New Pseudo Random Number Generators: Part 1

By
Namir C. Shammas

INTRODUCTION

Random numbers are usually calculated using pseudo random number generators (PRNGs) to typically support games, simulations, special calculations, and cryptography. Using PRNGs allows you to reproduce sequences of random numbers by supplying the same seed. Over the last four decades I noticed several simple PRNGs used in programmable calculator and BASIC pocket computers. Here are some popular PRNGS that generates uniformly distributed random numbers between 0 and 1:

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

$$r_{i+1} = \text{frac}(147 \cdot r_i) \quad (2)$$

$$r_{i+1} = \text{frac}((\pi + r_i)^5) \quad (3)$$

Where r_i is the current random number and r_{i+1} is the new one. The function frac returns the fractional part of a real value. In the case of the first two PRNGs the initial seed must be a positive real number with a non-zero fractional part. Typically initial seeds are positive integers or real numbers, depending on the algorithm used. The above algorithms are simple enough to work in programmable calculators. They are simpler than the more popular linear congruential methods more commonly used in computer applications. The linear congruential methods have the following general form:

$$x_{i+1} = (a \cdot x_i + c) \bmod m \quad (4)$$

Where all the variables are integers. The uniformly-distributed random number of each iteration is calculated as x_{i+1}/m. Higher bit-integers used to set values for *a* and *m*, give better random numbers with longer periods--before they start repeating the same sequence.

Much research has been done on PRNGs seeking random numbers with very long periods. In the case of the linear congruential methods, the periods depends on the value of the parameter m. The authors of the popular book "Numerical Recipes", Press et al, have criticized the linear congruential methods and recommend against using them. They regard the time and effort computer scientists spent in studying the algorithm's parameters, as a big waste of time! Cryptography and Monte Carlo simulations require random numbers with long periods and very low autocorrelations. Calculators programs are more forgiving of random numbers that are more auto-correlated and have shorted cycles--calculator applications rarely generate enough numbers to see them start repeating.

My goal in this first of three related papers, is to share with you the study I conducted in fine-tuning the type of uniformly-distributed PRNGs seen in the above equations.

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 (which I will simply call *factor*). The lower the factor the better the sequence of random number generated. The values for a calculated factor depend on the count of random numbers generated. This study is based on consistently generating sequences of 100,000 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 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.

I calculate the factor using:

$$\text{Factor} = 1000\,[|\text{mean} - 0.5| + |\text{sdev} - 1/\sqrt{12}|] + 100\,(\text{max\_autoCorrel} - \text{min\_autoCorrel})) + 100 \cdot \text{AutoCorrSum} + \text{ChiSqr10} + \text{ChiSqr20} / 2 \quad (5)$$

Equation (5) calculates the factor by adding 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.

Thus the calculated factor measures the deviation from the expected basic statistics (mean and standard deviation), goodness of distribution for the random numbers, and the level of their autocorrelations. I felt that these three aspects to be an adequate measure for the randomness of the numbers generated. While I certainly respect the diehard test, I feel it is an overkill.

Changing the weights used in equation (5) will change the value of the calculated factor. For the sake of consistency, I have stuck with equation (5) since earlier

studies about random numbers that I conducted in 2013. When I switched from using Excel to Matlab, I adjusted the way to calculate AutoCorrSum. It now takes the actual range of autocorrelations into account and not use a preset range of autocorrelations. Thus the statistic AutoCorrSum has become more in tuned with the actual random numbers generated.

I have calculated factors for the Matlab rand() function and also for different multipliers (like the ones in equations (1) and (2)). Table 1 shows the results. The rows Mean_30 and Sdev_30 show the mean and standard deviation, respectively, for the best 30 results, sorted by factor values. The column titled rand() shows the results for the Matlab rand() function. The seeds for rand() are decimals scaled up into integers, compared to the seeds for the other columns in the table. The scaling is needed because the seeds for the Matlab rand() function must be integers. The table also contains the statistics for the best initial seeds obtained in the range of (0.001, 0.999) and in steps of 0.001. The statistics for the multiplier 977 and for rand() seem to be slightly better than the other values in the table. The results for the arbitrarily chosen multiplier 787 put it at a close third. My target is to go, as much as possible, below the minimum factor values below 29. The new algorithms that I present here goes below this target value.

