

More PRNGs for Calculators

By

Namir C. Shammass

CONTENTS

Introduction	1
The Penalty Factor	2
Legacy Calculator PRNG Algorithm.....	10
New Calculator PRNGs	10
The Proposed PRNGs	11
Notes	17
Conclusions	17
Document History	19

INTRODUCTION

This paper looks at pseudo-random number generators (PRNGs) for calculators using tools developed with Matlab. The study uses a popular PRNG for calculators as the baseline to compare with the collection of new PRNGs for calculators.

PRNGs for calculators do not require the kind of strict randomness needed by computer applications, such as simulation or cryptography. Thus, the bar for calculator PRNGs is lower than that for computer PRNGs. Using PRNGs for simple calculator games or simulations require the generation of far fewer random numbers than the computer-based application counterparts. Thus, the cycle size of PRNGs, where random numbers start to repeat, is not a critical issue.

The purpose of PRNG functions is to take an input, usually the current random number (a positive floating-point number) and generate a new random number. The quality of generating PRNGs depends on the *chaos* created by the PRNG function. The more chaotic the output, the better is the PRNG function.

There are two types of PRNG functions. The first type relies on mostly integer calculations—the final step involves dividing a random integer by a big integer to generate a uniformly distributed random number in the range of $[0, 1]$. Integer-based PRNG functions depend on the maximum integer supported by the operating system. The good selling point of such functions is they are quite portable across different operating systems, hardware, and, in a good number of cases, different programming languages. The second type of PRNG functions works with floating-point numbers. As such, they are dependent on the number of decimals supported and number rounding. These functions are less portable than their integer-based counterparts. This study focuses on the second type of PRNG functions. I used Matlab because it quickly handled large arrays.

It is worth pointing out that designing good integer based PRNGs ranges between difficult and hard. Designing floating-point-based PRNGs is easier. In the end, you get what you pay for!

THE PENALTY FACTOR

There are several tests for measuring the randomness of a sequence of uniformly distributed random numbers. The most famous battery of tests is the *diehard* test. I have devised my own test which calculates a penalty factor. The lower this factor the better the sequence of random number generated. The values for a calculated penalty factor depend on the count of random numbers generated. This study is based on consistently generating numerous batches containing sequences of 10,000 random numbers. Lowering the count of random numbers generated tends to increase the values for the factors. One reason is the random numbers may appear slightly more auto correlated when they are fewer of them. The values for the factor depend on the following statistics related to the random numbers generated:

- The mean.
- The standard deviation.
- The maximum and minimum autocorrelations taken for 1 to 100 lags.
- The Chi-square statistic for a ten-bin histogram counting random numbers in bins of 0.1 width, between 0 and 1. I will call this statistic as ChiSqr10. The expected value in each bin equals the count of random numbers divided by 10.
- The Chi-square statistic for a twenty-bin histogram counting random numbers in bins of 0.05 width, between 0 and 1. I will call this statistic as

ChiSqr20. The expected value in each bin equals the count of random numbers divided by 20.

- The sum of product of autocorrelations (distributed in 20 equal-sized bins ranging from the minimum to the maximum autocorrelations) and their counts. Thus, the size of the bins is dynamic and depends on the distribution of the autocorrelations. I will call this statistic AutoCorrSum.
- The change-of-sign statistic. I discuss calculating this statistic below.
- Kolmogorov-Smirnov statistics. This part calculates the following two values:
 - $K_+ = \max(F_n(x) - F(x))$
 - $K_- = \max(F(x) - F_n(x))$

Where $F_n(x) = (\text{number of } x_i \leq x)/n$ and $F(x)$ is the theoretical cumulative distribution value.

Regarding the change-of-sign statistic, I examine the change of signs between the consecutive differences in the random numbers. An ideal PRNG would have the consecutive signs constantly and systematically alternating between positive and negative. However, real-world PRNGs will have the consecutive signs of the differences change few elements down. Let $D(n,1)$ be the number of change of signs from negative to positive every n differences. Also let, $D(n,2)$ be the number of change of signs from positive to negative every n differences. These values decrease exponentially with n and are highest at n equal 1. I calculate the chsStat as:

$$\text{chsStat} = (\sum D(i,1) \cdot i) / D(1,1) + (\sum D(i,2) \cdot i) / D(1,2) \text{ for } i=2, \dots, n \quad (1)$$

The values $D(1,1)$ and $D(1,2)$ will normalize the ratios and thus take care of the effect of the number of random numbers generated. An ideal PRNG will have $D(i,1)$ and $D(i,2)$ as zeros for all $i > 1$, yielding a chsStat value of 0. Multiplying $D(i,1)$ and $D(i,2)$ by i is a way to penalize larger delays in the change of signs. One can also multiply the values of $D(i,1)$ and $D(i,2)$ by I squared or some other power. Using powers greater than one serve only to magnify the effect delayed changes of signs.

