\MNew Pseudo Random Number Generators: Part 2

By
Namir C. Shammas

INTRODUCTION

Part 1 of this study focused on the train PRNG algorithms that generated good quality random numbers. The study in part 1 found that the best version of this algorithm is:

```
if |frac(10·r0)|<1e–7
   r0=frac((r0+pi)^5+log(counter));
end
ri1=frac(10·r0)
ri2=frac(10·ri1)
r1 =frac(1595·r0+1015 (ri1+725·ri2)))                    (A1)
```

The second best variant is:

```
if |frac(10·r0)|<1e–7
   r0=frac((r0+pi)^5+log(counter));
end
ri1=frac(10·r0)
ri2=frac(10·ri1)
r1 =frac(1619·r0+1031(ri1+737·ri2)))                     (A2)
```

Here is the general form of the above algorithms:

```
  k1=11*multipler+shift
  k2=7*multipler+shift
  k3=5*multipler+shift
  x(1)=initSeed
  for i=2 to maxElems
        if abs(frac(10* x(i–1)))<1e–7
          x(i–1)=frac((x(i–1)+pi)^5+log(i))
```



```
        end
        x2=frac(10*x(i–1))
        x3=frac(10*x2)
        x(i)=frac(k1*(x(i-1)+k2*(x2+k3*x3)))                    (A3)
    end
```

In addition, I will present two additional refinements for algorithm A3. These refinements are:

```
    k1=11*multiplier + shift
    k2=7*multiplier + shift
    k3=5*multiplier + shift
    x(1)=initSeed
    for i=2 to maxElems
        if abs(frac(10* x(i–1)))<1e–7
          x(i–1)=frac((x(i–1)+pi)^5+log(i))
        end
        x1=x(i–1)
        while abs(frac(1000*x1))<1e–7
           x1=0.1*x1+0.01
        end
        x2=frac(10*x(i–1))
        x3=frac(10*x2)
        x(i)=frac(k1*(x1+k2*(x2+k3*x3)))                    (A4)
    end
```

And:

```
    k1=11*multiplier + shift
    k2=7*multiplier + shift
    k3=5*multiplier + shift
    x(1)=initSeed
    for i=2 to maxElems
        if abs(frac(10* x(i-1)))<1e–7
          x(i–1)=frac((x(i–1)+pi)^5+log(i))
        end
        x1=x(i–1)
        while abs(frac(1000*x1))<1e–7
           x1=0.1*x1+0.01
```



```
            end
            x2=frac(10*x1)
            x3=frac(10*x2)
            x(i)=frac(k1*(x1+k2*(x2+k3*x3)))                    (A5)
        end
```

Algorithms A4 and A5 are very similar. They both have an additional while loop that ensures the additional temporary variable x1 has enough non-zero decimal places to prevent a numerical stall in generating subsequent random numbers. The statement that assigns the value to variable x2 (which appears in red) is different in algorithms A4 and A5. This is a subtle difference that should generate some variations in the random numbers.

In this part we are going to expand on exploring the values for the multipliers and shift values that will be used in algorithms A3, A4, and A5. I will use three schemes to explore these values:

1. The first scheme is an exploratory and is therefore limited. It calculates the factor values, using algorithms A4 and A5, for the multipliers 145 and 147, and for the shift values of 0 and 2. This scheme will give us data that allows a quick comparison with similar results given by algorithm A3 in part 1.
2. The second scheme performs a more detailed exploration of algorithms A3, A4, and A5 using a wider range of multipliers. The scheme calculates the factor statistics for multipliers in the range of 100 to 1000 in steps of 10, with the following value patterns:
    a. Adding 1, 3, 5, and 7 to each selected multiplier. Thus, the enumerated list of multipliers is 101, 103, 105, 107, …, 991, 993, 995, and 997.
    b. Each enumerated multiplier has shift values of 0 and 2.
    c. The initial seed starts at 0.00135711 and moves up in increments of 0.001. This sequence of values ensures that the initial seeds have plenty of decimal places.
3. The third scheme explores the factors for random multipliers (with odd integer parts) and random shift values. The multipliers range from 100 to 1999. The shift values include zero along with the ranges (–1, –10) and (1, 10). This scheme of calculation is essentially a gamble that helps us explore whether random multipliers and shift values generate a significant number of good factors.

I felt that adding the two decision-making constructs to algorithm A4 and A5 protects the random number generated from poor initial seed values. They also protect from previous random number generated with one or two decimal places, even though the probability of such numbers is extremely low.

We will see what the proposed schemes will produce with algorithms A3, A4, and A5. This survey will also tell about the contribution of the algorithm's equations and the contribution of the combination of multiplier and shift values to the success of the train PRNG algorithm. My hope is to discover that it is the algorithm's equations that have the major influence. If this hypothesis is correct, then we should see many multiplier/shift values in either scheme 1 and/or 2 that yield good random number sequences. If my hypothesis is not accepted, then algorithms A1 and A2 are unique.

THE RESULTS OF SCHEME 1

This scheme compares algorithms A4 and A5 with A3 for the multipliers 145 and 147, and also for the shift values of 0 and 2. Table 1 below is a copy of Table 3 that you saw in part 1 of the article. It contains the sought results applied to algorithm A3.

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 1. Statistics for various multiplier/shift combinations and using initial seeds that range from .001 to 0.999, in increments of .001 (Algorithm A3).