Parameter	Multiplier						
	rand()	127	145	147	577	787	997
Minimum	29.7636	30.3087	32.6161	32.5187	31.1263	29.4823	29.8569
Maximum_30	35.4215	35.4926	36.3289	36.4672	35.5368	35.7442	35.3705
Mean_30	33.502	33.64681	35.43605	34.79971	33.96834	34.35561	33.3976
Sdev_30	1.50041	1.421245	0.929797	1.130787	1.228065	1.466423	1.40077
First Seed	1	0.001	0.001	0.001	0.001	0.001	0.001
Last Seed	999	0.999	0.999	0.999	0.999	0.999	0.999
Seed Increment	1	0.001	0.001	0.001	0.001	0.001	0.001

*Table 1. Statistics for frac(multiplier*random_number) for various multipliers and using initial seeds that range from .001 to 0.999, in increments of .001.*

Table 2 shows another set of similar results. The table contains the statistics for the best initial seeds obtained in the range of (0.0001, 0.9999) and in steps of 0.0001. The results show that adding a decimal place in the changing the initial seed values yields better factor values. The statistics for the factor values in Table 2 are better than their counterparts in Table 1.

Parameter	Multiplier						
	rand()	127	145	147	577	787	997
Minimum	29.2985	27.3908	28.9075	29.0369	27.5771	29.1636	27.7466
Maximum_30	31.9601	32.4034	32.0317	32.3906	31.9905	31.7282	31.4312
Mean_30	30.8831	31.598	30.9431	31.4740	31.1185	30.6955	30.3002
Sdev_30	0.79728	1.16429	0.925599	0.809699	0.934245	0.847015	1.026946
First Seed	1	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001
Last Seed	9999	0.9999	0.9999	0.9999	0.9999	0.9999	0.9999
Seed Increment	1	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001

*Table 2. Statistics for frac(multiplier*random_number) for various multipliers and using initial seeds that range from .0001 to .9999, in increments of .0001.*

THE NEW PRNGS

I spent several months running multiple computers round the clock and testing all kinds of PRNG algorithms. Whoever said that creating PRNG algorithms is not trivial, sure knew what he was talking about! I usually employed the built-in PRNGs in Matlab and Excel to supply the initial seeds needed to start the PRNG algorithms. This approach is based on the thinking that the studied PRNG algorithms should do well for arbitrary initial seeds for the random numbers.

In the last phase of my study, I came to realize that the seeds for the PRNG algorithms played a bit more significant role than I originally thought. I therefore changed my approach and started to directly supply the initial seed values. The process involved generating factor values for a range of initial seed values. I started with a small seed value and increased it by 0.001, 0.0001, and 0.00001. Smaller values covered a more detailed range of seed values. While this process is repeatable and deterministic, I was surprised that Matlab repeatedly failed to replicate the *transplanted* factors. By transplanted I refer to the seeds that had generated very low factors values, being stored in text files, and read in other PRNG Matlab programs. For some reason which has escaped me, using good seed values did not generate as good factor values!! Sometimes, the factor values were too high and not acceptable. I will explain the solution to this problem later on.

The algorithms I present here are similar and are basically variants of the same core method. The first and best version is:

```
x(1)=initSeed;
for j=2:maxElems;
  if abs(frac(10* x(j-1)))<1e-7
    x(j-1)=frac((x(j-1)+pi)^5+log(j));
  end
  x2=frac(10*x(j-1))
```



```
    x3=frac(10*x2)
    x(j)=frac(11*145*(x(j-1) + 7*145*(x2 + 5*145*x3)))
  end
```

The above pseudo-code generates maxElems random numbers stored in the array x. The variable initSeed represents the initial seed. The function frac returns the fractional part of a real number. The if statement ensures that the last random number has more than one decimal digit. I used a comparison with a small number instead of an equality test with zero, because the function frac(x) (defined as x – fix(x) or as x – floor(x)) has a rounding problem or bug. It took me a while to detect the issue with the function frac. If the tested condition is true, the code assigns a new value to the last random number. This value is the fractional part of the sum of two terms. The first term uses equation (3). The second term is the natural logarithm of the loop counter. The variables x2 and x3 calculate temporary additional random numbers, based on the previous random number. If that number has only one decimal digit, the values in variables x2 and x3 are zero. If the value of x(j) ends with a single decimal, then the numbers generated ill not be random! The magic multiplier here is 145. I have tried a wide range of numbers between 100 and 1000 that end in 5 and in 7. The numbers 145 and 147 gave excellent factor values that reflect very weak autocorrelations between the random numbers generated. This algorithm can easily generate factor values that are slightly below 5, compared to minimum factor of 29 for Matlab's Mersenne-Twister algorithm. It also generates a sizeable sequence of initial seeds that have factor values falling below 10.

