation and the INCRND operation would be easily decipherable. The former has the desirable feature that every one of the 190-odd "differences" follows every possible binary number (a good "randomy" touch), but a good deciphering program would soon recognize output (N+256) is generally output (N)+1. While the INCRND operation does not have this flaw, the decipherer would soon see every output number is usually (not always, because of jumbling) followed by a completely predictable number. In both, the fact that many first-order differences *never* occur, while many others occur at twice the expected frequency, would let a lot of cats out of the bag!

Both of the surface flaws (but not the deeper one of low heterogeneity) are corrected fairly simply in subroutine DUBRND (cf. listings), at the cost of 5 program bytes and 5 microseconds execution time. This can be viewed as a sequence of 4 modules: DUBRND, a minor complication of INCRND which decrements a new zero-page "addend" working register (in addition to incrementing the RND seed) at the start of each new sequence; SIMRND; ADDRND, which starts the "variation" of the output by adding the sequence addend; and SIMJUM. While there are still only 256 different sequences before exact repetition occurs, every one has a different initial seed and a different addend. The decipherer might need a bit more time to figure out what was going on, since the addend operation produces (when the addend is an odd number) sequences that no linear core algorithm could yield.

Nothing would be easier than to produce greater complexity and much longer non-repeating sequences. But *why?* Unless one could correct the heterogeneity flaw, this would not create a puzzle of genuine interest to cryptologists. This is a good point to argue the value of speed and aperiodicity in pseudorandom generators (code efficiency, always nice if you can get it, seems less important here). If used with a fast D/A converter to simulate "white noise", SIMRND alone would cause output fluctuations at frequencies near 60,000 Hz, but its short non-repeating sequence means periodicity near 400 Hz. The INCRND-SIMRND combination would lower the peak frequency to near 40,000 Hz, but would have an exact-repetition frequency of less than 1 Hz. (Some added coding might be needed to equalize the timing of each operation; this would lower these values slightly.) A repeating sequence of 256 may be too short, while one greater than 65K may be too long.

Improved heterogeneity, not length of the non-repeating sequence, is the challenge to theoretical minds of a higher order than my own!). Given enough complex logic, I have no doubt that a much closer approximation to true randomness than that achieved by my simple logic is attainable, especially if the keys are initialized (something not required by my logic). But if this involves a large increase in execution time, it will be useful only to the few who possess superfast machines. It seems to me that mere statistical tests are not searching enough to reveal the orderliness in short sequences; many routines that (with long sequences) pass such tests with flying colors would fail the heterogeneity test.

Decimalizing the Pseudorandom Sequence

This can be done simply by a module like SELDEC (cf. listings), which allows only the 100 "natural" decimal numbers (00 to 99) to be output, while rejecting the 156 numbers that have a hex value in either high or low nibble. Although SELDEC needs only 13 program bytes, the mean execution time per decimal number output is (2.56 T+32.6) microseconds, where T is the mean execution time of the binary generator being used. Even if the binary generator is SIMRND alone, the execution time per decimal number output is over 70 microseconds, and the timing penalty rises quickly as the binary generator becomes more complex. Also, the length of the non-repeating sequence is only 25K.

The more complex (25 program byte) module DECRND is much faster, with a mean execution time per decimal number output of (1.28 T+24.4) microseconds. Even with SIMRND alone, where its advantage is minimal, the mean time is less than 45 microseconds, and increases only to 65 microseconds with the complex DUBRND/SIMRND/ADDRND/SIMJUM generator logic (that with SELDEC requires over 110 microseconds per decimal output).

The operation of DECRND outputs 200 decimal numbers (each repeated once, but in a non-repeating sequence of 200) for every 256 binary numbers output by the basic generator. It first detects the 160 binary numbers that have a decimal value in the high nibble and passes these through a decimal-adjust module. All the "natural" decimals emerge unaltered. The "unnaturals", 60 numbers of the type (0-9)(A-F) emerge "adjusted" to duplicates of type (0-9)(0-5). The task of the SPECOP module, to which the 96 numbers of type (A-F)(0-F) were sent, is to reject 56 and convert the other 40 into the "missing" duplicates of type (0-9)(6-9).

SPECOP clears the carry and does 2 left-rotates through the carry. It can now reject the 32 numbers of the original type (A,B)(0-F), that have cleared the carry. With 2 more left-rotates, the previously low nibble (0-F) is now the high, and the low nibble is (6,7) with carry either set or clear. One of the (6,7) sets is accepted, the other is converted to (8,9) by an EOR #$0E. Finally it uses a CMP #$A0 to reject the 24 numbers with a hex high nibble and allows the 40 of type (0-9)(6-9) to exit.

DECRND does not output all the 100 different decimals in any sequence of 100 consecutive numbers (H_0 is near 50). Some numbers occur once, others are duplicated, and many are absent; this is corrected in the next 100 numbers. It is not only much faster than SELDEC, but (with a 65K binary generator) yields a non-repeating sequence of 51K decimals. One can concatenate successive outputs to form large numbers of $2n$ decimal digits; if n is odd, wraparound allows generation of 51K different large numbers before exact repetition. The degree of "stable disorder" is probably higher than that with SELDEC, and perhaps even higher (as a percentage of the maximum) than that of the binary generator since decimalizing is a kind of jumbling. I have not tested this because I am not much

interested in decimals. For the many who are, DECRND may be useful. For me, the interest lies in the unusual (less easily decipherable) conversion of hex to decimal numbers, a process far more complex than my hex-jumbling modules, which hints to really elaborate jumbling operations being able to create higher levels of heterogeneity.