I calculate the penalty factor using:

$$\text{Factor} = 1000 [|\text{mean} - 0.5| + |\text{sdev} - 1/\sqrt{12}|] + 100 (\text{max_autoCorrel} - \text{min_autoCorrel}) + 100 \cdot \text{AutoCorrSum} +$$

$$\text{ChiSqr10} + \text{ChiSqr20} / 2 + 10 \cdot \text{chsStat} + 10 (K_+ + K_-) \quad (2)$$

Equation (2) calculates the penalty factor by including the following weighted terms:

- One thousand (the weight) times the sum of the following sub-terms:
 - The absolute difference between the mean and its expected value, 0.5.
 - The absolute difference between the standard deviation and its expected value, $1/\sqrt{12}$.
- One hundred (the weight) times difference between the maximum and minimum autocorrelation values. The maximum and minimum autocorrelations have positive and negative values, respectively. This term adds a special penalty for the extreme autocorrelation values.
- One hundred (the weight) times the value of the statistic AutoCorrSum. This term adds a special penalty for the general autocorrelation values. A dispersed distribution of the autocorrelation values contributes to a higher factor value. By contrast, a distribution of the autocorrelation values concentrated near zero, contributes little to the factor value.
- The value of the ChiSqr10 statistic.
- Half the value of the ChiSqr20 statistic.
- Ten times the change-of-sign statistic.
- Ten times the sum of the K_+ and K_- values.

Thus, the calculated penalty factor measures the following:

- The deviation from the expected basic statistics (mean and standard deviation).
- The goodness of distribution for the random numbers.
- The level of the autocorrelations.
- The change of sign of the differences between random numbers.
- The closeness of the cumulative distribution of the numbers generated to the ideal cumulative distribution.

Here is a sample Matlab function that test a PRNG algorithms, performs the various statistics and returns a penalty factor.

```
function factor = rngRandoDoral(maxElems,bShowResults)
%UNTITLED2 Summary of this function goes here

if ~exist('bShowResults','var') || isempty(bShowResults)
    bShowResults=false;
```

```

end
if bShowResults, fprintf('r + 1/(phi+r))*997 rng test\n'); end
x=zeros(maxElems,1);
rng('shuffle','twister');
x(1)=rand;
for j=2:maxElems
    x(j) = rando(x(j-1));
end
factor=calcFactor(x,bShowResults);
if isnan(factor), factor=1e99; end
end

function x = frac(x)
    x=x-fix(x);
end

function r=rando(r)
    r = mod(997/r,1);
end

function factor = calcFactor(x, bShowResults)
% Calculate the factor statistic for the array of random nnumbers x.

    if nargin < 2, bShowResults = false; end
    maxElems=length(x);
    meanx=mean(x);
    sdevx=std(x);
    % get the first 100 autocorrelation values
    acArr=autocorrArr(x,1,100);
    % calculate the chisquare for the 10-bin histogram
    numBins=10;
    expval=maxElems/numBins;
    [N1,ev1]=histcounts(x,numBins);
    chiSq10=sum((N1-expval).^2/expval);
    numBins=20;
    expval=maxElems/numBins;
    [N2,ev2]=histcounts(x,numBins);
    chiSq20=sum((N2-expval).^2/expval);
    numBins=20;
    [N3,ev3]=histcounts(acArr,numBins);
    ev3c=ev3(2:length(ev3));
    autoCorrSum = sum(dot(N3,abs(ev3c)));
    chsStat=chs(x);
    [Kplus,Kminus]=KStest(x);
    factor = 1000*(abs(meanx-0.5)+abs(sdevx-1/sqrt(12)))+100*(max(acArr)-
min(acArr))+100*autoCorrSum+chiSq10+chiSq20/2;
    factor = factor + 10*chsStat + 10*(Kplus + Kminus);
    if bShowResults
        fprintf('Mean = %g\nSdev = %g\n', meanx, sdevx);
        fprintf('Min = %g\nMax = %g\n', min(x), max(x));
        fprintf('Max lags = 100\n');
        fprintf('Auto correlation array\n');
        disp(acArr);
        fprintf('10-Bin Histogram\n');
        disp(N1); disp(ev1);
        fprintf('Chi-Sqr10 = %g\n', chiSq10);
        fprintf('20-Bin Histogram\n');
    end
end

