Complex Pseudorandom Sequences from Interlaced Simple Generators

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

My first effort at generation of a sequence of pseudo-random numbers (*DDJ* # 32) was primarily aimed at minimizing the *time* required to generate successive 8-bit binary numbers. The basic logic outputs a non-repeating sequence (NRS for short) of all 256 possible numbers. This "core" sequence can be transformed into a limited number of variants by simple mathematical operations. Such variations can be *concatenated* to yield a longer NRS. However, this logic entails a great sacrifice of the *quality* of the numerical sequence, that is easily recognizable as non-random and is not difficult to decipher.

Interest in random-number generation has been increasing in the microcomputing community. Bungay and Martin (*KILOBAUD* # 28, p. 46) designed a *hardware* generator controlled by a "noisy" clock, that is said to yield a "truly" random sequence of 8-bit numbers. Timing was not given, but was implied to be slow although fast enough for use with a BASIC interpreter. The unpredictability of the next number in the sequence, however, disqualifies such generators for some important applications, e.g. "hashing" algorithms or cryptography.

All this has tempted me to create more complex generator logic, so that the output will be a closer approximation to "true" randomness. Since the instruction sets of 8-bit microprocessors are inefficient for "number-crunching", complexity is achieved by *interlacing* the output of n independent simple generators. The concept is probably not novel, although its value for enhancing the quality of a pseudorandom sequence is not common knowledge (I did not find it in the elaborate discussion of random numbers in Knuth's *Seminumerical Algorithms*, perhaps because it's too simple.) An interlacing algorithm is easy to implement in microprocessor code, and makes possible a planned *synthesis* of very diverse numerical outputs with extremely long NRS.

Simulation of "True" Randomness

Although I have no expertise in probability theory, a few principles are demonstrable at an elementary level:

1. The occurrence of any sequence of 256 all-different numbers is wildly improbable, even though the number of possible sequences of this type is 256 factorial, for which Stirling's approximation gives the astronomical number 2.43×10^{525}. This is dwarfed by the total number of all possible sequences, 256^{256} or 3.23×10^{616}. The probability of an all-different sequence is an infinitesimal $7.5 \times^{-92}$, and no self-respecting simulation should ever generate such a sequence! Other *kinds* of sequences have immensely higher probability, that can be calculated from formulas for permutations and combinations. There are so many types of sequences that it is impractical to do such calculations for any significant fraction of them. For the type consisting of 192 different numbers, of which 64 are duplicated, there exist 1.03×10^{572} possible sequences, and the probability of occurrence (immeasurably higher than that of the all-different sequence!) is still a ridiculously low 4.2×10^{-45}. The maximum probability (still abysmally low) for this type is near 187 different numbers, with 1.6×10^{574} possible sequences. This analysis, inadequate though it is, suggests that generator design should aim at a zero-order heterogeneity (defined as H_0 in my previous *DDJ* article) near 187.
2. The probability that any given 8-bit number will *not* be the next number in a random sequence is 255/256, roughly 0.99609, and that it will not occur in the next n numbers is $(99609)^n$. In a one-page sequence, the probability that a given number will not occur is 0.3672; therefore the probability that it *will* occur is only 0.6328 – very far from certainty! The probability of its occurrence rises to 0.8652 for 2 pages (512 consecutive numbers), 0.9505 for 3 pages,

and 0.9818 for 4 pages. I am unable to imagine program logic that would even approximate these specifications, even if it were inordinately complex. One can guess that realistic logic should try *not* to include every possible 8-bit number in any 3-page (or even longer) output sequence. This accords with an H_0 value of 187, that would exclude 69 of the possible numbers from any one sequence page.

3. One could extend the above line of reasoning to the 16-bit level, viewing 2 successive outputs as one number. The probability of occurrence of any one of the 65536 possible numbers is of course very low, and the calculations would be tedious. One can guess that only a fraction of the possible numbers will occur even if the NRS exceeds any realistic limits. On the other hand, frequence recurrence of the same number is improbable. The proper strategy may be to *try* to generate as many different numbers as one can, and generate as long an NRS as one is likely ever to require.

An Experimental Generator Algorithm

Logical complexity is a perilous sea. More program and working memory, worse timing, diminishing returns, poor predictability, formidable testing problems, and the many other evils that complexity so generously bestows!

