New Pseudo Random Number Generators: Part 3

By
Namir C. Shammas

INTRODUCTION

This part of the trilogy articles looks at expanding and maturing the factor statistic by adding the results from the change-of-sign test and the Kolmogorov-Smirnov statistics. This article shows the recalculations of the results of the train A3 algorithm for multipliers between 101 and 997. In addition the study compares the train A3 algorithm with popular PRNGs using both the old and new expressions for calculating the factor. This comparison should give us an idea about the increase in the factor values due to the addition of two randomness measurements.

THE PENALTY FACTOR

The values for the updated factor depend on the following statistics related to the random numbers generated. The new components of the factor appear in red text:

- 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 $F_n(x)$ = (number of $x_i <= x)/n$ 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:

$$\text{chsStat} = \Sigma D(i,1)*i)/D(1,1) + \Sigma D(i,2)*i)/D(1,2) \text{ for } i=2,\ldots,n \quad (1)$$

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 new 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 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.
- Ten times the change-of-sign statistic.
- Ten times the sum of the K_+ and K_- values.

Thus the calculated 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.

Scheme 2 Take 2

In this section I show the updated results of scheme 2 calculations (from part 2) using the updated factor calculations. Recall that the second scheme performs a more detailed exploration of algorithms A3 using a wide 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:

- 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.
- Each enumerated multiplier has shift values of 0 and 2.

- The initial seed starts at 0.00135711 and moves up in increments of .001. This sequence of values ensures that the initial seeds have plenty of decimal places.

Table 1 shows the best factors, that are less than 36, obtained using scheme 2 calculations with the new way of computing the factor. The multiplier 145 and shift value of zero are still in the lead! Their factor value has increased by about 29. This increase is due to adding the change-of-sign and Kolmogorov-Smirnov statistics.

Factor	*Initial Seed*	*Multiplier*	*Shift*
33.3619	0.724357	145	0
33.6723	0.115357	353	0
33.7169	0.094357	201	0
33.8032	0.991357	533	0
33.831	0.196357	315	2
33.8967	0.080357	453	0
33.9862	0.959357	351	0
34.0005	0.418357	273	0
34.0431	0.485357	167	2
34.0469	0.307357	113	0
34.1301	0.403357	273	2
34.1408	0.499357	327	2
34.1854	0.916357	261	2
34.2571	0.172357	463	0
34.2746	0.127357	251	2
34.3102	0.719357	107	2
34.4704	0.420357	253	0
34.5107	0.796357	133	0
34.5195	0.215357	473	2
34.6314	0.889357	425	2
34.6603	0.612357	415	0
34.7684	0.427357	285	0
34.8012	0.547357	333	2
34.823	0.661357	225	0
34.8369	0.553357	161	2
34.869	0.506357	551	0
34.9021	0.608357	131	0
34.9123	0.274357	521	2
34.9154	0.343357	253	2

Factor	*Initial Seed*	*Multiplier*	*Shift*
34.9272	0.137357	391	0
34.9292	0.672357	283	0
34.9643	0.173357	263	0
35.0019	0.465357	145	2
35.039	0.255357	381	0
35.0439	0.886357	527	2
35.0592	0.989357	335	2
35.0732	0.136357	477	0
35.0811	0.802357	153	2
35.1242	0.609357	155	2
35.1358	0.837357	377	2
35.1803	0.444357	351	2
35.1908	0.501357	191	0
35.2341	0.454357	181	2
35.2844	0.214357	341	2
35.3251	0.393357	345	0
35.3422	0.537357	297	0
35.366	0.986357	541	0
35.4436	0.569357	131	2
35.4449	0.053357	123	2
35.4638	0.625357	115	0
35.5538	0.717357	211	0
35.5592	0.820357	307	2
35.5654	0.404357	325	2
35.5683	0.211357	433	2
35.5707	0.508357	513	2
35.5893	0.270357	287	2
35.5956	0.991357	163	0
35.6039	0.745357	393	2
35.6462	0.573357	213	2
35.6873	0.384357	215	2
35.6878	0.138357	275	0
35.6958	0.221357	241	2
35.7551	0.495357	307	0
35.7765	0.029357	471	2
35.794	0.943357	361	0
35.8169	0.449357	141	2
35.8247	0.981357	455	2

Factor	*Initial Seed*	*Multiplier*	*Shift*
35.8408	0.544357	153	0
35.8444	0.789357	441	2
35.8494	0.325357	263	2
35.861	0.823357	435	2
35.8648	0.863357	223	2
35.9125	0.797357	543	2
35.9642	0.517357	247	2

Table 1. The best factor values obtained from using scheme 3 with the new factor calculations.

Table 2 shows a summary of factor ranges obtained in the scheme 2 calculations. Figure 1 shows the histogram for the data in Table 2. I deliberately divided the range of 0 to 40 into the range of 0 to 35 and 35 to 40. If you combine these two ranges then Figure 1 would clearly show an exponential decay in the number of high factor values.