```

```

disp(N2); disp(ev2);
fprintf('Chi-Sqr20 = %g\n', chiSq20);
fprintf('20-Bin Autocorrelation Histogram\n');
disp(N3); disp(ev3);
fprintf('Sum autocorrel product = %g\n', autoCorrSum);
fprintf('Change of sign stat = %g\n', chsStat);
fprintf('K+ = %g and K- = %g\n', Kplus, Kminus);
fprintf('Factor = %g\n', factor);
end
end

function acArr=autocorrArr(xdata,fromLag,toLag)

numLags=toLag-fromLag+1;
acArr=zeros(numLags,1);
j=1;
for i=fromLag:toLag
    acArr(j)=autocor(xdata,i);
    j=j+1;
end
end

function res = autocor(xdata,lag)
%UNTITLED2 Summary of this function goes here
% Detailed explanation goes here
maxElems=length(xdata);
res=corrcoef(xdata(1:maxElems-lag),xdata(lag+1:maxElems));
res=res(1,2);
end

function sumx=chs(x)
% Function CHS calculates the change of sign (between subsequent random
% numbers) moment. The function counts the number of consecutive positive
% and negative changes of sign. The last nested loop calculates the
% statistic returned by this function. This value is the sum of:
%
% sum = sum of difference(count,:) * count / difference(1,:)
%
% Keeping in mind that difference(1,:) is a good value that counts the
% sign flips that happens one neighbor down. The values for
% difference(n,:) for n>1 are not desirable. The smaller, the better. The
% value difference(2,:) is the number of sign flips that occur
% two neighbors down. The value difference(3,:) is the number of sign flips
% that occur three neighbors down, and so on.

n=length(x);
nby2=fix(n/2);
Diff=zeros(nby2,2);
countPos=0;
countNeg=0;
s1=sign(x(2)-x(1));
if s1>0
    bIsPos=true;
    countPos=1;
else
    bIsPos=false;
    countNeg=1;

```

```

end

for i=3:n
    s2=sign(x(i)-x(i-1));
    % was positive and is still positive
    if s2>0 && bIsPos
        countPos=countPos+1;
    % was negative and is now positive
    elseif s2>0 && ~bIsPos
        bIsPos=true;
        countPos=1;
        Diff(countNeg,2)=Diff(countNeg,2)+1;
        countNeg=0;
    % was negative and is still negative
    elseif s2<0 && ~bIsPos
        countNeg=countNeg+1;
    % was positive is and is now negative
    elseif s2<0 && bIsPos
        bIsPos=false;
        countNeg=1;
        Diff(countPos,1)=Diff(countPos,1)+1;
        countPos=0;
    end
end

if s2>0
    if countPos>0, Diff(countPos,1)=Diff(countPos,1)+1; end
else
    if countNeg>0, Diff(countNeg,2)=Diff(countNeg,2)+1; end
end

i=2:nby2;
d=Diff(2:nby2,:);
sumx=0;
for j=1:2
    sumx = sumx + dot(d(:,j),i)/Diff(1,j);
end
end

function [Kplus,Kminus]=KStest(x)
    x=sort(x);
    n=length(x);
    diffMaxPlus=-1e+99;
    diffMaxMinus=-1e+99;
    i=1;
    for xv=0.001:.001:1
        F=xv;
        while x(i)<=xv && i<n
            i=i+1;
        end
        Fn=1;
        if i<n, Fn=(i-1)/n; end
        diff=Fn-F;
        if diff>diffMaxPlus, diffMaxPlus=diff; end
        diff=-diff;
        if diff>diffMaxMinus, diffMaxMinus=diff; end
    end
end

```

```

Kplus=sqrt(n)*diffMaxPlus;
Kminus=sqrt(n)*diffMaxMinus;
end

```

Each PRNG test function has an accompanying function doAll() which runs nine tests for the PRNG test function and calculates the minimum, maximum, mean, standard deviation, and confidence interval for the mean best 30 penalty values. The doAll() function obtains two flavors of these values:

1. Using the mean best 30 penalty factors for each row.
2. Using all of 270 best 30 mean values.

Here is a sample doAll() function:

```

function msg=doAll(runNum,maxElems,countRepeat,bShutDown)

if nargin<4, bShutDown = false; end
if nargin<3, countRepeat=1000; end
if nargin<2, maxElems=100000; end
if nargin<1, runNum=fix(1000000*rand(1,1)); end

sFilename=strcat('commonRng_run', num2str(runNum),'.csv');
fid=fopen(sFilename,'wt');

fprintf(fid,
'Method,Min,Max,Mean,Sdev,CountFailed,Mean30,Sdev30,Factors(30)->\n');

maxiter = 9;
m = 30;
gmean = zeros(maxiter,1);
data = [];
for iter = 1:maxiter
    [minx,maxx,meanx,sdevx,factorArr] =
rngRandoDoralStats(maxElems,countRepeat,runNum);
    fprintf(fid, 'r=frac(997/r),%g,%g,%g,%g,0,','minx,maxx,meanx,sdevx');
    factorArr=sort(factorArr);
    n=length(factorArr);
    if n>m, n=m; end
    data = [data; factorArr(1:n)];
    fprintf(fid,'%g,%g','mean(factorArr(1:n)),std(factorArr(1:n))');
    fprintf(fid,'%g','factorArr(1:n-1)');
    fprintf(fid,'%g\n',factorArr(n));
    gmean(iter) = mean(factorArr(1:n));
    fprintf('Mean30 = %g\n',gmean(iter));
end
fprintf(fid,'\n');
fprintf(fid,'Min,Max,Mean,Sdev,CI Lower, CI Higher\n');
fprintf(fid,'%g,', min(gmean));
fprintf(fid,'%g,', max(gmean));
fprintf(fid,'%g,', mean(gmean));
fprintf(fid,'%g,', std(gmean));
SEM = std(gmean)/sqrt(length(gmean)); % Standard Error
ts = tinv([0.025 0.975],length(gmean)-1); % T-Score
CI1 = mean(gmean) + ts*SEM;

```



```

SEM = std(data)/sqrt(length(data)); % Standard Error
ts = tinv([0.025 0.975],length(data)-1); % T-Score
CI2 = mean(data) + ts*SEM;
fprintf(fid,'%g,%g\n', CI1(1), CI1(2));
fprintf(fid,'%g,', min(data));
fprintf(fid,'%g,', max(data));
fprintf(fid,'%g,', mean(data));
fprintf(fid,'%g,', std(data));
fprintf(fid,'%g,%g\n', CI2(1), CI2(2));
fclose(fid);

msg='Done!';
for i=1:7
    beep;
    pause(3)
end

if bShutDown
    system('shutdown -s')
end
end

```

I ran the calculations by calling the doAll() functions with values of 10,000 and 100,000 for the maxElems and countRepeat parameters, respectively.

The doAll() function writes its results to a .csv file. Here is a partial view of the spreadsheet of such a file:

Method	Min	Max	Mean	Sdev	CountFailed	Mean30	Sdev30	Factors (30) ->	
r=frac(997/z)	107.094	239.021	147.754	11.7865	0	112.35	2.10703	107.094	109.088
r=frac(997/z)	107.679	680.152	147.77	12.0936	0	112.939	1.88342	107.679	108.666
r=frac(997/z)	106.149	213.683	147.767	11.7797	0	112.634	2.14914	106.149	108.895
r=frac(997/z)	110.175	77859.6	148.526	246.035	0	112.9	1.42358	110.175	110.424
r=frac(997/z)	110.068	495.132	147.714	11.8311	0	113.674	1.14592	110.068	111.573
r=frac(997/z)	108.295	384.061	147.742	11.9036	0	113.015	1.80584	108.295	109.607
r=frac(997/z)	106.288	343.971	147.708	11.765	0	112.552	1.84699	106.288	109.624
r=frac(997/z)	105.72	506.141	147.718	11.7897	0	112.431	2.85629	105.72	106.653
r=frac(997/z)	106.269	218.686	147.726	11.8225	0	112.451	1.74473	106.269	109.187
Min	Max	Mean	Sdev	CI Lower	CI Higher				
112.35	113.674	112.772	0.41572	112.452	113.091				
105.72	115.423	112.772	1.94942	112.538	113.005				

Figure 1. A partial view of a sample Excel worksheet showing output results.

Figure 1 shows the summary statistics in the last two rows. The first bottom row shows the statistics of the Mean30 column. The second bottom row shows the statistics of the 9 sets of best 30 mean penalty factors. The values in the column Mean are higher than those in column Mean30 because they are based on all the mean penalty factors and not just the best 30 values. The result in the red font is the main value of interest for our study.

LEGACY CALCULATOR PRNG ALGORITHM

The study uses the following legacy calculator PRNG as the comparison baseline:

$$r_{i+1} = \text{frac}(997 \cdot r_i) \quad (3)$$

The above algorithm appears in the Stat Pac I (page 04-01 of the pac's manual) for the HP-67 programmable calculator. A previous study that I conducted on the same topic showed that equation (3) performed relatively well (and better than other simple PRNGs that HP used in their stat pacs) given its simplicity.

