Some portable very-long-period random number generators

George Marsaglia and Arif Zaman
Supercomputer Computations Research Institute and Department of Statistics, The Florida State University, Tallahassee, Florida 32306

(Received 11 December 1992; accepted 11 February 1993)

It is found that a proposed random number generator ran2, recently presented in the *Numerical Recipes* column [W. H. Press and S. A. Teukolsky, Comput. Phys. **6**, 521–524 (1992)], is a good one, but a number of generators are presented that are at least as good and are simpler, much faster, and with periods "billions and billions" of times longer. They are presented not necessarily to supplant ran2, but to put it in perspective. Any serious user of Monte Carlo methods should have a variety of random number generators from which to choose. In addition to two specific programs, one in Fortran and one in C, a framework is offered within which the readers can easily fashion their own generators with periods ranging from 10^{27}–10^{101}.

INTRODUCTION

In the *Numerical Recipes* column of this Journal, the Editors, Press and Teukolsky, presented a portable random number generator, ran2 (Ref. 1). Since we have come to expect good things from the Numerical Recipes Project, ran2 is likely to be widely used. That's o.k., it is a good generator. But approval through that column invites constructive dissent; we offer some here.

Our comments relate mainly to the algorithms used in ran2. They were suggested by L'Ecuyer.[2] Implementation of the algorithms was provided in the *Numerical Recipes* column, and they are discussed in more detail in the Editors' book(s), *Numerical Recipes in Fortran(C)*.[3]

We first suggest considerations that might have made ran2 better. Then we suggest several alternatives that are simpler, faster, and have longer periods.

The appellation "portable" is not precise. It generally means that the program will compile and run properly on most computers, but brevity and simplicity are implicit criteria. The essence of ran2's portability is the means to carry out modular arithmetic on 31-bit integers without overflow on multiplication. This is necessary for users not wanting to—or not willing to—use the automatic integer arithmetic modulo 2^{32} that is built into most modern 2's complement CPU's.

But circumventing the CPU's built-in capabilities does not come cheap—at least for ran2. It uses three integer divisions, six limited multiplications, and from four to six additions for each random number, as well as the manipulations necessary to address, fetch, and store elements in a table.

The built-in arithmetic modulo 2^{32} that most modern CPU's provide is, in our view, a great asset that should be exploited for random number generation. The authors of ran2 view its use as "abusing the compiler," although their remark may have been tongue-in-cheek. So we will suggest methods for two levels of "portability": one—for the timid or disadvantaged—that circumvents multiplicative overflow in what we think is a much simpler way than that of ran2; and the other for those millions of users who can take advantage of such a marvelous way of generating a nearly satisfactory random number by means of a simple statement such as n=69069*n. This latter method is often the one that is implemented in system generators—for example, in CDC's and Vax's, but by modern standards the period is too short and trailing bits are unsatisfactory.

Taking advantage of such a remarkably simple, fast and nearly satisfactory method for our first step, then combining it with one of a number of simple very-long-period generators based on addition or subtraction, provides a composite generator that we rank highly in categories of speed, simplicity, very long periods and satisfactory performance on tests of randomness. We hope that—after reading this and perhaps trying some of your own versions—you will agree.

We provide a number of ways to achieve such combinations. And a simple fix allows them to be used with compilers that forbid overflow on 32-bit multiplication.

In using a deterministic sequence to simulate randomness we are all, as Von Neumann said, "in a state of sin." Every deterministic sequence will have problems for which it gives bad results, and a researcher should be able to call on a variety of random number generators for comparing results. While interesting theory can lead to new kinds of random number generation, in the end it is, alas, an empirical science. Only through collective experience can we hope to reach a state where we may "go forth and sin no more."

I. COMMENTS ON ran2

The generator `ran2` uses two 31-bit congruential sequences: $x_1, x_2, x_3, \ldots$ and $y_1, y_2, y_3, \ldots$. It keeps the x's shuffled in a table: $iy(1), \ldots, iy(32)$. To get a new random 31-bit integer, it generates a new y, uses y to form j in $\{1,\ldots,32\}$ with an integer division and an add, outputs $iy(j)-y$ and stores a new x in location j. (Note that indexing the table from 0 to 31 would save an addition when forming j, while "and"-ing y with 31 would be a much faster way to produce a j.)

The key device of `ran2`, in our view, is subtraction of the elements of the two congruential sequences. There is considerable empirical evidence, and theoretical support, for the belief that combining two simple random number generators produces a better generator. See, for example, Ref. 4. We prefer to combine sequences with two entirely different algebraic structures, rather than two congruential sequences, as in `ran2`. And we think a satisfactory combination can be provided with much less cost than in `ran2`.