My program MIXSIM (cf. listings, 6502-coded) presents one way of interlacing the output of *n* simple generators (each one yielding a different sequence of the 256 different 8-bit binary numbers) into a synthesized sequence with an NRS of $n(256)^n$, or more than 50 million if $n = 3$. The interlacing logic is not very complex. Location MEMEX "remembers" the value of the X index register that selected the generator used in the previous subroutine call. This is reloaded into X and decremented to shift control to the next lower generator in the sequence. If MEMEX was zero, however, the X register is RESET. In the example X becomes 02, and generator G_2 always outputs its "next number", resetting MEMEX to 02 just before exit. The *next* call will use X = 01. It first loads the previous G_2 output into the accumulator, and if this was *not* zero branches to the normal generator logic to output the "next number" of the G_1 sequence (resetting MEMEX to 01). If the previous G_2 was zero, normal G_1 generator logic will be *bypassed* and the output will be created by a "pseudo-generator". The LDA RND+2,X instruction is an "antibug" that guards against a *double-bypass* of normal generator logic. It loads the content of the *next higher* location to test whether it is also zero. When G_1 is being tested, it is trying to load the "previous output" of the non-existent G_3, that will in reality be a non-zero constant used by the logic of G_0. The BEQ is therefore not executed, and a "pseudo" number is fabricated by loading the previous G_1 output and forced-branching to DEJUM, that complements the 7 low-order bits. Since the G_1 "seed" is unchanged, the effect is to *offset* (or desynchronize) the G_2 and G_1 generators. The frequency of normal G_1 output is 255/256 that of the always-normal G_2 output. Therefore G_2 and G_1 will not *both* re-initialize until G_2 has output 256^2 numbers, and G_1 has output 255 X 256 normal and 256 pseudo numbers.

The logic also desynchronizes the G_0 generator, in a more complex way. The guard instruction ensures that all pseudo G_1 outputs will be followed by a normal G_0. Since 1/256 of the normal G_1 outputs will be 00, 255 pseudo G_0 outputs will occur for every 256^2 G_2 outputs, together with (255 X 256) + 1 normal G_0 outputs. Simultaneous re-initialization of all 3 generators therefore will not occur until 256^3 G_2 numbers have been output. The other generators will have output as many numbers, but 256^2 G_1 and 255 X 256 G_0 outputs will have been "pseudo".

The Logic of the Simple Generators

Although MIXSIM uses an X-indexed version of my earlier SIMRND generator, because I feel it has a near-optimal balance of speed, simplicity, and output heterogeneity, other types of generators might interlace as well. One requirement of interlacing is that each of the generators yield a distinctly different sequence of the 256 different numbers; otherwise the same number will show recurrence at regular intervals. The variant sequences are produced in MIXSIM by appending my earlier ADDRND module. The base location for the 3 addends is RND+3. Since all the variants have low heterogeneity in the low-order bits, this is improved by appending the "jumbling" module SIMJUM, with the added advantage that its DEJUM instruction can be used by the pseudo-generator logic.

Many distinct versions of MIXSIM are possible. The G_0 addend location (RND+3) can be any one of 255 numbers, since 00 is excluded to ensure unconditional operation of the G_1 logic. The G_1 addend can be any one of 255 numbers, since 00 is allowed, and the G_2 addend can be any one of 254 numbers. Since every permutation of addends yields a distinct long sequence, more than 16.5 million versions are possible. Perhaps some of these will yield a "better-looking" long sequence than others. I instinctively feel that addends with quite different bit-patterns are preferable, but instinct is far from an infallible guide in the jungles of complexity!

The Possibility of Greater Complexity

As one seeks to improve sequence quality by more complex logic, the number of *possible* ways becomes even larger and the difficulty of foreseeing flaws becomes greater. I see *no* way of creating a sequence that will satisfy the specifications of "true" randomness that I lightly touched on earlier in this note. What *may* be fabricable is a sequence that would be extremely difficult to decipher. MIXSIM is not. Its G_2 output appears regularly as the first of every sequence-of-three, and repeats after every 3 X 256 outputs. One fairly simple way of correcting this would be to add (following the RESET instruction) my earlier INCRND module, to increment the G_2 seed location at the start of each full G_2 sequence. This will increase the NRS of G_2 to 256^2 and (if there are no pitfalls!) the long-sequence NRS to billions of numbers. Checking this out would be tedious, even if the INCRND logic is not applied to any of the other generators, to avoid augmenting the number of initialization states beyond 256^4.