NEW CALCULATOR PRNGS

The PRNGs in this study generally focus on one or more of the following features used to generate random numbers:

- Explore *simple* variations of the baseline PRNG, by replacing the integer 997 with other values such as 1003, 9997, 97, and so on. In all cases, the initial seed MUST HAVE a (significant multidigit) fractional part and preferably a number greater than 0 and less than 1.
- Using multiple occurrences of a random number in the expressions generating new random numbers.
- Using two, three, and even five seeds. The PRNGs then tap into the last two, three, and five random numbers in an arbitrary fashion.
- Using PRNG functions based closely and loosely on the popular Linear Congruential Methods (LCM) used to generate PRNGs for computers.
- The set before last of PRNGs (equations 39 to 52) tweaks the baseline PRNG by testing the addition of different fractional parts to the integer 997. These functions can handle initial seeds (and by mere chance, any random number) that are positive integers. My hope is that the fine-tuning of the baseline PRNG would yield an interesting find.
- The last set of PRNG functions (equations 53 to 55) adds π to the current random number allowing such algorithms to handle initial seeds (and by

mere chance, any random number) that are non-negative integers. Again, my hope with this set of algorithms is that the fine-tuning of the baseline PRNG would yield an interesting find. Stay tuned!

Why so many PRNG algorithms you may ask? Perhaps the answer points us to Nobel laureate Linus Pauling who said, “The best way to have a good idea is to have lots of ideas.”

THE PROPOSED PRNGS

Table 1 shows the list of PRNGs, the first being the the baseline PRNG. The functions in table 1 are sorted by function/sequence number. Table 2 displays the functions sorted by their mean penalty factor values. The tables use colored results to easily classify them. Please read the Notes section that follows this section to learn about the coloring of the results and the PRNG equations that appear in the tables.

<i>Folder Number</i>	<i>PRNG Function</i>	<i>Average Mean30</i>	<i>Eqn Code</i>
0	$r = \text{frac}(997*r)$	112.899	
1	$r = \text{frac}(997/r)$	112.772	
2	$r = \text{frac}(9997*r)$	112.549	
3	$r = \text{frac}(9997/r)$	112.230	
4	$r = \text{frac}(1003*r)$	112.345	
5	$r = \text{frac}(1003/r)$	112.930	
6	$r = \text{frac}(10003*r)$	112.934	
7	$r = \text{frac}(10003/r)$	112.536	
8	$x = \text{frac}(\text{abs}(\pi*(r(i-1)-0.5)))$ $r(i) = \text{frac}(127/x+x)$	112.627	Eqn 8
9	$x = \text{frac}(\text{abs}(\pi/2*(r(i-1)-0.5)))$ $r(i) = \text{frac}(127/x+x)$	112.715	Eqn 9
10	$x = \text{frac}(\text{abs}(\pi*(r(i-1)-r(i-2))))$ $r(i) = \text{frac}(127/x+x)$	112.863	Eqn 10
11	$x = \text{frac}(\text{abs}(\pi/2*(r(i-1)-r(i-2))))$ $r(i) = \text{frac}(127/x+x)$	112.760	Eqn 11
12	$x = \text{frac}(\text{abs}(\sin(\pi/2*(r(i-1)-0.5))))$ $r(i) = \text{frac}(127/x+x)$	112.906	Eqn 12
13	$x = \text{frac}(\text{abs}(\sin(\pi/2*(r(i-1)-r(i-2))))))$ $r(i) = \text{frac}(127/x+x)$	112.586	Eqn 13
14	$r = \text{frac}(97*r)$	113.813	

<i>Folder Number</i>	<i>PRNG Function</i>	<i>Average Mean30</i>	<i>Eqn Code</i>
15	$r = \text{frac}(103 * r)$	113.693	
16	$r = \text{frac}(127 / r + r, 1)$	112.669	
17	$r(i) = \text{frac}(127 / r(i-1) + r(i-2))$	112.802	
18	$r(i) = \text{frac}(127 / r(i-1) + r(i-3))$	112.797	
19	$r = 0.2 + 10 * r + 120 * r^2 + 1200 * r^3$	118.373	Cubic1
20	$r(i) = 10 * r(i-1) + 123 * r(i-2)^2 + 1234 * r(i-3)^3$	112.826	Cube2
21	$r1 = \text{frac}(997 * r)$ $r2 = \text{frac}(1003 * r)$ $r = \text{frac}(127 / r1 + r2)$	112.893	Eqn21
22	$r1 = \text{frac}(997 * r(i-1))$ $r2 = \text{frac}(1003 * r(i-2))$ $r(i) = \text{frac}(127 / r1 + r2)$	112.674	Eqn22
23	$r(3) = \text{rand}$ $r(2) = \text{rand}$ $r(1) = \text{rand}$ for $i = 4 : \text{maxElems}$ $r1 = \text{mod}(997 * r(i-1), 1)$ $r2 = \text{mod}(1003 * r(i-3), 1)$ $r(i) = \text{mod}(127 / r1 + r2, 1)$ end	112.557	Eqn23
24	for $i = 5 : -1 : 1$ $r(i) = \text{rand}$ end for $i = 6 : \text{maxElems}$ if $r(i-3) > 0.5$ $r1 = \text{mod}(997 * r(i-1), 1)$ $r2 = \text{mod}(1003 * r(i-5), 1)$ else $r1 = \text{mod}(997 * r(i-2), 1)$ $r2 = \text{mod}(1003 * r(i-4), 1)$ end $r(i) = \text{mod}(127 / r1 + r2, 1)$ end	112.815	Eqn24
25	for $j = 5 : -1 : 1$ $x(j) = \text{rand}$ end for $j = 6 : \text{maxElems}$	112.886	Eqn25