As for the shuffling: It helps. One of us (GM) invented it in the early 1960s, but abandoned it later. If you go to the trouble of maintaining a table of recent random numbers and indices, you might as well get the benefits of equally good results and far longer period with lagged Fibonacci or subtract-with-borrow[4,5] or some such method. (With a table of 32 words you can get a period on the order of 2^{1024} rather than the 2^{62} of `ran2`. One of our examples below provides a period of 2^{309} from a table of ten 31-bit integers.)

Still, if you prefer the simple shuffle used in `ran2`, outputting differences, you have the choice of the two congruential sequences. In `ran2` they are

$$x_n = 40014 x_{n-1} \bmod 2^{31} - 85$$

and

$$y_n = 40692 y_{n-1} \bmod 2^{31} - 249.$$

We suggest that you might consider

(1) Using moduli of 32 rather than 31 bits. This takes advantage of the full computer word, and it costs no more—indeed, often less. In high-level languages that view negative integers as those with leading bit 1, you then get random signed integers, which may be floated to get uniform reals on either (0,1) or (−1,1). The latter are more desirable in many simulations.

(2) If you still want prime moduli of 31 bits, why not make then *safeprimes*, and choose the multipliers to provide fast implementation? A safe prime p is one for which $(p-1)/2$ is also prime. Then half of the residues of p are primitive roots. Factors of $p-1$ are undesirable; they provide periodic subsequences of the full period. If 2 is a primitive root of a safeprime p, then so is every odd power of 2, providing multipliers that simplify the generating procedure. If the generators are to be used in combinations, their lattices are of no importance. The two greatest safeprimes of 31 bits are $2^{31}-69$ and $2^{31}-535$. Both have 2 as a primitive root, so any odd power of 2 may be used as a multiplier, providing overflow-free multiplication by a shift, and thus avoiding the manipulations of `ran2`. The two largest safeprimes of 32 bits are $2^{32}-209$ and $2^{32}-1409$. While 2 is not a primitive root of either, $2^{15}+1$ is for the larger and $2^{14}-1$ for the smaller one, as are others within ± 1 of a power of 2. They provide overflow-free multiplication by means of shifts and adds.

(3) Whatever method you use to generate and combine two sequences, there is the problem of how to handle arguments and returned values in the resulting subroutine. With the Fortran `ran2` you use `ran2(idum)` in expressions in the main program, initializing with idum negative. The value of idum is maintained in the main program and changed by the subroutine `ran2`, which is delivered an address rather than a value. Thus `ran2` can only be called from the main program. The subroutine must deal with both an argument idum and the returned value for `ran2`. Inefficient. The modern trend for random number generators is to call them with an empty argument list, such as `u=ran2( )`. The problem of initializing is handled by an entry point in the subroutine, and default entry values are put in the subroutine to make its use fool proof. We illustrate such a structure in our program `mzran`, below.

(4) There is another point that bears consideration. Users of a random number generator might want a random positive integer, a random signed integer, a random real on [0,1) or on (−1,1), or even a random byte for addressing a table. All these can be provided if the underlying generator is for 32- rather than 31-bit integers, as mentioned above, but even if the generator produces a 31-bit integer, the practice of multiplying by `.5**31` then returning a real on [0,1) (or (0,1) after a fix), "wastes" many of the bits of the integer, and makes reconstructing a (necessarily limited) random integer costly. Why not have the subroutine return an integer (preferably, a signed integer), as, say, `iran2( )`? Then users would have the full capabilities of generating random integers or masked parts of integers, or uniform reals UNI on [0,1) or VNI on (−1,1) by the simple expedient of Fortran statement functions such as `UNI( ) =.5+.2328306e -9*iran2( )` or `VNI( ) =.4656613e-9*iran2( )`. These would be coded in-line, and cost no more than would their (irreversible) inclusion in the subroutine itself.

II. SOME NEW GENERATORS

From our point of view, a random number generator is just a set of computer instructions that combine parts of a current set of integers to provide a new integer to serve as the output of the generator. This is usually done in a subprogram; before a return to the main program, the current set is updated to provide for the next iteration.

If only a few values are involved, the current set might be designated, say `a, b, c, d,` with a new `n` formed from them, then "promotions": `a=b; b=c; c=d; d=n` form the new current set. If the number of elements in the current set is more than perhaps 5 or 6, the current set is maintained in a "circular" table, with pointers used to indicate elements to be used in the generating procedure.