From	*To*	*Frequency*
0	35	32
35	40	188
40	50	170
50	60	88
60	70	62
70	80	36
80	90	22
90	100	30
100	200	47
200	300	6
300	400	9
400	500	2
500	600	5
600	700	4
700	800	3
800	900	1
900	1000	6
1000	More	15

Table 2. The count for the factor values obtained from using scheme 3 with the new factor calculations.

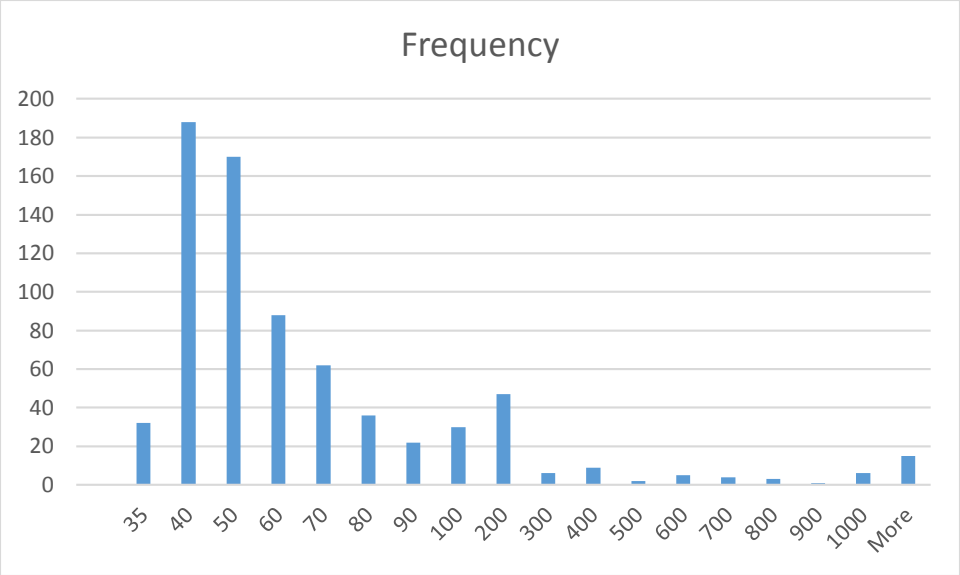

Figure 1. The histogram for the factor values obtained from using scheme 3 with the new factor calculations.

Performing a power fit for the following model:

$$\text{Ln(factor)} = a + b\ \ln(\text{InitSeed}) + c\ \ln(\text{Multiplier})$$

Gives the following results using the Excel regression tool:

SUMMARY OUTPUT

Regression Statistics	
Multiple R	0.470756
R Square	0.221612
Adjusted R Square	0.219458
Standard Error	0.775015
Observations	726

ANOVA

	df	*SS*	*MS*	*F*	*Significance F*
Regression	2	123.6387	61.81933	102.9211	4.65E-40
Residual	723	434.2684	0.600648		
Total	725	557.907			

	Coefficients	*Standard Error*	*t Stat*	*P-value*	*Lower 95%*	*Upper 95%*
Intercept	0.773713	0.299536	2.58304	0.009989	0.185649	1.361777
Ln(InitSeed)	-0.24162	0.028989	-8.33493	3.9E-16	-0.29853	-0.18471
Ln(Multip)	0.505349	0.04895	10.32378	2.11E-23	0.409248	0.60145

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

factor = 2.167801 * InitSeed^ 0.785355 * multiplier^ 1.657564

Which is roughly close to:

factor = 2 * InitSeed^(3/4) * multiplier^(3/2)

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.

If we create histograms (in steps of 50) for the multipliers for factors less than 40, in the range of 40 up to 50, in the range of 50 up to 60, and in the range of 60 up to 70 we get Figures 2, 3, 4, and 5. When you sequentially examine these figures, you see a wave that is moving from left to right. These histograms confirm the trend that has factors increasing with increasing multiplier values.