Table 2 shows comparable results for algorithm A4.

Parameter	Multiplier and Shift			
	145 and 0	147 and 0	145 and 2	147 and 2
Minimum	3.75116	12.6993	5.68713	9.25253
Maximum_30	5.10216	13.8469	10.1069	10.4772
Mean_30	4.667854333	13.44649	7.263524333	10.02553233
Sdev_30	0.32928692	0.286336323	1.649109961	0.348349407

Parameter	Multiplier and Shift			
	145 and 0	147 and 0	145 and 2	147 and 2
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 2. Statistics for various multiplier/shift combinations and using initial seeds that range from .001 to 0.999, in increments of .001 (Algorithm A4).

Comparing the results in Tables 1 and 2 we see that:

- The factors for multiplier 145 and shift 0 are better in Table 2. The same comment can be made for multiplier 147 and shift 2.
- There is a slight degeneration for the factors associated with multiplier 145 and shift 2 in Table 2. The same comment can be made for multiplier 147 and shift 0.

Table 3 shows comparative results for algorithm A5.

Parameter	Multiplier and Shift			
	145 and 0	147 and 0	145 and 2	147 and 2
Minimum	3.80687	12.6625	5.55847	8.84384
Maximum_30	5.26802	13.6893	10.3924	10.6712
Mean_30	4.819453333	13.36397667	7.027759667	9.870448333
Sdev_30	0.357998198	0.313678641	1.657806463	0.440525668
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 (Algorithm A5).

Comparing the results in Tables 1 and 3 we see that only the multiplier 145 and shift 2 give better results in Table 1 than in Table 3. The statistics for the other multipliers and shift values are better in Table 3.

Comparing the results in Tables 2 and 3 we see that only the multiplier 145 and shift 0 give better results in Table 2 than in Table 3. The statistics for the other multipliers and shift values are better in Table 3.

Keep in mind that the differences between corresponding statistics in the above tables are not drastically different. The results in Tables 2 and 3 are encouraging regarding the potentials of algorithms A4 and A5. Algorithm A5 seems to be slightly ahead of algorithms A3 and A4.

THE RESULTS FOR SCHEME 2

This section looks at the results of applying scheme 2 to algorithms A3, A4, and A5.

Algorithm A3

This next subsection present the results and Matlab code related to algorithm A3.

The Results of Algorithm A3

Table 4 shows the factors for algorithm A3 that are less than 5. Decreasing the increment in shift values would produce more factors in that same range. The table has three entries, out of eleven, with shift values of 2. The multipliers in Table 4 range from 113 to 533. This range is based on the factors sampled in this scheme.

Factor1	InitSeed1	Multip1	Shift1
4.26333	0.724357	145	0
4.55489	0.094357	201	0
4.59443	0.888357	353	0
4.72493	0.538357	261	2
4.72918	0.418357	273	0
4.74325	0.991357	533	0
4.79468	0.304357	453	0
4.79798	0.303357	273	2
4.92583	0.420357	253	0
4.98322	0.987357	315	2
4.99702	0.841357	113	0

Table 4. Results for algorithm A3 showing factor values in the range (0, 5).

Table 5 shows the factors for algorithm A3 that are in the range of (4, 5). Decreasing the increment in shift values would produce more factors in that same range. The table has 17 (out of 33) entries with shift values of 2. The multipliers in Table 5 range from 107 to 551. This range is based on the factors sampled in this scheme.

Factor1	InitSeed1	Multip1	Shift1
5.01079	0.485357	167	2
5.01333	0.015357	351	0
5.07161	0.953357	463	0
5.10548	0.112357	251	2
5.14921	0.215357	473	2
5.20462	0.930357	225	0

Factor1	InitSeed1	Multip1	Shift1
5.21013	0.499357	327	2
5.29201	0.796357	133	0
5.30827	0.797357	253	2
5.37103	0.608357	131	0
5.45269	0.095357	415	0
5.46604	0.889357	425	2
5.47284	0.404357	107	2
5.48734	0.491357	191	0
5.55406	0.415357	263	0
5.57514	0.465357	145	2
5.64224	0.547357	333	2
5.64842	0.650357	391	0
5.70464	0.699357	477	0
5.72252	0.625357	551	0
5.7824	0.672357	283	0
5.78384	0.824357	161	2
5.84541	0.274357	521	2
5.85243	0.444357	351	2
5.87229	0.427357	285	0
5.90282	0.569357	131	2
5.91345	0.989357	381	0
5.91832	0.802357	153	2
5.93678	0.624357	297	0
5.97883	0.042357	335	2
5.98493	0.108357	241	2
5.99238	0.468357	377	2
5.99465	0.692357	163	0

Table 5. Results for algorithm A3 showing factor values in the range (5, 6).

Table 6 shows the count for the factors in ranges of 5 and 10. Figure 1 shows the histogram for the data in Table 6. Notice that the factors in the range (5, 10) peak in the histogram. The histogram is skewed to the left, peaking quickly and then slowly decreasing.

Bin	Count	Bin	Count
0	0	80	22
5	11	90	4
10	195	100	2
15	106	110	2

Bin	Count	Bin	Count
20	77	120	1
25	50	130	0
30	33	140	3
40	66	150	4
50	34	160	1
60	21	170	0
70	31	180	0

Table 6. Results for algorithm A3 showing the count for factor values in different ranges.