If you study the expression that assigns a value to x(j) you will notice that it has a nested subexpression. I originally meant to enter the following statement:

```
x(j)=frac(11*145*x(j-1) + 7*145*x2 + 5*145*x3))
```

But erroneously typed the expression with the nested subexpression. When I realized my error, I developed another version of the PRNG function using the originally-intended expression. To my surprise, the results were by far not as good as the one in the above pseudo-code. This was a true stroke of luck for me!

I decided to apply two variations on the above algorithms:

- Replace 145 with 147.
- Add a small shift value of 2 to the products of 11*145, 7*145, and 5*145.

These above variations give the following general form of the PRNG for a user-defined multiplier (145 and 147 in our case) and shift values (0 and 2 in our case):


```
k1=11*multipler+shift
k2=7*multipler+shift
k3=5*multipler+shift
x(1)=initSeed
for j=2:maxElems
  if abs(frac(10* x(j-1)))<1e-7
    x(j-1)=frac((x(j-1)+pi)^5+log(j));
  end
  x2=frac(10*x(j-1))
  x3=frac(10*x2)
  x(j)=frac(k1*(x(j-1)+k2*(x2+k3*x3)))
end
```

Thus we have four versions of the algorithm:

1. Values of 145 and 0 for the multiplier and shift values, respectively.
2. Values of 147 and 0 for the multiplier and shift values, respectively.
3. Values of 145 and 2 for the multiplier and shift values, respectively.
4. Values of 147 and 2 for the multiplier and shift values, respectively.

Table 3 shows the statistics for the factor values for the above four variations in the multiplier and shift values. In the case of the multiplier 145, using a shift of 2 slightly degrades the factor values. By contrast, in the case of the multiplier 147, using a shift of 2 benefits the factor values.

Parameter	Multiplier and Shift			
	145 and 0	147 and 0	145 and 2	147 and 2
Minimum	4.08354	12.6918	5.49014	9.45088
Maximum_30	5.2639	13.7527	10.8129	11.4277
Mean_30	4.727039	13.357046	7.904233	10.260867
Sdev_30	0.362969	0.2905969	1.904074	0.4482475
First Seed	0.001	0.001	0.001	0.001
Last Seed	0.999	0.999	0.999	0.999
Seed Increment	0.001	0.001	0.001	0.001

Table 3. Statistics for various multiplier/shift combinations and using initial seeds that range from .001 to 0.999, in increments of .001.

Table 4 shows the effect of using the multipliers 145 and 147 with and without a small shift of 2 using smaller seed increment values. Since this table is based on smaller seed increments and covers more seed values it is able to find better results than Table 3. Nevertheless, the comments made about comparing the statistics in the various columns of Table 3, still apply to Table 4.

Parameter	Multiplier and Shift			
	145 and 0	147 and 0	145 and 2	147 and 2
Minimum	3.85348	12.2075	5.31787	8.49994
Maximum_30	4.34976	12.9856	5.84596	9.40546
Mean_30	4.198123	12.76841667	5.681303667	9.202553667
Sdev_30	0.125885	0.198782975	0.136806277	0.250986366
First Seed	0.0091	0.0091	0.0091	0.0091
Last Seed	0.9999	0.9999	0.9999	0.9999
Seed Increment	0.0001	0.0001	0.0001	0.0001

Table 4. Statistics for various multiplier/shift combinations and using initial seeds that range from .0001 to 0.9999, in increments of .0001.

