Marsaglia Revisited: Rapid Generation of Fitted Random Numbers

Ferren MacIntyre
Graduate School of Oceanography
University of Rhode Island
Narragansett RI 02881 USA

Abstract

Source code is given for a fast hex-integer version of Marsaglia's well-known method for converting uniformly distributed pseudorandom numbers into any desired distribution.

A useful tool for developing programs which process complicated numerical data is a test data set having properties similar to the real data. One way to obtain such a test set is by generating pseudorandom (henceforth, simply "random") numbers with the same distribution Y as the data. But a typical random-number generator (RNG) produces a flat distribution U of numbers u_i in which all histogram bins are equally filled, within statistical fluctuations. Y can be easily found from U only if it has an inverse, so that one can solve $y_i = Y^{-1}(u_i)$, and even this may involve slow operations such as floating point or scaling. A Gaussian distribution can be obtained from uniform pairs at the cost of a log, square-root, sine, and cosine calculation for each pair **[BOX58]**. Anything more complex requires special techniques.

Let p_i be the point probabilities of a discrete random variable Y. **[VONN51]** suggested the obvious method of generating Y by testing each number u_i and incrementing Y's i-th bin y_i if

$$p_1 + p_2 + \ldots + p_i - 1 < u_i < p_1 + p_2 + \ldots + p_i \qquad (1)$$

but this is slow because it requires order of $n/2$ comparisons for each number, where n is the number of bins.

Marsaglia's approach **[MARS62]** requires a single comparison most of the time, and never more than $k - 1$ comparisons, where k is the number of significant digits needed to describe the distribution. According to its most accessible source (**[ABRA65]**, p. 951), Marsaglia is "the best all-around method for generating random deviates from a discrete distribution," but they give only a floating-point decimal version of the algorithm. Here we present a hexadecimal integer version, better adapted for rapid use on microcomputers.

We follow the notation, argument, and binomial-distribution example, of **[ABRA65]**. For an n-member distribution, let the probability of the i-th bin p_i be expressed by the k hexadecimal digits d as

$$p_i = d_{1i}\, d_{2i}\, d_{3i}\, d_{4i}\ \ldots\ d_{ki},\ i = 1, 2, 3, \ldots, n$$

where k will ordinarily be 4 in a 16-bit environment and 8 in a 32-bit environment. The representation of p_i may be truncated by setting any number of the d_{ki} s to 0 from the right, but it is a rare distribution which needs more than 4 significant figures.

Define

$$P_r = \sum_{j=1}^{n} d_{rj} \text{ for } r = 1, 2, \ldots, k$$

$$\prod s = \sum_{r=1}^{s} 10^{r-k} P_r \text{ for } s = 1, 2, \ldots, k$$

Since these are obscure abstractions, we illustrate with the binomial example used in **[ABRA65]**, first in decimal integer so that the numbers, except for the decimal point, are identical to those in **[ABRA65]**, and then in hexadecimal integer. Suppose the normalized binomial distribution of Table 1 is required:

x_j	p_j	
	dec	hex
0	3277	54A9
1	4096	8204
2	2048	2503
3	0512	0426
4	0064	0021
5	0003	0008
Sum	10000	0FFFF

Table 1. Binomial distribution in decimal and hexadecimal.

To illustrate the meaning of the parameters defined above, we turn this format on its side in Table 2:

	Value of random variable						P_r	$\sum P_r$	$10^{r-k}P_r$	$\prod r$	$10^{r-k}\sum P_r$	$10^{r-k}\sum P_r - \prod r$
r	0	1	2	3	4	5						
1	3	4	2	0	0	0	9000	9000	9	9	9	0
2	2	0	0	5	0	0	0700	9700	7	16	97	81
3	7	9	4	1	6	0	0270	9970	27	43	997	954
4	7	6	8	2	4	3	0030	10000	30	73	1000	9920
1	5	8	2	0	0	0	F000	**F000**	F	**F**	**F**	**0**
2	4	2	5	4	0	0	0F00	**FF00**	F	**1E**	**FF**	**E1**
3	A	0	0	2	2	0	00E0	**FFE0**	E	**2C**	**FFE**	**FD2**
4	9	4	3	6	1	8	001F	**FFFF**	1F	**4B**	**FFFF**	**FFB4**
Computed by:								**4pis**		**4pis**	**4sigs**	**4sigs**
Stored as:								**m.comp**				**m.off**

Table 2. Random integer distributions in decimal and hexadecimal format, and their associated parameters.