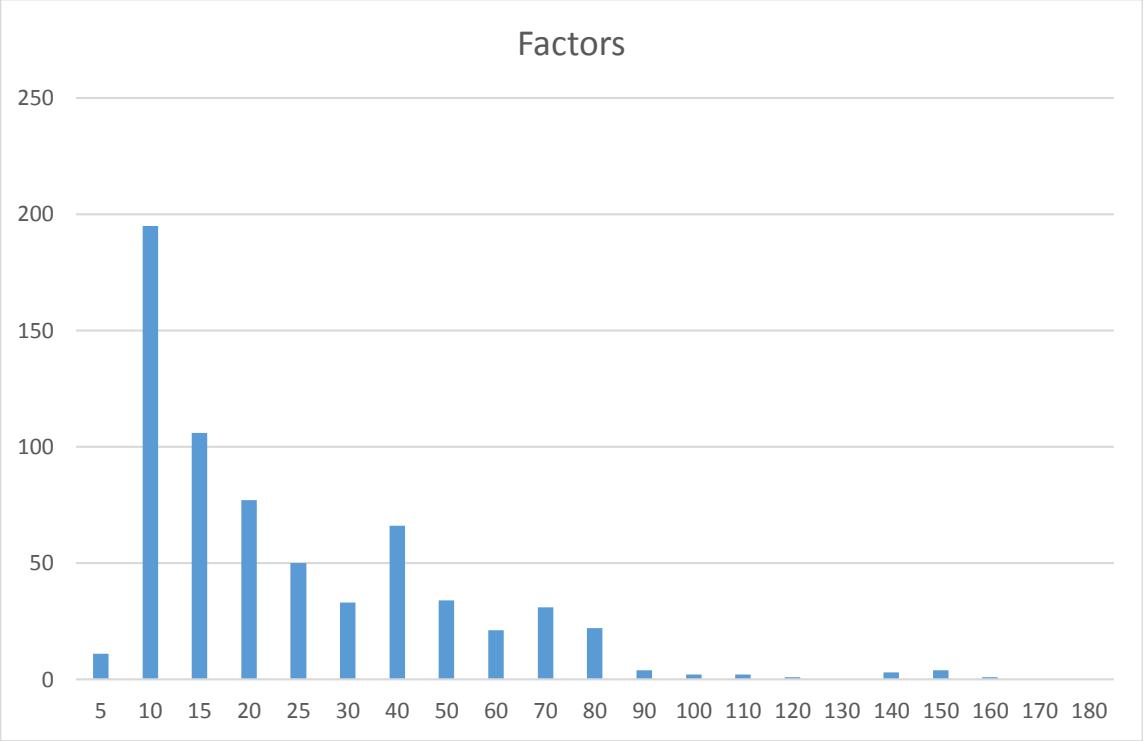

Figure 1. Results for algorithm A3 showing the count for factor values in different ranges.

Table 7 shows the count for the initial seeds in increments of 0.05. Figure 2 shows the histogram for the data in Table 7. The histogram shows a roughly uniform distribution for the initial seeds.

Bin	Count	Bin	Count
0.00	0	0.55	39
0.05	42	0.60	39
0.10	43	0.65	37
0.15	37	0.70	29
0.20	38	0.75	26
0.25	40	0.80	34

Bin	Count	Bin	Count
0.30	42	0.85	33
0.35	32	0.90	28
0.40	35	0.95	34
0.45	36	1.00	39
0.50	45		

Table 7. Results for algorithm A3 showing the count for initial seeds in different ranges.

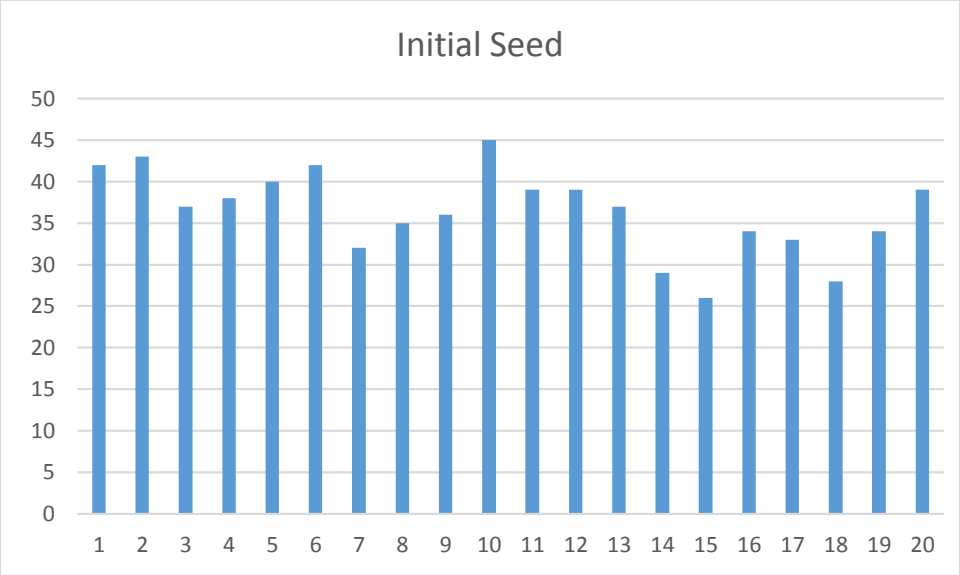

Figure 2. Results for algorithm A3 showing the count for initial seed values in different ranges.