The conclusion that we can draw from tables 3 and 4 is that we can either use the following core algorithm (call it version A1):

```
k1=11*145
k2=7*145
k3=5*145
x(1)=initSeed
for j=2:maxElems
  if abs(frac(10* x(j-1)))<1e-7
    x(j-1)=frac((x(j-1)+pi)^5+log(j));
  end
  x2=frac(10*x(j-1))
  x3=frac(10*x2)
  x(j)=frac(k1*(x(j-1)+k2*(x2+k3*x3)))
end
```

Or the following core algorithm (call it version A2):

```
k1=11*147+2
k2=7*147+2
k3=5*147+2
x(1)=initSeed
for j=2:maxElems
  if abs(frac(10* x(j-1)))<1e-7
    x(j-1)=frac((x(j-1)+pi)^5+log(j));
  end
  x2=frac(10*x(j-1))
  x3=frac(10*x2)
  x(j)=frac(k1*(x(j-1)+k2*(x2+k3*x3)))
end
```

Keeping in mind that algorithm A1 is better than A2. Both algorithms do much better than the PRNG algorithms I discuss in this paper,

Once you have a set of good initial seeds, the ideal approach becomes:

- Selecting the initial seed values that generate low factor values.

- Storing the initial seeds in data files.
- Reading these files in separate client programs.

As I stated earlier, this process, though sound in theory, has not worked for a reason that escapes me! The alternative is to generate sequences of random numbers for a sequence of initial seed values. The first seed value that generates a factor falling below a critical factor value, ends the iterative process and returns the factor value and the large sequence of random numbers. I will present, in the next section, Matlab code that implements this iterative process. If your application needs more random numbers then you can do one of the following:

1. Generate more random numbers using the new algorithms shown above. The initial seed for the new batch of random numbers is the last random number in the previous batch.
2. Rerun the iterative process to locate another sequence of random numbers that has an acceptable low factor value.

While the iterative process requires a bit more calculations, you have control over the range of initial seeds and seed increment values you want to use. Moreover, there is no need to track a data file containing a large set of initial seed values. This level of control is most valuable in obtaining different sets of good random numbers. The alternative is to be somewhat stuck with a set of initial seeds that you read from a data file and then pick, at random, one or more initial seeds.

THE MATLAB CODE

I started out my study using Excel. The advantage was the ability to view all the results on the Excel spreadsheets. However, running VBA in Excel proved to be slow. My curiosity drove me to translate the Excel VBA code into Matlab. Using Matlab, I avoided loops as much as possible and used vectorised expressions. This approach paid very handsomely and I enjoyed faster speed of execution using the same computers.

I finally accelerated the calculations by switching from using a stochastic approach (that uses Matlab's random number generating function to provide the initial seeds) into a deterministic approach by directly supplying the initial seed. The stochastic approach required that I repeat the process in order to get the mean, standard deviation, and other statistics for the factor values. This is justified based on the principle that the initial seeds themselves can be selected at random. The

deterministic approach did away with all of this repetitive work and simplified matter—one initial seed value gives a specific factor value.

Next, let's look at the Matlab function that generate random numbers. You can call this function in your Matlab applications. You can also comment out the fprintf statements in these functions so they will run in a *silent mode*, so to speak.