P_r is the sum of the row digits, in unaltered magnitude. $\sum P_r$ simply sums P_r row by row. $10^{r-k}P_r$ shifts these digits to the right, and $\prod r$ sums them by rows. $10^{r-k}\sum P_r$ is $\sum P_r$ similarly shifted right, and $10^{r-k}\sum P_r - \prod r$ is the difference of these last two, which when subtracted from the processed random number, reduces it to an index into the table described below. The bold entries are calculated and used by the program shown in Listing 1.

Next, create an array a{m} of at least **Size** = $\prod k - 1$ locations, which we will call a "marsaglian table." Fill it from the outlined hex portion of Table 1: five 0s, eight 1s, two 2s, four 0s, two 1s, five 2s, four 3s, etc. The desired distribution will be drawn from this table. Memory dumps of the decimal and hexadecimal versions of a{m} look like:

```
0 0 0 1 1 1 1 2    2 0 0 3 3 3 3 3
0 0 0 0 0 0 0 1    1 1 1 1 1 1 1 1
2 2 2 2 3 4 4 4    4 4 4 0 0 0 0 0
0 0 1 1 1 1 1 1    2 2 2 2 2 2 2 2
3 3 4 4 4 4 5 5    5

0 0 0 0 0 1 1 1    1 1 1 1 1 2 2 0
0 0 0 1 1 2 2 2    2 2 3 3 3 3 0 0
0 0 0 0 0 0 0 0    3 3 4 4 0 0 0 0
0 0 0 0 0 1 1 1    1 2 2 2 3 3 3 3
3 3 4 5 5 5 5 5    5 5 5
```

Now generate uniform random numbers u_i with digits

$$u_i = d_{1i}\, d_{2i}\, d_{3i}\, d_{4i},$$

and test each number u_i against P_i. Then – the crux of the operation – if

$$\sum_{i=1}^{s-1} P_i \le u_i < \sum_{i=1}^{s} P_i$$

chose the value in memory location

$$\{d_1 d_2 \ldots d_s + \textstyle\prod_s - 10^{r-k} \sum P_r\}.$$

In terms of our example, this last decision appears in decimal as:

If $0 \le u < 9000$	put	$y = ad_1$
$9000 \le u < 9700$		$y = ad_1d_2 - 81$
$9700 \le u < 9970$		$y = ad_1d_2d_3 - 954$
$9970 \le u < 10000$		$y = ad_1d_2d_3d_4 - 9927$

and in hex as:

If $0 \le u <$ F000	put	$y = ad_1$
F000 $\le u <$ FE00		$y = ad_1d_2 -$ E1
FE00 $\le u <$ FFE0		$y = ad_1d_2d_3 -$ FD2
FFE0 $\le u <$ FFFF		$y = ad_1d_2d_3d_4 -$ FFB4

Thus 15 times out of 16, only the first comparison and one bit-shift are required.

As a cautionary note, marsaglian extraction makes extreme demands of the uniformity of the underlying RNG. The reason can be seen by noting the difference between the frequency of a number in the distribution itself, and in the marsaglian table from which will be drawn. Thus, 4 appears oftener than 5 in the binomial distribution (64:3), yet more rarely in the table (3:8), a 20-fold difference in ratios. Thus any flaw in the RNG which biases the manner in which it selects from the table may be greatly amplified. *Caveat utor!*

As an example of a common distribution which has no inverse, imagine a need to test (or perplex) a cryptographic-analysis program, where the starting point is the letter-frequency distribution of English text. The approximate distribution is given in `letters` in Block 6 of the source code below. The space is included as an alphabetic character (in the sequence : space z y x ... c b a count) to approximate the average word length of English (4.5 letters), but it may be removed if the count at the end of `letters` is also reduced to 1A. Random text drawn from this distribution has the single-letter frequency count of English, as seen in Fig. 1, and differentiating it from a transposition cipher might be difficult. If encrypted by simple substitution, it could be distinguished from a meaningful text only by digraph (letter-pair) and higher-order correlations.