Table 8 shows the results of regression curve fittings between the factors and the multipliers and between the factors and the initial seed values. All four curves have very low coefficient of determinations. Figures 3 shows the data for the ln(factor) vs ln(multiplier). Figures 4 shows the data for the ln(factor) vs ln(initial seed). Both plots are quite scattered showing the lack of a reliable correlation between the factors and the multipliers and between the factors and the initial seed values.

ln(factor) vs ln(multiplier)		
	Slope	0.045624
	Intercept	1.750651
	R^2	0.009672
factor vs multiplier	Slope	0.933944
	Intercept	–247.662
	R^2	0.00393
ln(factor) vs ln(initial seed)		

	Slope	–0.01353
	Intercept	1.997758
	R^2	0.002627
factor vs initial seed	Slope	–1241.32
	Intercept	864.2488
	R^2	0.008449

Table 8. Results for algorithm A3 showing the linear and log-log regression coefficients between the factors and the multipliers and between the factors and the initial seeds.

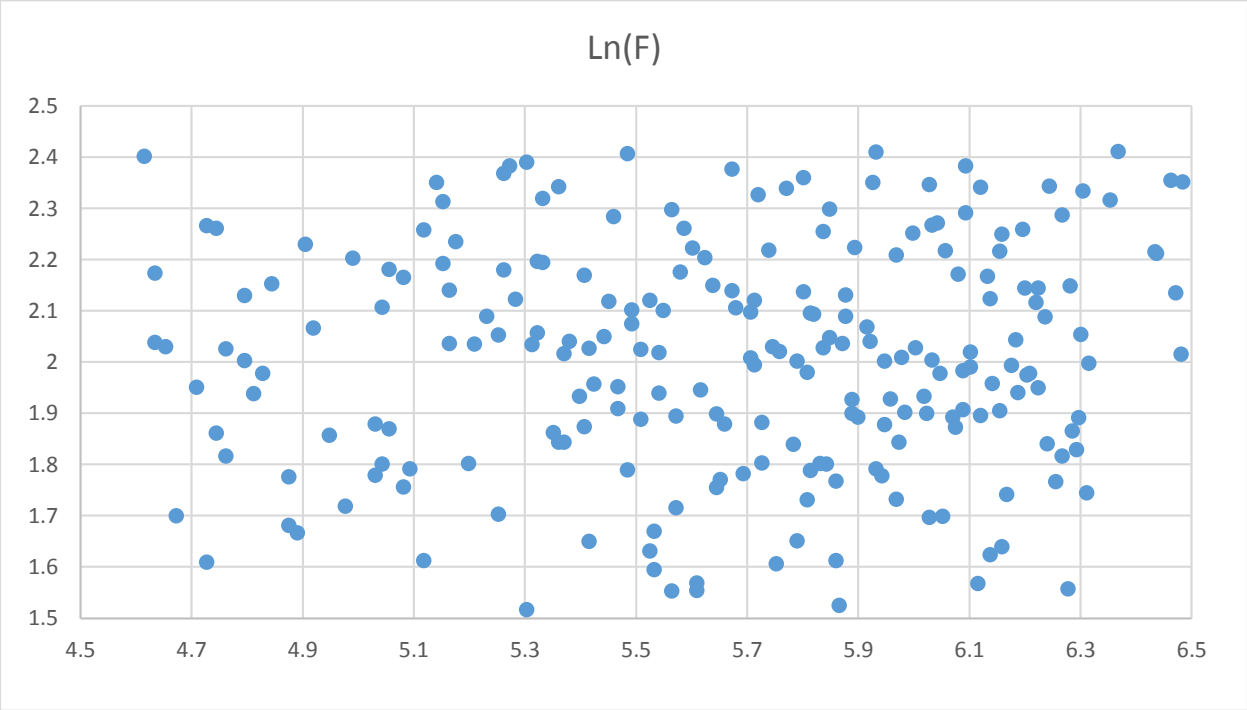

Figure 3. Results for algorithm A3 showing the values for ln(factor) vs ln(multiplier).