Here is the source code for the iterative Matlab function rngSimpleGrab which returns an array of random numbers:

```
function [factor,minFactor,lastInitSeed,x] =
rngSimpleGrab(maxElems,multiplier,shift,seedStart,seedIncr,maxFactor,maxit)
% Function generates random number.
%
% Copyright(c) 2015 Namir Clement Shammas
% email: nshammas@.aol.com
%
% INPUT
% =====
% maxElems - the number of random numbers.
% multiplier - the multiplier used to generate the random numbers.
% shift – the shift value.
% seedStart - the starting value for the sequence of initial seed values.
% seedIncr - the increment value for the sequence of initial seed values.
% maxFactor - the critical factor value. We seek random numbers that have a
% factor value below the value of maxFactor.
% maxit - the maximum number of iterations.
%
% OUTPUT
% ======
% factor - the first best factor or -1 if process fails
% minFactor - the smallest factor value encountered. Examine this returned
% value if the process fails, so you can have an idea about using
% maxFactor values in subsequent calls to this function.
% lastInitSeed - the last initial seed used by this function. This value
% is useful in making additional calls to this function to get additional
% arrays of random numbers.
% x - the array of random numbers.
%
%
  fprintf('RNG Special version (grab) with multiplier %g = %g\n',
multiplier);
  clk=clock;
  currtime=clk(4)+clk(5)/100+clk(6)/10000;
  fprintf('Current time is %g\n', currtime);
  initSeed=seedStart;
  minFactor=1e99;
  k1=11*multiplier+shift;
  k2=7*multiplier+shift;
  k3=5*multiplier+shift;
  for iter=1:maxit
    x=zeros(maxElems,1);
    x(1)=frac(initSeed);
    for j=2:maxElems
```

```
      if abs(frac(10*x(j-1)))<1e-7
        x(j-1)=frac((x(j-1)+pi)^5+log(j));
      end
      x2=frac(10*x(j-1));
      x3=frac(10*x2);
      x(j)=frac(k1*(x(j-1)+k2*(x2+k3*x3)));
    end
    factor=calcFactor(x);
    if isnan(factor), factor=1e99; end
    if factor<minFactor, minFactor=factor; end
    if factor<maxFactor, break; end
    initSeed=initSeed+seedIncr;
  end

  % if random generation process fails to meet the factor value criteria
  % then return -1 and an empty array of random numbers
  if factor>=maxFactor
    factor=-1;
    x=[];
    fprintf('Process failed to generate random numbers that meet the critical
factor value\n');
  end
  lastInitSeed=initSeed;
end

function x = frac(x)
  x=x-fix(x);
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 firrst 100 autocorrelation values
  acArr=autocorrArr(x,1,100);
  % calcul 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)));
  factor = 1000*(abs(meanx-0.5)+abs(sdevx-1/sqrt(12)))+100*(max(acArr)-
min(acArr))+100*autoCorrSum+chiSq10+chiSq20/2;

  if bShowResults
    format long
    fprintf('Mean = %g\nSdev = %g\n', meanx, sdevx);
```

```
    fprintf('Min = %g\Max = %g\n', min(x), max(x));
    fprinf('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');
    disp(N2); disp(ev2);
    fprintf('Chi-Sqr20 = %g\n', chiSq20);
    fprintf('20-Bin Autocorrelation Histogram\n');
    disp(N3); disp(ev3);
    fprintf('Sum product = %g\n', autoCorrSum);
    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
```

Remember to test the value of factor for being less than zero. If it is, then the function has failed to generate random numbers whose factor value falls below that of maxFactor. A little bit of testing ahead will help you hone in on the maxFactor values that go along well with the number of random numbers you wish the function to generate. Keep in mind that fewer random numbers yield higher factor values.

Here is a sample session with the above Matlab function:

```
>> [factor,minFactor,lastInitSeed,x] =
rngSimpleGrab(100000,145,0,.001,.001,10,100);
RNG Special version 2 (grab) with multiplier 145 = Current time is 17.3642
>> factor

factor =

    6.9247
```



```
>> lastInitSeed

lastInitSeed =

    0.0040

>> x(1:10)

ans =

    0.0040
    0.3800
    0.2343
    0.2229
    0.4452
    0.9386
    0.5625
    0.8379
    0.9834
    0.2177
```

The above function call requests the generation of 100,000 random numbers using 145 as the magic multiplier and a shift value of 0 (basically no effective shift valur). The function uses 0.001 as both the starting seed and as the seed increment. The function call specifies that the maxFactor be assigned 10 and that there is at most 100 iterations to search for the first appropriate array of random numbers. The first array of random numbers that has a factor below 10, has a factor of 6.92. The parameter lastInitSeed reports that the last initial seed value used was 0.004. Since the initial seed value is 0.001 and the increment in the seed value is also 0.001, we quickly deduce that it took four iterations to give us the results we seek.

Here is a sample code snippet for issuing multiple calls to the above PRNG function:

```
initSeed=0.001;
seedIncr=0.001;
% first call to PRNG function
[factor,minFactor,lastInitSeed,x] =
rngSimpleGrab(100000,145,0,initSeed,seedIncr,10,100);
%
% use random numbers in array x
...
initSeed=lastInitSeed+seedIncr;
% second call to PRNG function
[factor,minFactor,lastInitSeed,x] =
rngSimpleGrab(100000,145,0,initSeed,seedIncr,10,100);
% use random numbers in array x
...
```


A SIDE DISCUSSION

Before I proceed to my conclusion, some of the readers may take a second look at equations (1) and (2) and ask the following questions:

- What if an application supplies an initial value to r_0 by accident to have no fractional part?
- What is the value of r_0 is the perfect inverse of the multiplier (997 or 147)? This would lead to a value of r_1 with no fractional part. Consequently all subsequent random numbers will be zeros!