In principle, if there are r elements in the current set of 32-bit integers, it is possible to produce a sequence of 2^{32r} integers before repetition—that is, before the initial (seed) set reappears. We will provide numerous examples that attain that period or come very close to it, or which attain the

maximal period that one fewer seed value would provide, in a few cases where that simplifies the generating process.

The constraints we have in developing such examples are: (1) The resulting computer operations must be simple and fast, and (2) the underlying mathematics must be such that we can rigorously establish the period.

Then, guided by experience and theory that suggest combining two different generators produces an even better one, we choose two different generators and combine them—usually by addition or subtraction—to form the composite generator, whose period will be the product of the component periods, or close to it. (Periods often have a few powers of 2 in common, reducing the lcm.)

Experience has shown better results when the two generators to be combined use very different arithmetic. Thus, in the list below, we stick to combining one of the first two generators, which use multiplication, with one of the last 14, all of which use addition or subtraction: in all, 28 generators. We have not yet tested all such combinations, but those we have tried so far have passed all tests for uniformity and independence.

Some candidates for combination generators

No.	Modulus	Sequence	Approx. period
(1)	2^{32}	$x_n = 69069x_{n-1} + \text{oddconst}$	2^{32}
(2)	2^{32}	$x_n = x_{n-1} * x_{n-2}$	2^{31}
(3)	2^{32}	$x_n = x_{n-1} + x_{n-2} + \text{``}c\text{''}$	2^{58}
(4)	2^{31}	$x_n = x_{n-1} + x_{n-2} + \text{``}c\text{''}$	2^{59}
(5)	2^{31}	$x_n = x_{n-2} + x_{n-3} + \text{``}c\text{''}$	2^{86}
(6)	$2^{31}-69$	$x_n = x_{n-3} - x_{n-1}$	2^{62}
(7)	$2^{31}-69$	$x_n = x_{n-4} - x_{n-1}$	2^{94}
(8)	$2^{31}-61$	$x_n = 2x_{n-3} - x_{n-2} - x_{n-1}$	2^{93}
(9)	$2^{31}-69$	$x_n = x_{n-3} - 2x_{n-4}$	2^{124}
(10)	$2^{31}-1$	$x_n = x_{n-4} - x_{n-5} - \text{``}c\text{''}$	2^{155}
(11)	$2^{31}-5$	$x_n = x_{n-8} - x_{n-10} - \text{``}c\text{''}$	2^{307}
(12)	$2^{32}-10$	$x_n = x_{n-2} - x_{n-5} - \text{``}c\text{''}$	2^{160}
(13)	$2^{32}-18$	$x_n = x_{n-2} - x_{n-3} - \text{``}c\text{''}$	2^{95}
(14)	$2^{32}-5$	$x_n = x_{n-1} - 2x_{n-2}$	2^{64}
(15)	$2^{32}-5$	$x_n = x_{n-1} + x_{n-2} - 2x_{n-3}$	2^{95}
(16)	$2^{32}-5$	$x_n = 2x_{n-5} - x_{n-4} - x_{n-1}$	2^{160}

Comments: (1)–(2) use 32-bit multiplication [but (1) can be implemented so as to avoid overflow—see below]. (3)–(5) are add-with-carry and (10)–(13) are subtract-with-borrow generators, described in the next section. (4)–(11) are for 31-bit integers and may be implemented in Fortran or other languages that have only signed integers, while (12)–(16) are best suited for machine language or C implementations, which provide for 32-bit positive (long) integers.

III. SEQUENCES SUCH AS $x_n = x_{n-1} + x_{n-2} + \text{``}c\text{''} \bmod 2^{31}$

Several generators on the above list, with a c in quotes, are add-with-carry (AWC) or subtract-with-borrow (SWB) generators described elsewhere.[4] They were developed to provide immensely long periods, on the order of 2^{1500} or more. But simpler versions may be used to provide candidates suitable for the above list. Consider, for example, the AWC generator (4): $x_n = x_{n-1} + x_{n-2} + \text{``}c\text{''} \bmod 2^{31}$. The c here is the carry bit, not a constant. It may change with each call, depending on whether the addition modulo 2^{31} produces overflow. An implementation works as follows: with a current pair i,j and current c, form $s = i + j + c$. If that sum exceeds the modulus, $m = 2^{31}$, replace s by $s - m$ and set $c = 1$, else keep s and set $c = 0$. Then promote: $i = j$; $j = s$, output s, and the new i, j, c are ready for the next generating process. We choose $m = 2^{31}$ for use with Fortran compilers, while 2^{32} is better for C implementations or for machine language. With $m = 2^{32}$, such AWC generators are particularly well suited for machine language implementation, since the carry bit is automatically set with each addition.