Modules, Superinstructions, and Macros

All are blocks of code, goal-oriented to produce a desired effect, and *insertable* into programs that require the effect. Insertion saves the time lost by a subroutine call and return. The word "module" is quite general. Macros tend to be rather big modules. I view my simpler modules as "superinstructions". In fact, the idea for the RISJUM module came from an earlier exploration of the emulation in 650X code of useful instructions missing in the set. The ancestor of RISJUM used an EOR #$80, and emulated a rotate-right-without-carry operation. I presume that emulator programs use this kind of thing to convert code from one kind of instruction set to another.

The creation and recognition of modular structure in programs allows many variants of an operation, each fitting the particular need for speed, code-efficiency, or complexity in the unending game of trade-offs. Many programs contain well-written (and well-hidden) modules to do various tasks, whose value as optimized superinstructions usable in other programs is given no emphasis. Similar tasks in other programs are often badly coded (unless the programmer re-invents the optimal coding). A rich module/subroutine library would make programming both easier and more efficient.

Copyrights

I am copyrighting the listed modules (and their combinations) with the same "free-diffusion" clause used for my program EDITHA (*DDJ* #25). Note that the f-d-c allows totally unrestricted use in association with all f-d-c software, but does not *forbid* its use in programs protected by the usual copyright. However, in such programs, f-d-c software is entitled to protection *equal* to that of the associated software (use only by prior permission). I am aware that a lot of unacknowledged use of software is going on. As an advocate of diffusion, I do not disapprove of this. I *would* object to a "double-standard" in which "borrowers" incorporated f-d-c software into their programs without permission, then made waves if someone else "borrowed" *their* work. This raises the question: what about coding of these algorithms for other micros than the 650X? You cannot copyright an algorithm. Thousands of people could easily code them for the 8080 etc.—who, if anyone, can then claim an exclusive copyright?

Table 1. Heterogeneity Numbers (H_1-H_4) for Linear Core (+ Jumbling) Algorithms

CORE	alone	SIMJUM	RISJUM	SIMJUM RISJUM	JUMSIX	TETJUM	ROLJUM
(N + $7F)	1	129	129	5	3	132	65
	1	66	64	4	5	70	38
	1	36	64	7	4	43	22
	1	20	16	10	5	28	22
(5N + 1)	64	187	205	187	119	148	161
	16	156	64	183	100	157	143
	4	143	63	158	90	163	144
	1	146	16	145	96	166	147
(9N + 1)	32	137	199	163	105	138	135
	4	118	64	161	86	136	124
	1	137	64	150	79	118	140
	1	141	16	162	93	142	125
(13N + 1)	64	175	198	179	127	158	162
	16	160	64	182	108	158	146
	4	162	63	158	70	159	150
	1	160	16	171	98	171	147
(17N + 1)	16	107	193	151	89	124	121
	1	130	64	140	90	122	121
	1	123	64	132	84	131	134
	1	133	16	140	95	161	130

```
A5 C1    SIMRND LDA RND (load seed)
ØA              ASL A   (X 2)
ØA              ASL A   (X 4)
38              SEC     (to add 1)
65 C1           ADC RND (X 5 + 1)
85 C1           STA RND (next seed)

C9 CØ    COMJUM CMP #$CØ (bit 7 = bit 6?)
1Ø Ø2           BPL NEXT (yes, retain #)
49 FF           EOR #$FF (no, complement)
           NEXT

1Ø Ø2    SIMJUM BPL NEXT (bit 7 = Ø, retain #)
49 7F           EOR #$7F (complement all but 7)
           NEXT

4A       RISJUM LSR A (bit Ø to carry, zero b7)
9Ø Ø2           BCC NEXT (bit Ø = Ø, retain #)
49 FF           EOR #$FF (bit Ø = 1, complement)
           NEXT

1Ø Ø2    SIMJUM BPL RISJUM (for 2 jumblings)
49 7E           EOR #$7E

24 C1    JUMSIX BIT RND (bit 6 to V flag)
5Ø Ø2           BVC NEXT (if = Ø, retain #)
49 BF           EOR #$BF (complement all but 6)
           NEXT

C9 CØ    TETJUM CMP #$CØ (quad jumbling)
1Ø Ø2           BPL OMIT
49 FE           EOR #$FE
49 Ø1     OMIT  EOR #$Ø1
1Ø Ø2           BPL NEXT
49 21           EOR #$21
           NEXT

C9 CØ    ROLJUM CMP #$CØ  (quad jumbling)
1Ø Ø5           BPL ROLOP (half have set carry)
49 FF           EOR #$FF (sets N for half)
1Ø Ø1           BPL ROLOP (N and carry clear)
38              SEC     (N and carry both set)
2A       ROLOP  ROL A  (all  numbers new)
           NEXT

E6 C2    INCRND INC COUNT
DØ Ø2           BNE SIMRND
E6 C1           INC RND (next sequence of 256)
         SIMRND

E6 C2    DUBRND INC COUNT
DØ Ø4           BNE SIMRND
E6 C1           INC RND
C6 C3           DEC ADDEND
```

Continued on pg. 19