To make equations (1) and (2) more fool-proof we can add a shift fraction value. Which shift value should we chose? The choice is infinite! The values that come to mind are equal to the multiplier divided by 1000. Thus, adding a shift term (equal to the multiplier divided by 1000) to equations (1) and (2) we get:

$$r_1 = \text{frac}(997 \cdot r_0 + 0.997) \quad (6)$$
$$r_1 = \text{frac}(147 \cdot r_0 + 0.147) \quad (7)$$

Table 5 shows results for the equations with the shift terms using initial seeds that range from 0.001 to 0.999, in increments of 0.001. Comparing tables (1) and (3) shows that equations with the additional shift term do not give significantly better results that the equations without the shift terms.

Parameter	Multiplier					
	127	145	147	577	787	997
Minimum	29.8682	31.0564	31.0564	30.9853	28.6979	28.3841
Maximum_30	35.2288	35.1612	35.1612	34.8198	34.8958	35.4354
Mean_30	33.16524	33.78283	33.78283	33.57075	33.28571	33.82235
Sdev_30	1.480006	1.205951	1.205951	1.241225	1.373499	1.718366
First Seed	0.001	0.001	0.001	0.001	0.001	0.001
Last Seed	0.999	0.999	0.999	0.999	0.999	0.999
Seed Increment	0.001	0.001	0.001	0.001	0.001	0.001

*Table 5. Statistics for frac(multiplier*random_number + multiplier/1000) for various multipliers and using initial seeds that range from .001 to 0.999, in increments of .001.*

Table 6 shows results for equations with the shift term and using initial seeds that range from 0.0001 to 0.9999, in increments of 0.0001. Comparing tables (2) and (6) shows that equations with the additional shift term do not give significantly better results that the equations without the shift terms. Comparing the results of

tables (5) and (6), the results of the latter table are, as expected, better. This improvement is due the more detailed scan of the initial seed values.

Parameter	Multiplier					
	127	145	147	577	787	997
Minimum	27.6922	28.0743	27.3145	27.7758	27.7573	27.6159
Maximum_30	32.0609	31.8837	31.5481	31.4436	31.7078	31.3784
Mean_30	30.91123	30.97038	30.5363	30.2215	30.6938	30.46164
Sdev_30	1.112528	0.847509	0.97083	1.02107	0.93838	0.825438
First Seed	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001
Last Seed	0.9999	0.9999	0.9999	0.9999	0.9999	0.9999
Seed Increment	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001

*Table 6. Statistics for frac(multiplier*random_number + multipler/1000) for various multipliers and using initial seeds that range from .0001 to .9999, in increments of .0001.*

The conclusion to draw is that you can add the shift term if that makes more sense mathematically, but you will not gain much.

What about other simple PRNGs used for calculators? In order to be somewhat complete, I dug into more vintage calculator manuals and found the following equations:

$$r_1 = \text{frac}((997 \cdot r_0 + k^2) / 199) \quad (8)$$
$$r_1 = \text{frac}(9821 \cdot r_0 + 0.211327) \quad (9)$$

Where k is a progressive counter that cycles between 1 and 25, calculated using:

$$k = \text{general\_loop\_counter mod } 26 + 1$$

The general loop counter is the one used to generate the random numbers.

I also tried the following equations:

$$r_1 = \text{frac}((997 \cdot r_0) / 199) \quad (10)$$
$$r_1 = \text{frac}((577 \cdot r_0 + k^2) / 199) \quad (11)$$

My expectations for equation (10) are low. I chose equation 10 somewhat arbitrarily. I felt the outcome to be acceptable but not spectacular.

Table 7 shows results for the equations (8) through (11) using initial seeds that range from 0.001 to 0.999, in increments of 0.001. As expected, equation (10) generated poor sets of random numbers. Equation (11) did not perform well either.

Equations (8) and (9) perform well meeting the minimum level of factors around 30.