Implementation for the SWB generators such as (12) are similar, except that one does the subtraction and adds m if the result is negative, with the new c 0 or 1 accordingly. The essence of an implementation of (12) has modulus $m = 2^{32} - 10$, five current integer values, say, v, w, x, y, z, and current c. The generating procedure is: if $y < v + c$, then $s = v + c - y - 10$ and $c = 1$, else $s = y - v - c$ and $c = 0$. Then s is the output and promotions provide the new set of five: $v = w$; $w = x$; $x = y$; $y = s$. The period is 2^{160} for any five initial values, not all zero.

All are assumed 32-bit positive integers, routine for C but not for Fortran. A Fortran implementation is possible but too tricky; better is to use one such as (10) with 31-bit integers, getting 2^{155} five-tuples out of any five 31-bit seeds not all zero.

IV. THE GENERATOR mzran

We now give details of a generator that combines sequences (1) and (6) from the list. For any choice of seed value for (1) and any three seed values for (6), not all zero, the composite period is $2^{32}(p^2 + p + 1) > 2^{94}$, with $p = 2^{31} - 69$.

This is perhaps the fastest of the possible combinations for Fortran, although not by much—all of them are quite fast. Versions in C will be faster because one need not accommodate Fortran's quaint insistence that an integer with leading bit 1 is negative.

Other combinations will have a similar structure, only details on maintaining current values and arithmetic will vary. The reader is invited to try his own combinations: As with a Chinese restaurant menu, choose one from category (1)–(2) and one from (3)–(13). Combine them and savor the resulting combination.

We use the constant 1013904243 for sequence (1), but any odd constant will do (as will almost any multiplier congruent to $\pm 3 \bmod 8$). If `n` is the current integer in this sequence, the next `n` may be generated with the single instruction `n=69069 *n+1013904243.` For For those compilers without a switch that permits the full modulo 2^{32} arithmetic inherent in modern CPU's, this can be effected with shifts and masks—see below.

We suggest this implementation in Fortran

```
function mzran()
save i,j,k,n
data i,j,k,n/521288629,362436069,16163801,1131199299/
mzran=i-k
if(mzran.lt.0) mzran=mzran+2147483579
i=j
j=k
k=mzran
n=69069*n+1013904243
mzran=mzran+n
return
entry mzranset(is,js,ks,ns)
i=1+iabs(is)
j=1+iabs(js)
k=1+iabs(ks)
n=ns
mzranset=n
return
end
```

Implementing `mzran` requires consideration of the way that random number subroutines are likely to be used. As mentioned in our comments on `ran2`, we think the subroutine should be invoked with an empty argument, as in `mzran( )`, and that an entry statement should be used to set seed values, with default values provided to make usage foolproof. And a random 32-bit integer should be returned, leaving the option of random reals on (0,1) or $(-1,1)$ to the user through statement functions such as `UNI( )=.5+.2328306e-9*mzran( )` or `VNI( )=.4656613e-9*mzran( )`. This costs no more and provides far greater versatility.

Default seed values for i, j, k, n are included, via a data statement, for users who forget to, or do not care to, or do not know how to, initialize by using a statement in the main program such as `m=mzranset(is, js, ks, ns)`, with `is, js, ks, ns` any four legal Fortran integers.

Note that `mzran` can be made to work without taking advantage of 32-bit multiplication. Merely replace the segment `n=69069*n` by this segment:

```
n=ishft(3533*ishft(n,-16)+iand(n,65535),16)+3533*iand(n,65535)+1
```

It slows the routine down a bit, but still makes `mzran` many times faster than `ran2`. If your compiler will not allow overflow on 32-bit multiplication, substitute the above segment. (The names of shift and "and" functions may vary with some compilers.)

Note that `mzran` combines the congruential generator, which produces 32-bit integers, with the generator (6), which produces 31-bit integers. It takes the sum modulo 2^{32}. That might seem strange, but the reader should be able to convince himself that: *if x is a uniform random integer on $\{0,...,a-1\}$, and y is an independent random integer with any possible distribution, then $x+y$ (or $x-y$) modulo a is uniform on $\{0,...,a-1\}$.*

Because the periods of (1) and (6) are relatively prime, the period of the composite `mzran` is $2^{32}(p^2+p+1)$, which is about 2^{94} or 10^{28}.

V. AN IMPLEMENTATION IN C

We now give an example of a combination generator in the programming language C, in which we have 32-bit positive integers. We combine (1) and (13) from the above list. The period of (1) is 2^{32} and that of (13) is $(p^3-p^2)/3$, about 2^{95}, with $p=2^{32}-18$. So the composite program, say `mzran13`, has period some 2^{125}. And, as with `mzran` and other combinations of (1)–(2) with (3)–(16), it produces eminently satisfactory random 32-bit integers.