<i>Folder Number</i>	<i>PRNG Function</i>	<i>Average Mean30</i>	<i>Eqn Code</i>
	<pre> if r(j-3)>0.5 r1 = mod(997*x(j-1),1) r2 = mod(997*x(j-5),1) else r1 = mod(997*x(j-2),1) r2 = mod(997*x(j-4),1) end if r(j-3)<0.5 x(j) = mod(127/r1+r2,1) else x(j) = mod(127/r2+r1,1) end end </pre>		
26	$r = \text{frac}(0.2 + 10 \cdot \sqrt{r} + 120 \cdot r + 1200 \cdot r^{1.5})$	112.769	ShamPoly1
27	$r(i) = \text{frac}(10 \cdot \sqrt{r(i-1)} + 123 \cdot r(i-2) + 1234 \cdot r(i-3)^{1.5})$	112.564	ShamPoly2
28	$r = \text{frac}(999 \cdot r)$	112.862	
29	$r = \text{frac}(999.9 \cdot r)$	112.874	
30	$r = \text{frac}(25214.9039 \cdot r + 0.31779)$	113.233	LCM101
31	$r = \text{frac}(65793 \cdot r + 0.42823)$	112.950	LCM102
32	$r = \text{frac}(168439 \cdot r + 0.8263247)$	113.349	LCM103
33	$r = \text{frac}(1839 \cdot r + 0.8347)$	112.893	LCM104
34	$r(i) = \text{frac}(25214.9039 \cdot r(i-1) + 168439 \cdot r(i-2) + 0.31779)$	112.989	LCM201
35	$m = 2^{24}$ $r = \text{mod}(25214903917 \cdot r + 11, m) / m$	112.687	LCM001
36	$m = 2^{13}$ $r = \text{mod}(65793 \cdot r + 4282663, m) / m$	114.945	LCM002
37	$m = 2^{23}$ $r = \text{mod}(16843009 \cdot r + 826366247, m) / m$	222.700	LCM003
38	$r = \text{frac}(111111 \cdot r)$	113.126	
39	$r = \text{frac}(997.1111 \cdot r)$	112.689	
40	$r = \text{frac}(997.2222 \cdot r)$	112.842	
41	$r = \text{frac}(997.3333 \cdot r)$	112.675	
42	$r = \text{frac}(997.4444 \cdot r)$	112.638	
43	$r = \text{frac}(997.5555 \cdot r)$	112.609	
44	$r = \text{frac}(997.6666 \cdot r)$	112.768	

<i>Folder Number</i>	<i>PRNG Function</i>	<i>Average Mean30</i>	<i>Eqn Code</i>
45	$r = \text{frac}(997.7777 * r)$	112.578	
46	$r = \text{frac}(997.1234 * r)$	112.581	
47	$r = \text{frac}(997.2345 * r)$	113.028	
48	$r = \text{frac}(997.3456 * r)$	112.412	
49	$r = \text{frac}(997.4567 * r)$	112.732	
50	$r = \text{frac}(997.5678 * r)$	112.891	
51	$r = \text{frac}(997.6789 * r)$	112.793	
52	$r = \text{frac}(997.7890 * r)$	112.736	
53	$r = \text{frac}(997 * (\pi + r))$	112.660	
54	$r = \text{frac}(1003 * (\pi + r))$	113.06	
55	$r = \text{frac}(9997 / (\pi + r))$	112.682	

Table 1. The list of equations sorted by equation number.