A less interesting approach would be to modify MIXSIM to use 4 (or more) different generators. This would retain the transparency of the G_2 output, but (if problem-free) greatly lengthen the NRS. One could of course go to *higher* levels of interlacing, with additional logic that would interlace the output of generators of the MIXSIM type. I remain skeptical that the resulting output could withstand even moderate analysis without revealing its pseudorandomness, even if the generating logic becomes undecipherable.

Some Rough Tests of the MIXSIM Output Sequence

If the initial state of MEMEX is 02, with seeds (G_0, G_1, G_2) of 00, 01, 02 and addends of $59, $A6, 00, and 10 successive sequences of 256 numbers are run, the zero-order heterogeneity (H_0) ranges from 176 to 184, with a mean of 180.7 and standard deviation of 3.13. Numbers that occur only once (in a set of 256) range from 106 to 120, mean 114.8 and s.d. 5.96. Numbers that are duplicated range from 52 to 60, mean 56.5, s.d. 3.10. Numbers that are triplicated range from 8 to 11, mean 9.4, s.d. 1.07. Even an analysis as simple as this is sufficient to prove pseudorandomness (or so it seems to me!)

A different analysis involved running MIXSIM continuously, counting the number of each of the 256 possible 8-bit outputs until one of them was output 256 times. With the same initial states used in the previous test, this test routine terminated after outputting the number $4C 256 times. All other numbers occurred from 190 to 194 times. The egregious surplus of a unique number among the over 49000 numbers in the long sequence is glaringly non-random. This odd result led me to change the seeds to 11, 22, 33 (with the same addends). This run yielded 256 $C8 values, with all other numbers occurring from 213 to 217 times. The "preferred number" is therefore determined by the values selected for the seeds. Changing the addends to 57, A6, 00 had no notable effect, $C8 still being preferred. Since the fundamental logic cannot cause such a preference in the entire NRS, I presume that the "preferred" number *changes* in various short sequences, but have not bothered to test this thoroughly. I did run MIXSIM in an endless loop for about 25 minutes, using the same initial conditions as in the first paragraph of this section, to get it about half-way through its NRS. When halted, MEMEX was 02 and the seeds had become 06, E3, 8D. Running the second test routine from that point yielded $10 as the "surplus" number, with the others occurring from 198 to 201 times. The inherent monotony of the output was revealed by one run of the first test routine, giving an H_0 of 183 with 119 once-occurring numbers, 55 duplicates, and 9 triplicates. So even an amateur cryptanalyst would know a lot about MIXSIM even from a sequence as short as 512 numbers! It's not that this *type* of sequence is so improbable. There are 4.8×10^{579} possible sequences of 118 singles, 57 duplicates, and 8 triplicates This type is therefore 10000 times more probable than the duplicates-only sequences I dealt with at the beginning of this note. What is incredible is that even 2 of them should occur in a truly random numerical sequence, let alone that *all* sequences should be of the same type! The simplicity of the underlying logic is transparent. A clever analyst (and any large organization is likely to have at least one brilliant analyst) would have no trouble deciphering it. Mere *length* of the NRS represents no genuine challenge. As with the simple generators I discussed in *DDJ* #32, the problem is the creation of *heterogeneity*.

The interlacing logic module is too simple to be undecipherable, but that was not its goal. It is fast, adding only about 16 cycles to the mean timing per output (that totals about 45 logic-only cycles). The cost in memory locations is greater: 38 program-logic and 7 working bytes. The output does represent a closer approximation to true randomness, since it consists of nearly 200,000 successive one-page sequences that are all different and of a type that is 10^{54} times more probable than the 256-different-number sequences of its underlying simple generators.

The next step may be much more costly and yield diminishing returns. It ought to be based on a much more detailed analysis of sequence probability, perhaps beyond the 256-byte level, so one will know what kinds of sequences need to be synthesized. Merely avoiding monotony of single-page sequences is not enough. The question is: what kind of variety is most probable?