We again use our recommended structure for a random number generator: An empty argument list, default seed values and an entry to set the seed values. Here is the program:

```
typedef unsigned long int unlong;
unlong x=521288629, y=362436069, z=16163801, c=1,
       n=1131199209;
ulong mzran13()
{ long int s;
  if (y>x+c) {s=y-(x+c); c=0;}
  else { s=y-(x+c)-18; c=1; }
  x=y; y=z; return (z=s) + (n-69069*n+1013904243);
};
void ran13set(ulong xx, ulong yy, ulong zz, long nn)
{ x=xx; y=yy; z=zz; n=nn; c=y>z; }
```

This program is a bit tricky to figure out, but potential users can verify it by comparing with a more transparent version that combines (1) and (13) with the specified seed values, perhaps in double precision Fortran or in C with some redundant but more expository code.

VI. OTHER COMBINATIONS

Many other combinations are possible and worth considering. For example, for years we used a generator called COMBO as our absolutely-reliable-against-which-all-others-are-compared generator. It combines sequence (2) with the sequence $x_n=x_{n-1}-x_{n-3} \bmod 2^{30}-35$ (not listed—its period is "only" some 2^{57}). Over the years, COMBO has resisted all efforts to refute its randomness, and frequently a disparity between its results and those of another generator have shown the inadequacy of the latter. (Not all tests of randomness are based on problems for which we know the underlying probabilities. For many tough problems we can only compare results from different generators—another reason for having a variety of generators available.)

Sequence (2) seems as good as (1) for combining with one (or more) of those methods (3)–(16) based on addition or subtraction. It has period 3×2^{29} for any two odd seed values, not both 1.

Sequence (1) has been widely—and successfully—used by itself. It is, for example, the system generator on Vax's, and it, combined with a shift-register generator, provides the widely used McGill random number generator Super Duper. The shift register used in Super Duper may also be considered for use in combinations with one of (3)–(12) that use addition or subtraction. Effected by the two Fortran statements

```
n=ieor(n,ishft(n,-15)); n=ieor(n,ishft(n,17))
```

its period is $2^{32}-2^{21}-2^{11}+1$. It has provided seemingly as good combinations as (1) or (2) with (3)–(16), but we tend

to prefer multiplication over shifts and exclusive-or's for scrambling bits. But if you want to try using it in place of (1) or (2), you get another 14 combinations, making a total of 42 from which to choose.

VII. SUMMARY

We agree that the generator `ran2`, advocated by L'Ecuyer[2] and implemented in the *Numerical Recipes* column[1] and books,[3] is a good one, but suggest that it could be made better, and simpler, by choosing different multipliers and moduli for the two congruential generators it uses. We further suggest that 32-bit generators are preferable (and cost no more), and that the generator should be called with an empty argument list, with seed values set by a separate entry and default seed values provided.

Then we propose considering up to 28 different kinds of new generators that are simpler than `ran2` and have much longer periods. Two exemplary programs, `mzran` in Fortran and `mzran13` in C, are offered. They have our recommended calling procedure, entry, and default values for seeds. Furthermore, `mzran` may be easily modified to make it meet the most restrictive requirements of portability: no multiplications that exceed 32 bits.

Spurred by the challenge put forward by Press and Teukolsky to refute the randomness of `ran2`, we have subjected it to our battery of tests DIEHARD, many of which are described in Ref. 4. It has passed all of them, but so too have `mzran`, `mzran13` and various other combinations of sequences (1) or (2) with one from (3)–(16) in the above list. Since all are simpler, faster, provide a greater variety of random output and have longer periods, we recommend readers consider adopting them—not necessarily to replace `ran2`, but perhaps to put it in perspective. Everyone who does serious Monte Carlo research should have several methods available.

REFERENCES

1. W. H. Press and S. A. Teukolsky, "Portable random number generators," Comput. Phys. **6**, 521–524 (1992).
2. P. L'Ecuyer, "Efficient and portable random number generators," Commun. ACM **31** (6), 742–774 (1988).
3. W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in Fortran and Numerical Recipes in C* (Cambridge U.P., Cambridge, England, 1992), 2nd. ed.
4. G. Marsaglia, "A current view of random number generators, keynote address," *Proceedings, Computer Science and Statistics: 16th Symposium on the Interface* (Elsevier, 1985).
5. G. Marsaglia and A. Zaman, "A new class of random number generators," Ann. Appl. Probability **1** (3), 462–480 (1991).