The point is not that we have made an inadequate attempt to produce literature with a monkey at a marsaglian typewriter. (The most coherent thing this version produces are occasional sequences like "seamd base," "taste near," "digtle sails," and "sleet and." If you are after "words" that sound more like English, one approach that yields a greater harvest with less effort

```
.table
aeot   abbbccccddddddddeeeeeeeeeffffgggggghhhhhhhhhhhhhhiiiiiiii
iiiiiklllllllmmmmmnnnnnnnnnnnnnnpprrrrrrrrrrrrssssssssssttttuuuu
uvwwwwwyyy aabbbbbbbbccccccddddddddddeeeeffffffghhhiijjkkkkkkkkk
kllllllllmnnnnnnnnnnnnooooooooooooooppppppppppppppqqrrrrsssssstttt
tttuuuuuuuuuuuuuuuvvvvvvvvvvvvvvvwwwwwxxyyyyyyyyyyyyyyyzzzz    a
aaaaaaaabccccccccccddddeeeeeeffffffffffffgggggggiiiiiiiiikkkkk
kkkkkllllllllllmmmmmmmmmmmmmmmmnnnnnnnnnooooooooooqqqqqqqqqqqqqrrr
rrrrrrrssssstttttttttttttvvvvvvwwwxxxxxxxyzzzzzzzz
flat.text
emxogbzbyjhcjjuybl obl aysqovrrlzdvdjbevyuplmkifepkoobqlppvhlbf
usugrnafadyepebllfwjepicymzxyfu oligk kgxosefanthbfhuxyj lpuzhnc
efbjcs vsb omwg yaasekrfeyrkfhgxkatlnca rbpcdznoqfewcyvmhluppyz
itjmbzbvjyfeevhwrmqoyqhwf bfmqs vnakjbbikyvmnsxhbfm ytfvfmzxggox
ygrrevckowocdoufezqjcbtnhaxxdcdcs itusvtnwzwvlkumgudenttxmusymcu
ouzepkdymcinnhqqenasixfxj hmvdofilqb kvwnctvvwdbxexhktqjznkprurr
xhgzprdikxvxvaylqyfwnrcrl mdlpyicyyawuuljdklmch ihjlpzj anpzrzfo
hvobsctgqvauvunkyol ipxtnygvcqmlnkfdi tarvqbdjgwgdhlgepjbnuq ggl
shape.text
o nrtnedao asnseeoataohci e  eu o dhat g tuaebmheid aat emnlpec
acamdsaso th  a hta ha st  ht p toetdtprnleea ckaw t dhho  oseee
nb seaelr heot tcwvhift ddtu nautthia aor  agc et oseor oe o drm
hyed b roe ia sht t hcb vaan dakmdh e oobndncho acomdkete fkod
h o latka dshel a ao shnhstccedn eawcahm a  n oep tnetdnleh co
  rhfn  codntp lt tm  aah e y ait  szona ooaawsfhw cooee itg  rh
 t alaedg ehot mttt tteatihheht dcowaich  vgdoethooesd faoawpor
 oat nihise a  mdbas  eetet l   m ataa sn kteebo hthhno ca eou r
```

Fig. 1 Random distribution of letters. *Top:* the array from which marsaglian extraction is made. *Center:* Random text with a uniform distribution of letters. *Bottom:* Random text with the normal English letter distribution.

than a complete 27×27 digraph frequency table is to include the phonetic digraphs – ng, th, wh, sh, etc. – as alphabetic characters.)

The point is that any chosen distribution can be used by writing a single word like **letters**. I have been known to draw samples for homework assignments from random distributions: the answers are all supposed to come out the same (e.g., 42), but the questions are sufficiently different to keep students honest.

Source Code

The code, for good but irrelevant reasons, is written in National Instrument's IBCL, a dialect of Jim Callahan's HS/Forth™ V2.1, once compatible with Forth-79, but considerably reworked and cast in silicon. It differentiates typographic case, but all of its own vocabulary is in lower case. It lacks **quan**s or other multiple-cfa structures, so I use constants with **'** instead of littering the code with **@** from variables (storing being rarer than using). Constants, like German nouns, begin with a capital letter.

Its **'** is the old state-smart tick which always does the same thing (from the user's point of view), does not require thinking about, is easy to test, and takes up less room on the line than its idiot child in Forth 83.

Non-standard words in the source code include the defining words **array** and **carray** (no apology: *I* didn't leave them out of the Standard!), which want a size on the stack at compilation

and an index at execution. **dump** (address count →) dumps count bytes of memory from address. It may help in reading the assembler code to know that TOS is always in register bx for fastest access. **0!** and **1+!** (address →) zero and increment address, respectively. **execute@** and **10/** are simply optimized versions of **@ execute** and **10 /**.

The dialect lost its case statement somewhere, so I make do with arrays holding the execution address (**' <name> cfa**) of the suite of choices. This provides a choice of operators selected by an index value – adequate for present needs – with nearly no overhead cost.

Block 1 sets up the requisite storage (but beware of overrunning the arrays with longer distributions!), and some debugging words to print out various intermediates. If random numbers larger than FF are needed, **ran.tab** (which is the a{*m*} mentioned above) must be an **array** instead of a **carray**.