<i>Folder Number</i>	<i>PRNG Function</i>	<i>Average Mean30</i>	<i>Eqn Code</i>
3	$r = \text{frac}(9997 / r)$	112.230	
4	$r = \text{frac}(1003 * r)$	112.345	
48	$r = \text{frac}(997.3456 * r)$	112.412	
7	$r = \text{frac}(10003 / r)$	112.536	
2	$r = \text{frac}(9997 * r)$	112.549	
23	<pre> r(3)=rand r(2)=rand r(1)=rand for i=4:maxElems r1 = mod(997*r(i-1),1) r2 = mod(1003*r(i-3),1) r(i) = mod(127/r1+r2,1) end </pre>	112.557	Eqn23
27	$r(i) = \text{frac}(10 * \text{sqrt}(r(i-1)) + 123 * r(i-2) + 1234 * r(i-3) ^ 1.5)$	112.564	ShamPoly2
45	$r = \text{frac}(997.7777 * r)$	112.578	
46	$r = \text{frac}(997.1234 * r)$	112.581	
13	<pre> x = frac(abs(sin(pi/2*(r(i-1)-r(i-2)))))) r(i) = frac(127/x+x) </pre>	112.586	Eqn 13
43	$r = \text{frac}(997.5555 * r)$	112.609	
8	<pre> x = frac(abs(pi*(r(i-1)-0.5))) r(i) = frac(127/x+x) </pre>	112.627	Eqn 8

<i>Folder Number</i>	<i>PRNG Function</i>	<i>Average Mean30</i>	<i>Eqn Code</i>
42	$r = \text{frac}(997.4444 * r)$	112.638	
53	$r = \text{frac}(997 * (\pi + r))$	112.660	
16	$r = \text{frac}(127 / r + r, 1)$	112.669	
22	$r1 = \text{frac}(997 * r(i-1))$ $r2 = \text{frac}(1003 * r(i-2))$ $r(i) = \text{frac}(127 / r1 + r2)$	112.674	Eqn22
41	$r = \text{frac}(997.3333 * r)$	112.675	
55	$r = \text{frac}(9997 / (\pi + r))$	112.682	
35	$m = 2^{24}$ $r = \text{mod}(25214903917 * r + 11, m) / m$	112.687	LCM001
39	$r = \text{frac}(997.1111 * r)$	112.689	
9	$x = \text{frac}(\text{abs}(\pi / 2 * (r(i-1) - 0.5)))$ $r(i) = \text{frac}(127 / x + x)$	112.715	Eqn 9
49	$r = \text{frac}(997.4567 * r)$	112.732	
52	$r = \text{frac}(997.7890 * r)$	112.736	
11	$x = \text{frac}(\text{abs}(\pi / 2 * (r(i-1) - r(i-2))))$ $r(i) = \text{frac}(127 / x + x)$	112.760	Eqn 11
44	$r = \text{frac}(997.6666 * r)$	112.768	
26	$r = \text{frac}(0.2 + 10 * \text{sqrt}(r) + 120 * r + 1200 * r^{1.5})$	112.769	ShamPoly1
1	$r = \text{frac}(997 / r)$	112.772	
51	$r = \text{frac}(997.6789 * r)$	112.793	
18	$r(i) = \text{frac}(127 / r(i-1) + r(i-3))$	112.797	
17	$r(i) = \text{frac}(127 / r(i-1) + r(i-2))$	112.802	
24	for i=5:-1:1 r(i)=rand end for i=6:maxElems if r(i-3)>0.5 r1 = mod(997*r(i-1),1) r2 = mod(1003*r(i-5),1) else r1 = mod(997*r(i-2),1) r2 = mod(1003*r(i-4),1) end r(i) = mod(127/r1+r2,1) end	112.815	Eqn24
20	$r(i) = 10 * r(i-1) + 123 * r(i-2)^2 + 1234 * r(i-3)^3$	112.826	Cube2

<i>Folder Number</i>	<i>PRNG Function</i>	<i>Average Mean30</i>	<i>Eqn Code</i>
40	$r = \text{frac}(997.2222 * r)$	112.842	
28	$r = \text{frac}(999 * r)$	112.862	
10	$x = \text{frac}(\text{abs}(\pi * (r(i-1) - r(i-2))))$ $r(i) = \text{frac}(127/x + x)$	112.863	Eqn 10
29	$r = \text{frac}(999.9 * r)$	112.874	
25	for j=5:-1:1 x(j)=rand end for j=6:maxElems if r(j-3)>0.5 r1 = mod(997*x(j-1),1) r2 = mod(997*x(j-5),1) else r1 = mod(997*x(j-2),1) r2 = mod(997*x(j-4),1) end if r(j-3)<0.5 x(j) = mod(127/r1+r2,1) else x(j) = mod(127/r2+r1,1) end end	112.886	Eqn25
50	$r = \text{frac}(997.5678 * r)$	112.891	
21	$r1 = \text{frac}(997 * r)$ $r2 = \text{frac}(1003 * r)$ $r = \text{frac}(127/r1 + r2)$	112.893	Eqn21
33	$r = \text{frac}(1839 * r + 0.8347)$	112.893	LCM104
0	$r = \text{frac}(997 * r)$	112.899	
12	$x = \text{frac}(\text{abs}(\sin(\pi/2 * (r(i-1) - 0.5))))$ $r(i) = \text{frac}(127/x + x)$	112.906	Eqn 12
5	$r = \text{frac}(1003/r)$	112.930	
6	$r = \text{frac}(10003 * r)$	112.934	
31	$r = \text{frac}(65793 * r + 0.42823)$	112.950	LCM102
34	$r(i) = \text{frac}(25214.9039 * r(i-1) + 168439 * r(i-2) + 0.31779)$	112.989	LCM201
47	$r = \text{frac}(997.2345 * r)$	113.028	
54	$r = \text{frac}(1003 * (\pi + r))$	113.06	