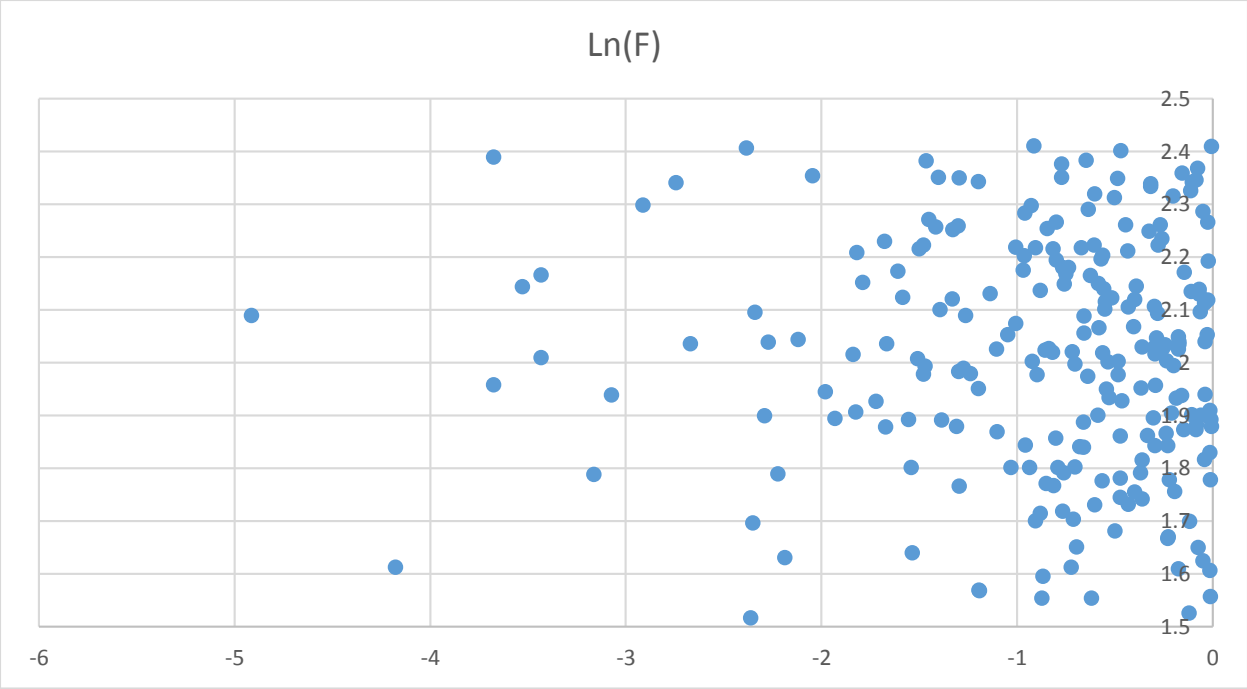

Figure 4. Results for algorithm A3 showing the values for ln(factor) vs ln(initial seed).

Performing a power fit for the following model:

Ln(factor) = a + b ln(InitSeed) + c ln(Multiplier)

Gives the following results using the Excel regression tool:

SUMMARY OUTPUT

Regression Statistics	
Multiple R	0.600868
R Square	0.361043
Adjusted R Square	0.35928
Standard Error	1.060122
Observations	728

ANOVA

	df	*SS*	*MS*	*F*	*Significance F*
Regression	2	460.4015	230.2008	204.8306	3.04E-71
Residual	725	814.7982	1.12386		

Total	727	1275.2

	Coefficients	*Standard Error*	*t Stat*	*P-value*	*Lower 95%*	*Upper 95%*
Intercept	-3.97275	0.412067	-9.64103	8.88E-21	-4.78173	-3.16376
Ln(InitSeed)	-0.38096	0.039891	-9.5501	1.94E-20	-0.45927	-0.30264
Ln(Multip)	1.094468	0.067115	16.3074	3.65E-51	0.962706	1.226231

The above results show that the power relation between the variables, albeit it a weak one, is:

factor = 0.018821644 * InitSeed^0.683206039 * multiplier^2.987594154

Which is roughly close to:

factor = 0.02 * InitSeed^0.5 * multiplier^3

Which hints at a trend that increases the factor values with increasing initial seed values and multiplier values. This explains why lower factor values are associated with lower multiplier values.

The Matlab Code for Algorithm A3

The function rngSimpleVerA3 generates the random numbers and returns the factor value:

```
function factor = rngSimpleVerA3(maxElems,multiplier,shift,initSeed)
%UNTITLED2 Summary of this function goes here
%   Detailed explanation goes here
  fprintf('RNG version A3 with multiplier %g and shift %g and initial
seed = %g\n', multiplier,shift,initSeed);
  x=zeros(maxElems,1);
  k1=11*multiplier+shift;
  k2=7*multiplier+shift;
  k3=5*multiplier+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
  factor=calcFactor(x);
```



```
  if isnan(factor), factor=1e99; end
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);
  % calculte 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
    fprintf('Mean = %g\nSdev = %g\n', meanx, sdevx);
    fprintf('Min = %g\nMax = %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 autocorrel 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
```

The function runA4 calls function rngSimpleVerA3 repeatedly to obtain the following results:

- The 30 minimum factors, for a given multiplier and shift values, and for a range of initial seed values.
- The mean and standard deviation of the best 30 factors.
- The exit code which tells the caller if the function terminated early due to minimum factors that were high.