Block 2 extracts single digits from 4-digit hex numbers and either leaves them in place or moves them far right. This could all be done with masks and division in high level.

Block 3 sets up vectored execution of the words on Block 2, and adds a general right-shift-digit utility.

4pis and **4sigs** on Block 4 calclulate the various functions described above, while **!table** fills the table from which the random numbers will be drawn. **!offset** , which calls **4pis** and **4sigs**, also puts a minimum value for the size of **ran.tab** into **Size**.

Block 5 shows one way around the RNG problem. The dialect having lost its own RNG, I had to build one. Since both high- and low-order bits of a linear congruent RNG are nonrandom, these are discarded. To extend the range, two independent RNGs are added together. I make no claims about this one except that it works better than some 30 related attempts. An RNG must be at least this good to work with marsaglian extraction: If you have a better RNG, by all means use it, or better yet, send me a copy.

shaper on Block 6 does the real work, taking a random number and converting it into a table entry. The rest of the block, except for **m.setup**, is utility words. **h.binomial m.setup** reproduces the binomial example from **[ABRA65]**. **.text** emits an ASCII character instead of a number. **.table** prints the marsaglian table, which for illustrative purposes only we show with the letters which it will later produce. The possibly curious nature of the printing words comes from using a Macintosh™, which knows nothing about columns or tabs, as a front end to a National Instrument's MacBus™ (over a SCSI link with Creative Solutions/Pixelwerk's BusTalk™) running an NEC V50™ (IBM-AT™/80286™ clone) which is actually doing the work – all this, again, for good but irrelevant reasons.

Acknowledgement

This research was supported by a grant from A/S Pixelwerks Ltd, which is gratefully acknowledged.

References

[ABRA65] Abramowitz, M and Stegun, I.A., 1965. *Handbook of Mathematical Functions.* (Dover, NY).

[BOX58] Box, G.E.P., and Muller, M.E., 1958. A note on the generation of random normal deviates. *Ann. Math. Inst.* **10**:610.

[MARS62] Marsaglia, G., 1962. Random variables and computers. *Proc. 3rd Prog. Conf. on Probability Theory.* Also *Math. Note* **#260**, Boeing Scientific Res. Labs.

[VONN51] Von Neumann, J., 1951. Various techniques used in conjunction with random digits, Monte Carlo method. Nat. Bur. Stds. (US) *Appl. Math. Series* **12**:36.

Ferren MacIntyre, a high-school dropout with an eventual PhD in Physical Chemistry from MIT, was a research professor of Chemical Oceanography at the time this article was written. Finding the "Peer Review" process – on which salary and research money depended – to be an operational oxymoron, he is now active in Forth-and-Macintosh-based image analysis with Imaging Norway, Postboks 1008, Bergen, N-5001 Norway. The pay is no better, but time is better spent now and the aggravation level is a whole lot lower.*

**That is a polite way of saying that the number of peers (who understand his intentionally obscure specialty) is very small, and reviews seldom went to them.*