Some Thoughts About the Role of *DDJ*

Few journals would dream of publishing this kind of disquisition on tiny algorithms. Computer scientists will deem it "trivial", a favorite pejorative computerese word. Mathematicians would (quite rightly) deprecate the lack of generality and abstract thinking. On the other hand, many amateurs might find neither the purpose nor the presentation crystal-clear. *DDJ* is for in-between types, like me. Feedback in response to some earlier articles has revealed a fascinating variety of human personalities, of the inventive type, most of them, not surprisingly, interested in useful code, often with a practical goal and a concern for optimal timing, re-enforcing my belief that it really matters, and that small modules have their own uses. This restores morale that sometimes erodes when I compare my *haikus* with the Homeric software epics I see everywhere!

A few correspondents are more interested in concepts than code, and some even empathize with the traces of idealism that at times surface in whatever I write. I suspect that latent idealism is at the heart of *DDJ* and sustains its open-to-all, unstructured and unhierarchic outlook, that contrasts so vividly with the *Weltanschauung* of the commercial or subsidized technical journals. These are run by some elite person or group, whose expertise enables them to detect and exclude obvious errors. That's a plus, but the expert human mind also tends to reject the unfamiliar and the unconventional. Domination by experts or an establishment tends to crush novel thought patterns (that will indeed usually be imperfect at birth). This is not yet a problem for the editors of *DDJ* (though they have lots of *other* problems!), who don't seem to know what heresy is. Still, the second law of thermodynamics is inexorably at work. Fires cool, and (if, as I hope, *DDJ* survives) one cannot but wonder into what pattern *DDJ* will crystallize.

Continued on pg. 41

set = 0 will increment the memory text line number displayed on the top character line of the screen. The memory text line formerly at the top of the screen will appear as the bottom line on the screen. This occurs regardless of the display mode (20 or 24 line screen).

It is important to note that in the 20 line mode, the highest four memory text lines (20 through 23) do not participate in the scrolling, although they are accessible by the CPU. These lines are used by the "subtext" command, which is described below.

Scrolling down is accomplished by scrolling up N-1 times, where N is the number of character lines displayed on the screen. Since the scrolling hardware is read internally at the leading edge of the vertical display signal VDISP, scrolling should be held off until immediately after this event. VDISP is made available to the CPU on bit 1 of the status port. The CPU should store the last value of this bit and test the new value to detect a change from 0 to 1.

The CPU may reset the scrolling hardware by outputting to the screen port with bit 5 set = 0. This will override any data on bit 4 and will cause the top line of the display to be memory text line 0.

Crawling

The VDM-2 includes the hardware necessary for the movement of the display vertically in increments of one scan. When synchronized with the scrolling, this feature allows for smooth text movement up or down the screen as if the screen were a window gliding over an unbroken column of text.

In such crawling it is ideal if the top displayed line is gradually eclipsed by the top of the screen while the new bottom line emerges gradually at the same time. Since in the 24 line mode of the VDM-2 the partially obscured top and bottom lines are in fact the same text line in memory, this ideal condition is not realized. Instead, a one-line "preblank" curtain descends over the top line at such strategic times.

Crawling is controlled by bits 0 through 3 of the screen port. Bit 0 sets the direction (0 = up, 1 = down), and bits 1 and 2 select one of four crawl rates. Three of these are generated in the hardware as submultiples of the 60 Hz vertical rate. One (both bits = 1) provides for an externally generated crawl clock, which may be, for example, an oscillator whose frequency is controlled by a joystick. Bit 3 is the "go" bit which initiates a crawl sequence when set = 0 during a screen port output.

Upon issuance of a "crawl up" sequence, the display will wait until the onset of VDISP and will then appear to "jump down" one character line. The old top line will still appear at the top of the display, but it will be displaced down by one character line. The display will gradually crawl back up to its original state, advancing one scan for each crawl clock. When the starting point is reached, bit 0 of the status port (CRAWL DONE) will set = 1 (it sets = 0 immediately upon issuance of a crawl sequence command).

It should be obvious that if, immediately following the issuance of a "crawl up" command and before the onset of VDISP, the new bottom line is written into the old top line and the display is scrolled up one line, the result will be the disappearance of the top line and a slow scroll up, with the old top line appearing at the bottom of the screen with its new contents.