The parameter multiplier and shift pass the arguments for the multiplier and shift values, respectively. The parameter start passes values for the starting initial seed values. The parameter incr passes the values for the increment in seed values.

```
function [minFactorArr,meanx,sdevx,exitCode]=runA3(multiplier, shift,
start, incr)
% version 2
clc

MaxRandNums=100000;
minFactor=1e99;
minInitSeed=1e99;
maxMinFactArrCount=30;
minFactorArr=zeros(maxMinFactArrCount,4);
% initialze the first column to have very large values
minFactorArr(:,1)=1e10;
initSeed=start;
iter=0;
maxBailOutIters1=150;
criticalFactor1=1000000; % to bail out for VERY bad process
```



```
maxBailOutIters2=10*maxMinFactArrCount+50;
criticalFactor2=100; % to bail out
exitCode=0;

while initSeed<1

  x = rngSimpleVerA3(MaxRandNums,multiplier,shift,initSeed);
  fprintf('Factor = %g with init seed = %g\n', x, initSeed);

  % Add new value to the end of the array if it is less than the last
  % array element. For the first maxMinFactArrCount iterations the
  % calculate factor values will replace the very large values used
  % to intiialize the array minFactorArr. Calling the function
sortrows()
  % helps the new value to bubble up and be replaced by any old large
  % array-intializing values. Beyond maxMinFactArrCount iterations the
  % calculator factors and their related parameters will compete fir
  % best places with the help of function sortrows().
  if x<minFactorArr(maxMinFactArrCount,1)
    minFactorArr(maxMinFactArrCount,1)=x;
    minFactorArr(maxMinFactArrCount,2)=initSeed;
    minFactorArr(maxMinFactArrCount,3)=multiplier;
    minFactorArr(maxMinFactArrCount,4)=shift;
  end

  lastBestFact= minFactorArr(1,1);
  minFactorArr=sortrows(minFactorArr,1);
  if lastBestFact~=minFactorArr(1,1), fprintf('**** '); end
  fprintf('Min factor = %g at initseed = %g\n\n', minFactorArr(1,1),
minFactorArr(1,2));
  initSeed=initSeed+incr;

  % test if the process is not doing well.
  iter=iter+1;

  % First test if process is DOING HORRIBLY!!!
  if iter>maxBailOutIters1 && minFactorArr(1,1)>criticalFactor1
    exitCode=2;
    break; % bail out!!
  end

  % Second test if process is not doing well
  if iter>maxBailOutIters2 && minFactorArr(1,1)>criticalFactor2
    exitCode=1;
    break; % bail out!!
  end

end % main while loop

meanx=mean(minFactorArr(:,1));
sdevx=std(minFactorArr(:,1));
```



```
end
```

The function goAllA3 calls function runA4 for a range of multipliers. This function enables you to obtain the results of a wide range of odd multiplier (from 100 to 990) in batches. You can supply the parameters mult1 and mult2 with the arguments of 101 and 990 but the function will most likely execute on your machine for several days! Thus the parameters mult1 and mult2 permit you to chop this big task into small pieces and possibly run the code on several machines. The seqNum parameter is an integer (1, 2, 3, and so on) that is used in building the name of the output file. An argument of 1 for this parameter yields and output file of resAllA3_seq1.csv. An argument of 2 for this parameter yields and output file of resAllA3_seq2.csv, and so on.

```
function msg=goAllA3(mult1,mult2,seqNum)
% version A3

if ~exist('mult1','var') || isempty(mult1)
  mult1=100;
end
if ~exist('mult2','var') || isempty(mult2)
  mult1=990;
end

% delete any existing temporary .tmp files
delete('*.tmp');

primaryFilename='resAllA3_seq';

if ~exist('seqNum','var') || isempty(seqNum)
  seqNum=1;
  resfile=strcat(primaryFilename, num2str(seqNum), '.csv');
  while exist(resfile,'file')
    seqNum=seqNum+1;
    resfile=strcat(primaryFilename, num2str(seqNum), '.csv');
  end
end

initSeed=0.00135711;
stepSeed=0.001;
% create a temporary filename
resfile=strcat(primaryFilename, num2str(seqNum), '.tmp');
fid=fopen(resfile,'wt');
% write the column headings to the file
for i=1:30
  fprintf(fid, 'Factor%d,InitSeed%d,Multp%d,Shift%d', i, i, i, i);
end
fprintf(fid, 'Mean30,Sdev30,ExitCode\n');
count=0;
```



```
maxCount=100;
multAddArr=[1 3 5 7];
for iMult=mult1:10:mult2;
  for iAdd=1:length(multAddArr);
    multiplier=iMult+multAddArr(iAdd);
    for shift=0:2:2

[minFactorArr,meanx,sdevx,exitCode]=runA3(multiplier,shift,initSeed,st
epSeed);
        [nrows,ncols]=size(minFactorArr);

        count=count+1;
        if count>=maxCount
          count=1;
          fclose(fid);
          fid=fopen(resfile,'at');
        end

        for i=1:nrows
          for j=1:ncols
            fprintf(fid,'%g,',minFactorArr(i,j));
          end
        end
        fprintf(fid,'%g,%g,',meanx,sdevx);
        if exitCode==2
          fprintf(fid,'999\n');
        elseif exitCode==1
          fprintf(fid,'555\n');
        else
          fprintf(fid,'0\n');
        end

      end % for shift
  end % iAdd
end % for iMult

fclose(fid);

% now rename the temporary file into the permanent filename
resfileFinal=strcat(primaryFilename, num2str(seqNum), '.csv');
% now store the name of the backup file
resfileBak=strcat(resfileFinal,'.bak');
movefile(resfile,resfileFinal);
% make a backup too!
copyfile(resfileFinal,resfileBak);
msg='Done!';

end
```