Listing 1. Source code for Marsaglian random numbers.

```
( 1.Constants, arrays, examples, printers) : it ; ( Ø4Ø188  FM)
1ØØ  array dist       ( 4-digit table of the random distribution)
  4  array m.offs     ( Index-corrections)
  4  array m.comp     ( Table of comparison values)
8ØØ carray ran.tab    ( Table from which random #s are drawn)
Ø constant Cnt        ( # of entries in distribution)
Ø constant Indx       ( Running index into random table)
Ø constant Size       ( Minumum size for ran.tab )
: h.binomial hex     3 4Ø 2ØØ  8ØØ 1ØØØ  CCD  6 ;
: !dist     dup ' Cnt ! Ø do i dist ! loop ;
: .dist     Ø dist 4Ø dump ;
: .off      4 Ø do i m.offs @ 5 u.r loop ;
: .comp     4 Ø do i m.comp @ 5 u.r loop ;
: ?cr ( i cnt -> | cr every cnt entries) mod not if cr then ;

( 2.Digit extraction)                              ( Ø4Ø188  FM)
( Move digit to right: 1234 -> 1, 2, 3, 4)
code rd4 ( d1d2d3d4 -> d4) bx ØØØF iw and. end-code
code rd3 ( d1d2d3d4 -> d3) bx ØØFØ iw and.  cx  4 iw mov.
        ax bx xchg. ax  4 shr. ax bx xchg. end-code
code rd2 ( d1d2d3d4 -> d2) bx ØFØØ iw and.  cx  8 iw mov.
        ax bx xchg. ax  8 shr. ax bx xchg. end-code
code rd1 ( d1d2d3d4 -> d1) bx FØØØ iw and.  cx ØC iw mov.
        ax bx xchg. ax ØC shr. ax bx xchg. end-code

( Leave digit in place: 1234 - 1ØØØ, 2ØØ, 3Ø, 4)
code od3 ( d1d2d3d4 -> ØØd3Ø) bx ØØFØ iw and. end-code
code od2 ( d1d2d3d4 -> Ød2ØØ) bx ØFØØ iw and. end-code
code od1 ( d1d2d3d4 -> d1ØØØ) bx FØØØ iw and. end-code

( 3.Execution vectors. r.shift                     ( Ø4Ø188  FM)
create rs.vec ' rd1 cfa , ' rd2 cfa , ' rd3 cfa , ' rd4 cfa ,
create ns.vec ' od1 cfa , ' od2 cfa , ' od3 cfa , ' rd4 cfa ,
( Instant death to execute from outside the range, so enforce)
: digit.extract ( dddd i-> ...ØØØdi)
        Ø max 3 min   2* rs.vec + execute@ ;
: digit.isolate ( dddd i-> ..ØdiØ..)
        Ø max 3 min   2* ns.vec + execute@ ;

code [r.shift] ( n -> n/1Ø ) cx 4 iw mov. ax bx xchg.
        ax 4 shr. ax bx xchg. end-code
: r.shift ( n i -> n/1Ø**i)
        dup if Ø do [r.shift] loop else drop then ;
```

```
( 4.Calculate and ! m.offs, m.comp, marsaglian table Ø4Ø188  FM)
: 4pis  ( -> ) Ø   4 Ø do Ø
       Cnt Ø do i dist @ j digit.extract + loop + dup dup 5 u.r
       negate                  i m.offs  ! loop ' Size ! ;
: 4sigs ( -> ) Ø   4 Ø do Ø
       Cnt Ø do i dist @ j digit.isolate + loop + dup dup 5 u.r
       dup                     i m.comp  !
       swap 3 i -  r.shift     i m.offs +! loop drop ;
: !offset   4pis cr 4sigs cr .off cr ;
: !it   ( n -> )  Indx ran.tab c! ' Indx 1+! ;
: !table  ' Indx Ø! 4 Ø do             ( over the digits)
    Cnt Ø do                           ( over the distribution)
     i dist @ j digit.extract          ( # of locations to fill)
 ?dup if Ø do j !it loop then          ( Skip loop on Ø Ø lims)
    loop loop ;

( 5.Inelegant random-number generator)           ( Ø4Ø188  FM)
  9A constant Mul1            1417 constant See1
  A7 constant Mul2            25A3 constant See2
: rndm1 ( n -> random# <= n)
   See1  Mul1 * 5 + dup 2/ ' See1 ! swap
   ?dup if mod then abs ;
: rndm2 ( n -> random# <= n)
   See2  Mul2 * dup 2/ ' See2 ! swap
   ?dup if mod then abs ;
: .rn1  1ØØ Ø do i 1Ø ?cr 8ØØØ rndm1 5 u.r loop ;
: .rn2  1ØØ Ø do i 1Ø ?cr    2 rndm2 5 u.r loop ;
: .ran  1ØØ Ø do i 1Ø ?cr 8ØØØ rndm1 2 rndm2 8ØØØ * +
       2 rndm2 + 5 u.r loop ;
: m.random    8ØØØ rndm1 2 rndm2 8ØØØ * +  2 rndm2 + ;
: r.test 8Ø Ø do  m.random dup rd1 i 8 ?cr 5 u.r 5 u.r loop ;

( 6.Text sampler)                                ( Ø4Ø188  FM)
: shaper   dup Ø m.comp @ u< if rd1                    else
           dup 1 m.comp @ u< if 2 r.shift 1 m.offs @ - else
           dup 2 m.comp @ u< if 1 r.shift 2 m.offs @ - else
                                      3 m.offs @ -
   then then then  ran.tab c@ ;
: .text  i' 4Ø ?cr 61 + dup 7b = if drop space else emit then ;
: .table     Indx Ø do i ran.tab c@      .text loop cr ;
: flat.text   2ØØ 1 do 36Ø m.random 2Ø / .text loop cr ;
: shaped.text 2ØØ 1 do   m.random shaper .text loop cr ;
: m.setup  ( dist cnt ->) !dist !offset !table ;
: letters 315C 48 3F1 27 553 1E5 5FØ 147D B65 B4B 2D 2DØ 1ØDA
  EB9 51E 79A 1AA 2Ø D29 E3Ø 518 46D 1946 9B4 45A 381 1129 1B ;
letters m.setup
```