<i>Folder Number</i>	<i>PRNG Function</i>	<i>Average Mean30</i>	<i>Eqn Code</i>
38	$r = \text{frac}(111111 * r)$	113.126	
30	$r = \text{frac}(25214.9039 * r + 0.31779)$	113.233	LCM101
32	$r = \text{frac}(168439 * r + 0.8263247)$	113.349	LCM103
15	$r = \text{frac}(103 * r)$	113.693	
14	$r = \text{frac}(97 * r)$	113.813	
36	$m = 2^{13}$ $r = \text{mod}(65793 * r + 4282663, m) / m$	114.945	LCM002
19	$r = 0.2 + 10 * r + 120 * r^2 + 1200 * r^3$	118.373	Cubic1
37	$m = 2^{23}$ $r = \text{mod}(16843009 * r + 826366247, m) / m$	222.700	LCM003

Table 2. The list of equations sorted by the average Mean30 values.

NOTES

Note the following about the tables' information:

- The tables use named equation codes when the corresponding algorithm does not use a simple and short equation.
- The blue-colored results are those for values that are close to that of the baseline PRNG. These values are greater than 112.801 and less than 112.900.
- The red-colored results are those for values that are below the result of the baseline PRNG. These values are less than 112.800.
- The black-colored results correspond to algorithms that did not perform better than the baseline PRNG. Their values are equal to or greater than 112.900.
- If an equation uses the variable r without explicit indices means that the r to the right of the equal sign represents the current random number. The r to the left of the equal signs represents the new random number.
- The equations in the tables assume that the random numbers are never *actually* equal to zero (and neither should be their initial seeds).

CONCLUSIONS

None of the 55 PRNG algorithms did spectacularly better than the baseline PRNG. The following three algorithms that did relatively better than the baseline PRNG are:

$$r = \text{frac}(9997/r) \quad (4)$$

And,

$$r = \text{frac}(1003*r) \quad (5)$$

The third top place goes to the following equation:

$$r = \text{frac}(997.3456*r) \quad (6)$$


The top three equations require seeds that are positive numbers with a significant fractional part. One trick you can use to obtain this kind of seed is to calculate the value of $\log_{10}(x)$ where x is greater than 1 and less than 10.

The PRNGs based on the LCM methods did not do well as I hoped they would. The following PRNG which is similar to the baseline PRN:

$$r = \text{frac}(997/r) \quad (7)$$

Did slightly better than the baseline PRNG— $r=\text{frac}(997*r)$.

Adding π to the current random number did not give a significant improvement. The same can be said about adding various fractional parts to 997. You can still use the algorithms that add π to the current random number to protect against initial seeds (and the very unlikely intermediate random numbers) that are non-negative integers.

 The conclusion of this study, that took a lot of computer time to execute, is that simplicity wins again! I was hoping that more advanced algorithms show marked improvement over the baseline PRNG, but that did not happen. I had hopes that using multiple previous random numbers would add more randomness to the algorithms. In fact, I had conducted another very time-consuming study of similar PRNGs which I totally scrubbed. The weakness of that study is that only one batch of 10000 PRNGs were generated! I became concerned that single mean penalty factor values were not enough to capture the variation in that statistics. That is why I redesigned this study to perform nine runs for each PRNG and perform statistics on the best 270 (= 9 * 30) mean penalty factors.

I would like to apologize in advance if you notice spelling mistakes in this document. Including many lines of listings seems to disable the MS-Word spell checkers that decides on its own to give up on spell-checking regular text.

DOCUMENT HISTORY

<i>Date</i>	<i>Version</i>	<i>Comments</i>
7/1/2020	1.00.00	Initial release.