The function rngSimpleGrabA3 returns the array of random numbers you request along with the factor value, minimum factor value, and last initial seed value. This function iterates generating maxElems random number for a specified multiplier and shift values. The parameters seedStart and seedIncr specify the starting initial seed and its increment value, respectively. The parameter maxFactor designates the maximum factor you are willing to accept. The parameter maxit designates the maximum number of iterations.

```
function [factor,minFactor,lastInitSeed,x] =
rngSimpleGrabA3(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. If the absolute
% value of seedStart is equal to or greater than 1, the function uses
the Matlab rand()
% to generate a startSeed value.
% seedIncr - the increment value for the sequence of initial seed
values. If the absolute
% value of seedIncr is equal to or greater than 1, the function uses
the Matlab rand()
% and divide that number by 10 to generate a value for seed increment
in each iteration.
% 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);
  if abs(seedStart)<1
    initSeed=seedStart;
  else
    rng('shuffle','twister');
    initSeed=rand(1,1);
  end
  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=1e9; end
    if factor<minFactor, minFactor=factor; end
    if factor<maxFactor, break; end
    if abs(seedIncr)<1
      seedIncrTemp=seedIncr;
    else
      seedIncrTemp=rand(1,1)/10;
    end
    initSeed=initSeed+seedIncrTemp;
    if initSeed>1, initSeed=initSeed-1; end
    if initSeed<0, initSeed=initSeed+1; end
  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);
  % 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)));
  factor = 1000*(abs(meanx-0.5)+abs(sdevx-
1/sqrt(12)))+100*(max(acArr)-
min(acArr))+100*autoCorrSum+chiSq10+chiSq20/2;

  if bShowResults
    fprintf('Mean = %g\nSdev = %g\n', meanx, sdevx);
    fprintf('Min = %g\nMax = %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 autocorrel 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)
%
maxElems=length(xdata);
res=corrcoef(xdata(1:maxElems-lag),xdata(lag+1:maxElems));
res=res(1,2);
end
```

You can download the above Matlab files (and possibly more updated versions of them) and all of the other Matlab files from my website. Visit the following page and locate the bullet item with the title *New random number generator algorithms*:

http://www.namirshammas.com/NEW/mainNEW.htm

Algorithms A4 and A5

Surprisingly the calculations for algorithms A4 and A5 gave the same results as those of algorithm A3. This duplication is a result of supplying the random number generator function with seed values that are equal to 0.00135711 + 0.01*n. Since the initial seed values have eight decimal places, the decision-making constructs in the RNG loops were not invoked! This explains why the Matlab code for algorithms A4 and A5 gave the same results as those of algorithm A3. If we look back at the results of scheme 1, where the initial seeds are integer multiples of 0.001, we do see slight differences between the three algorithms.

Rather than redoing the time consuming calculations for the three algorithms using initial seeds that are multiples of 0.001, I have the following Matlab statement to use as a workaround that allows you to supply initial seeds that are multiples of 0.001 and still get the same results when using either algorithm. The programming trick is to take the first three decimals of the initial seed and append the digits 35711. Here is the Matlab decision-making construct that you can insert in the PRNG functions to ensure uniform results among the three algorithms:

```
if abs(frac(1000*initSeed))<1e-7
  initSeed=(frac(1000*frac(initSeed)) + 0.35711)/1000;
end
```


The if statement determines whether the initial seed has less than four decimals. If so, the code recalculates the initial seed to include more decimal places. The expression 1000*frac(initSeed) extracts the first three decimal places of the initial seeds as integers. The plus operator adds the decimal 0.35711 to that integer. The division by 1000 ensures that the expression returns a normalized seed value that is less than 1. I inserted the above decision-making construct in the Matlab files that can use them and posted these updated files on my web site.

Conclusion from Scheme 2 Calculations

The conclusion draw from scheme 2 calculations is that the initial seed values play an important role in generating good sequences of random numbers. To be more specific, supplying initial seeds with 7, 8, or more decimal places ensures the success of the process. Under these conditions, you can use algorithm A3 which is faster (due to having less code in the RNG loop) than the other two algorithms. The algorithms A4 and A5 have the code for making the process more fool proof. They provide a proverbial insurance policy against the degradation of pseudo random numbers generated. The bottom line is that you have a choice between the three algorithms. Pick what suits your computational needs.

THE RESULTS FOR SCHEME 3

The Results for Algorithm A3

Table 9 shows the first sample results obtained by applying scheme 3 to the A3 algorithm. The table shows the results for factors less than 20. Notice that all the multiplier and shift values in that table are integers! The table does show negative shift values. The table shows single entries for factors between 4 and 5, between 5 and 6, and between 6 and 7.

Factor1	InitSeed1	Multip1	Shift1
4.54273	0.173	453	0
5.83457	0.774	421	6
6.07763	0.504	341	8
8.55114	0.184	311	0
9.58164	0.268	341	-8
9.63086	0.379	491	0
9.89578	0.791	421	-6
10.9366	0.811	409	0
11.5956	0.413	409	2

Factor1	InitSeed1	Multip1	Shift1
12.2119	0.723	421	0
12.6918	0.423	147	0
12.7952	0.586	125	0
14.9647	0.457	341	0
15.0945	0.616	687	0
17.2147	0.282	669	0
18.2696	0.599	449	0

Table 9. The first same results for algorithm A3 using scheme 3.

Table 10 shows the second sample results obtained by applying scheme 3 to the A3 algorithm. The table shows the results for factors less than 20. Again, notice that all the multiplier and shift values in that table are integers! The table also shows negative shift values. The table shows single entries for factors between 5 and 6 and between 6 and 7.

Factor1	InitSeed1	Multip1	Shift1
5.14629	0.877	463	0
6.02807	0.722	345	0
7.28058	0.152	217	2
7.59552	0.968	217	-2
7.85173	0.031	463	-8
9.41398	0.992	463	8
10.0039	0.686	409	4
10.9354	0.065	241	0
10.9366	0.811	409	0
12.9065	0.539	409	-4
14.1771	0.613	317	0
15.2431	0.812	217	0

Table 10. The second same results for algorithm A3 using scheme 3.