When a "crawl down'. sequence is initiated, the subsequent onset of VDISP will cause the display to move down one scan, with the bottom scan of the bottom line obscured. This downward movement will continue, one scan for each crawl clock, until the bottom line has been completely obscured. The CRAWL DONE bit will now return t-o = 1. At this point the new top line should be written into the old bottom line and the screen should be scrolled down one line. The newly written line will not become visible until after a new "crawl down" sequence is initiated.

If the newly-written top line is that last of a sequence and it is desired that the crawling stop, issue the crawl down command and follow it immediately with an output to the screen port having the direction bit = 0 (up) and the GO bit = 1. The new line will appear at the top of the screen and on the next crawl clock will move up one scan. CRAWL DONE will then set = 1.

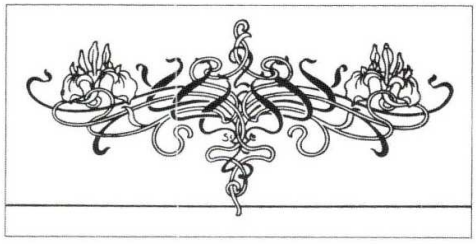
When the external crawl clock is selected it is useful to have an externally provided "go request" and "direction request" set of signals so that text movement may be controlled without use of the standard input channel such as the keyboard. Bits 4 and 5 of the status port are connected to the external device connector and are read by the VDM-2 when the status port is read. These bits are advisory to the CPU and have no function in the VDM-2 hardware. They will show a default "1" value when no external device is connected.

Subtext

In the section on scrolling it was mentioned that four memory text lines remain unseen when the VDM-2 is in the 20 line mode. These text lines may still be accessed by the CPU, and constitute a "subtext" area for information of secondary utility. Upon command from the CPU these lines can be made to appear at the bottom of the screen obscuring one or more of the text lines normally displayed.

Bit 5 of the control register (SUBTEXT 0) will cause the display of one or more of these lines when set = 1. The number of subtext lines to be displayed is determined by the binary value of bits 6 and 7 of the control register (SUBTEXT 1 and 2). Taking bit 7 as the most significant bit, values 0, 1, 2, and 3 represent 1, 2, 3, and 4 displayed subtext lines. These lines always end at the bottom of the screen and the topmost line is always memory text line 20. The subtext lines do not respond to crawl or scroll commands.

Subtext lines are intended for uses such as prompting, system status information display, menus, "jack-in-the-box" quick comebacks, etc. Use of the subtext should be avoided in the 24 line mode, since the subtext lines will be duplicates of lines visible elsewhere on the screen.



Interrupts

Interrupts have been left unimplemented in the current prototype S-100 version of the VDM-2. Since interrupts are a system consideration and since the VDM-2 is currently without a system, many configurations are possible.

As a minimum, the CRAWL DONE bit could cause an interrupt by its onset when interrupts are enabled. This would allow smooth scrolling with a minimum of CPU attention.