Parameter	Multiplier			
	Eqn 7	Eqn 8	Eqn 9	Eqn 10
Minimum	30.5687	30.107	107.877	48.6362
Maximum_30	37.9994	35.5601	122.76	62.4749
Mean_30	36.1859466	33.86539	117.5752333	58.24966
Sdev_30	1.91229979	1.40243048	4.480145762	3.414447744
First Seed	0.001	0.001	0.001	0.001
Last Seed	0.999	0.999	0.999	0.999
Seed Increment	0.001	0.001	0.001	0.001

Table 7. Statistics for equations (7) through (10) using initial seeds that range from .001 to 0.999, in increments of .001.

Table 8 shows results for the equations (8) through (11) using initial seeds that range from 0.0001 to 0.9999, in increments of 0.0001. Again equation (10) gave very disappointing results. By contrast, equation (9) gave somewhat unexpected good (but not excellent or very good) results.

Parameter	Multiplier			
	Eqn 7	Eqn 8	Eqn 9	Eqn 10
Minimum	31.9523	26.3079	98.4476	45.6044
Maximum_30	34.5385	30.9923	111.357	54.8764
Mean_30	33.53256	29.93942	108.714486	51.65601667
Sdev_30	0.74048	1.103937839	2.79935440	2.634965372
First Seed	0.0001	0.0001	0.0001	0.0001
Last Seed	0.9999	0.9999	0.9999	0.9999
Seed Increment	0.0001	0.0001	0.0001	0.0001

Table 8. Statistics for equations (7) through (10) using initial seeds that range from .0001 to 0.9999, in increments of .0001.

The final conclusion draw from looking at tables (5) through (8) is that the new PRNG algorithms perform better than legacy algorithms and their variants.

CONCLUSION

Literature research, extensive hands-on research, and luck have all contributed to the design of a new PRNG algorithm in three variants. The algorithm for the best variant is:

if $|\text{frac}(10 \cdot r_0)| < 1e{-}7$ then

$r_0 = \text{frac}((r_0 + \pi)^5 + \ln(\text{iter\_counter}))$
end
$r_{i1}=\text{frac}(10\cdot r_0)$
$r_{i2}=\text{frac}(10\cdot r_{i1})$
$r_1 =\text{frac}(1595\cdot r_0+1015\ (r_{i1}+725\cdot r_{i2}))$ (A1)

The second best variant is:

if $|\text{frac}(10\cdot r_0)|<1e{-}7$ then
$r_0 = \text{frac}((r_0 + \pi)^5 + \ln(\text{iter\_counter}))$
end
$r_{i1}=\text{frac}(10\cdot r_0)$
$r_{i2}=\text{frac}(10\cdot r_{i1})$
$r_1 =\text{frac}(1619\cdot r_0+1031(r_{i1}+737\cdot r_{i2}))$ (A2)

Depending on how accurate the implementation or your own user-defined version of the frac function, you using either conditions:

- $|\text{frac}(10\cdot r_0)|<1e{-}7$ when the function frac has some rounding and accuracy issues.
- $\text{frac}(10\cdot r_0)=0$ when the function frac is robust.

While the results are aimed at PC applications that can handle a large count of random numbers, you can still use the new algorithms with vintage calculators and new graphing calculators. Of course you will not have the luxury of a screening Matlab function. You should still be fine, since most likely you will need a relatively small population of random number of calculator applications.

What's in a name?

Names are valuable pointers used to refer to various types of objects in the world and universe around us. Since algorithms A1 and A2 have achieved a good goal, I feel they deserve a name. I will call the general algorithm that encompasses A1 and A2 as the *train algorithm*. I am using the word train since the algorithms use three evaluations (which I consider each as a *wagon* in a *train*) to generate a new random number.

What's next?

This paper showed that we zoomed in on two similar algorithms that help generated good quality random numbers. The next paper will cover a thorough

survey of odd multiplier values in the range of 101 to 999. The study will also include shift values of 0 and 2 that are applied to each multiplier value. Thus, the study will handle about 900 versions of algorithms A1 and A2. In addition the second part also presents exploratory surveys of using algorithms A1 and A2 diverse values of the multiplier and shift parameters.

DOCUMENT HISTORY

Date	*Version*	*Comments*
1/30/2015	0.90.00	Initial release.
2/28/2015	1.00.00	Added the if statement in the new PRNG algorithm, updated the Matlab code, and rechecked some results.