The Results for Algorithm A4

Table 11 shows the first sample results obtained by applying scheme 3 to the A4 algorithm. The table shows the results for factors less than 20. Notice that all the multiplier and shift values in that table are integers! The table does show negative shift values. The table shows single entries for factors between 5 and 6, and between 6 and 7. The table shows two entries for factors between 7 and 8.

Factor1	InitSeed1	Multip1	Shift1
5.41091	0.298	141	-6

Factor1	InitSeed1	Multip1	Shift1
6.37862	0.866	141	6
7.28302	0.507	375	6
7.62717	0.548	239	-4
8.37679	0.626	291	0
8.4842	0.709	289	0
8.55816	0.314	239	4
8.60262	0.928	257	0
9.55009	0.068	239	0
12.1659	0.352	107	0
12.5574	0.703	625	4
12.9834	0.366	525	0
13.2302	0.656	135	0
14.2601	0.648	375	-6
15.116	0.801	375	0
15.4994	0.034	141	0

Table 11. The first same results for algorithm A4 using scheme 3.

Table 12 shows the second sample results obtained by applying scheme 3 to the A4 algorithm. The table shows the results for factors less than 20. Notice that all the multiplier and shift values in that table are integers! The table also shows negative shift values. The table shows two entries for factors between 5 and 6 and one entry between 6 and 7. The table shows two entries for factors between 7 and 8.

Factor1	InitSeed1	Multip1	Shift1
5.85702	0.156	509	8
5.89594	0.451	167	4
6.66455	0.074	167	-4
7.47448	0.789	509	0
7.51505	0.608	653	0
8.36518	0.219	251	0
8.55916	0.530	509	-8
8.65133	0.845	461	0
8.72409	0.345	427	0
8.88876	0.828	157	0
9.92651	0.140	461	4
9.96368	0.224	167	0
11.0844	0.435	349	0
13.0555	0.985	519	0

Factor1	InitSeed1	Multip1	Shift1
19.1079	0.289	461	-4

Table 12. The second same results for algorithm A4 using scheme 3.

The Results for Algorithm A5

Table 13 shows the first sample results obtained by applying scheme 3 to the A5 algorithm. The table shows the results for factors less than 20. Notice that all the multiplier and shift values in that table are integers! The table does show negative shift values. The table shows single entries for factors between 4 and 5. The table shows multiple entries for factors between 5 and 6, and also between 6 and 7.

Factor1	InitSeed1	Multip1	Shift1
4.77390	0.563	201	0
5.25344	0.549	437	-4
5.37086	0.202	253	0
5.51552	0.896	191	0
6.16349	0.843	181	-6
6.22406	0.089	537	2
6.93895	0.464	253	6
9.06016	0.383	427	0
9.09334	0.932	363	0
10.2231	0.442	451	8
10.5937	0.173	437	4
11.5542	0.185	437	0
12.5604	0.044	253	-6
14.2622	0.013	181	6
14.3023	0.250	537	0
14.9033	0.298	537	-2
15.2807	0.853	523	0
16.6246	0.113	181	0

Table 13. The first same results for algorithm A5 using scheme 3.

Table 14 shows the second sample results obtained by applying scheme 3 to the A5 algorithm. The table shows the results for factors less than 20. Notice that all the multiplier and shift values in that table are integers! The table also shows negative shift values. The table shows three entries for factors between 5 and 6, one entry between 6 and 7, and two entries between 7 and 8 and between 8 and 9.

Factor1	InitSeed1	Multip1	Shift1
5.66269	0.898	391	0

Factor1	InitSeed1	Multip1	Shift1
5.68618	0.225	423	8
5.91169	0.857	477	0
6.27061	0.127	449	-6
7.15817	0.272	423	0
7.50197	0.925	405	0
8.3351	0.461	647	0
8.75151	0.304	437	2
9.72238	0.393	443	0
11.1123	0.254	443	-2
11.1374	0.641	443	2
11.5542	0.185	437	0
11.6272	0.666	371	0
13.9538	0.503	437	-2
14.5575	0.409	423	-8
16.3378	0.713	449	6
17.3422	0.213	741	8
18.4295	0.300	449	0

Table 14. The second same results for algorithm A5 using scheme 3.

The conclusion drawn from scheme 3 is that multipliers and shift values with fractional part seem to degrade the RNG process. Thus, avoiding random values, with fractional parts, for the multipliers and shift is highly recommended. Negative integer shift values show a reasonable promise for generating good random number sequences.

CONCLUSION

This part of the PRNG study confirms that algorithm A3 has succeeded to hold its own place in the lead. Scheme 1 showed a slight advantage obtained by algorithms A4 and A5 when using initial seed values with 3 decimal places.

Scheme 2 is the true highlight of this study. It shows that supplying initial seed values with 7 or more decimal places favors the A3 algorithm. This conclusion is made since the slightly more elaborate algorithms A4 and A5 yield results that exactly match algorithm A3. In other words, adding more protective decision-making constructs in the RNG loop does not give an advantage when using good initial seeds.

Scheme 3 showed that using random multipliers and shift values (with fractional part) did poorly. The integer-value multiplier and shift parameters were able to produce factor values below 40. The conclusion drawn here is that fractional parts in the multiplier and shift values *poison*, so to speak, the random number generation process.

DOCUMENT HISTORY

Date	*Version*	*Comments*
1/30/2015	0.90.00	Initial release.
2/28/2015	1.00.00	Almost a complete rewrite of this paper that introduced algorithms A4 and A5.