Bit 7 of the screen port has been tentatvely designated as INTERRUPT ENABLE. Its exact mode of functioning awaits the final design of the device.

```
BIT SUMMARY -- OUTPUT PORTS

Control port

 bit no.   Name          Function

   0       DISABLE     - Prevents memory decoder from responding
                         when set = 1
   1       FONT        - Causes font memory (writeable) to re-
                         spond to CPU memory references when
                         set = 1
   2       PAGE        - Causes high order four bits of screen
                         memory character word to respond to
                         CPU memory references when set = 1
   3       BLINK       - Determines whether cursor or video blinks
                         on characters having page 1 bit 2 set = 1.
                         0 causes video blink, 1 causes cursor blink.
   4       24          - Determines number of character lines per
                         screen. 0 causes 20 lines of 12 scan each.
                         1 causes 24 line display of 10 scans each.
   5       SUBTEXT 0   - Controls display of subtext lines at
                         bottom of screen. 1 = subtext displayed.
   6       SUBTEXT 1   - Least significant bit of pair
   7       SUBTEXT 2   - Most significant bit of pair
                         Bit pair value determines number of
                         subtext lines to be displayed. Number of
                         lines is one plus binary value of bit pair.

Screen port

 bit no.   Name          Function

   0       DIRECTION   - Sets direction of crawl. 0 = up.

   1       Speed 0
   2       SPEED 1     - As a binary pair (bit 1 least significant)
                         sets one of four crawl speeds:
                            value     speed
                              0       60 scans/ sec.
                              1       30 scans/sec.
                              2       15 scans/ sec.
                              3       externally clocked
   3       GO          - Initiates crawl operation when set
                         = 0
   4       SCROLL      - Increments the number of the memory text
                         line displayed at the top of the screen.
                         0 = increment
   5       RESET       - When set = 0 causes memory text line 0
                         to be displayed at top of screen.
                         Overrides effect of bit 4.
   7       INTERRUPT
             ENABLE    - reserved for use in interrupt version


BIT SUMMARY -- INPUT PORT

 Status port

bit no.     name          function

0       CRAWL DONE      Sets = 0 when crawl is in progress.

1       VDISP           Sets = 1 during vertical display time
                        (240 scans)

2       CRAWL CLOCK     Sets = 1 during crawl clock signal.
                        Crawl advances one scan at onset of
                        VDISP following leading edge of crawl
                        clock.

3       unused

4       EXT. CRAWL RQST   Sets = 0 when external crawl-control
                          hardware is requesting text movement

5       EXT CRAWL DIRECTION   sets = 0 when external crawl
                              control hardware requests crawl
                              down.

        (note- bits 4,5 are set = 1 when no external device
               is connected.)
```

Continued from pg. 35

Copyrights

As usual, I assert the free-diffusion-clause copyright defined in *DDJ* #32 and earlier articles. I recommend this to algorithmaniacs of my ilk, since it encourages users to let you know of unexpected applications *they* have found for your creations.

Hal Gordon is 60 years old, has a Ph.D (Biology, Harvard, 1947), and has been at UC – Berkeley since 1947. He is a low-level programmer in more ways than one, having learned how to use a CPU directly on an early KIM-1 and only recently upgraded to a SYM-1. His (still unrealized) dream is to equip these boards with sensors and effectors so that they can control real-time biological experiments. He sympathizes with the anti-AI views of Joseph Weizenbaum on the limits of human-language-oriented programs, believing that it's more instructive to interact with books and the right kind of human mind.

```
     (listing of subroutine MIXSIM, without addresses since
       it is fully relocatable)

A6 DF   MIXSIM LDX MEMEX    (load prev. G-index into X)
CA             DEX          (decrement X for next G)
30 0D          BMI RESET    (if = $FF, go reset X to 02)
B5 E1          LDA RND+1,X  (load G+1 seed into Acc.)
D0 0B          BNE SIMRND   (if ≠ 0, run normal G logic)
B5 E2          LDA RND+2,X  (load G+2 seed into Acc.)
F0 07          BEQ SIMRND   (if = 0, run normal G)
B5 E0          LDA RND,X    (load G seed into Acc.)
18             CLC          (clear carry for branch)
90 10          BCC DEJUM    (output pseudo-G number)
A2 02    RESET LDX #$02     (reset X to 02)

B5 E0   SIMRND LDA RND,X    (load G seed into Acc.)
0A             ASL A        (X 2)
0A             ASL A        (X 4)
38             SEC          (set carry to add 1)
75 E0          ADC RND,X    (X 5, + 1)
95 E0          STA RND,X    (store new seed)

18             CLC          (clear carry for addition)
75 E3          ADC RND+3,X  (add addend for G variant)

10 02          BPL STOREX   (bypass DEJUM if bit 7 = 0)
49 7F    DEJUM EOR #$7F     (complement lowest 7 bits)

86 DF   STOREX STX MEMEX    (store G-index value)
60             RTS          (exit subroutine)

    Zero-page locations used (all must be initialized,
      cf. text for restrictions):

DF   MEMEX  (must be 02 or 01 or 00)

E0   RND    (seed of G0)
E1   RND+1  (seed of G1)
E2   RND+2  (seed of G2)

E3   RND+3  (addend of G0, must not be = 00)
E4   RND+4  (addend of G1, ≠ RND+3 and ≠ RND+5)
E5   RND+5  (addend of G2, ≠ RND+3 and ≠ RND+4)
```