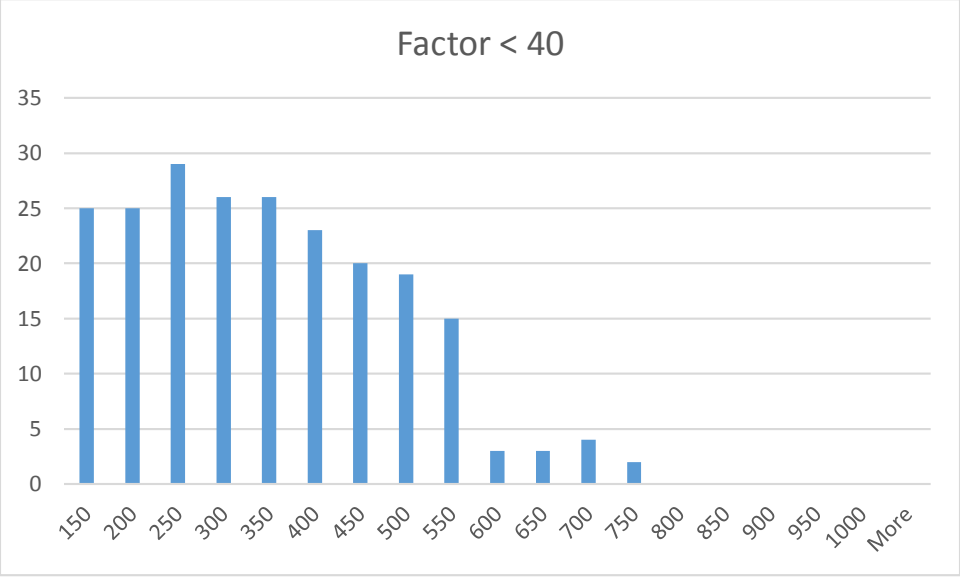

Figure 2. The histogram for the multipliers that have factors less than 40.

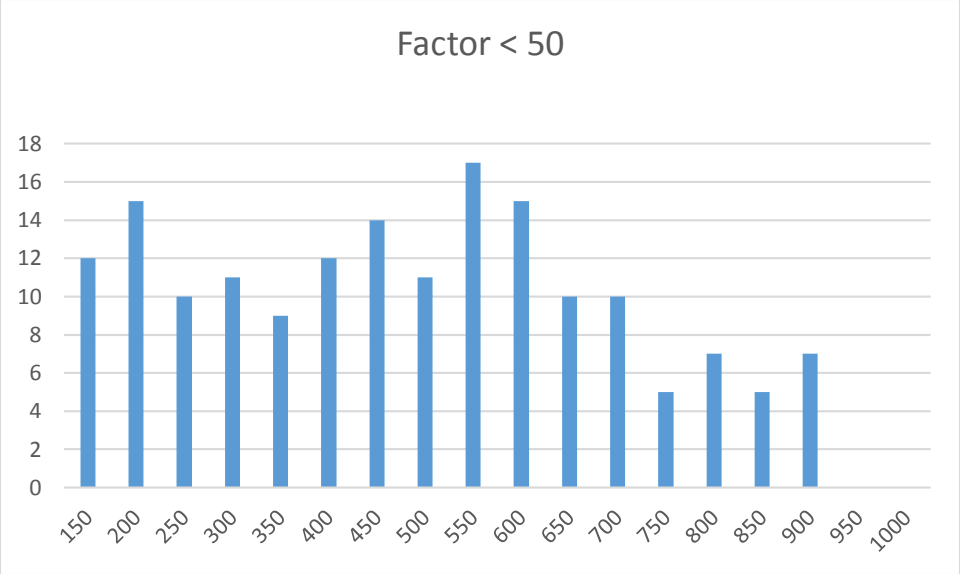

Figure 3. The histogram for the multipliers that have factors from 40 and up to 50.

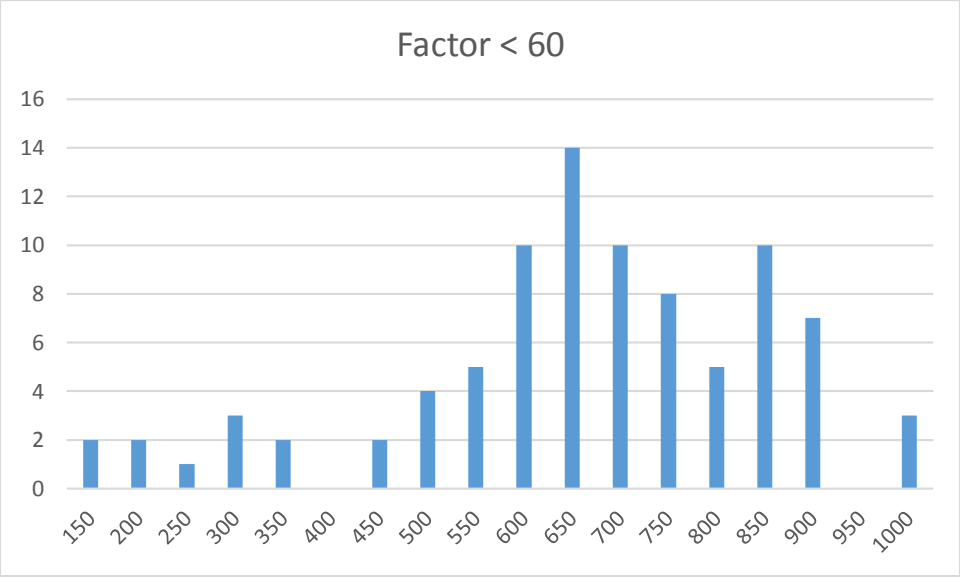

Figure 4. The histogram for the multipliers that have factors from 50 and up to 60.

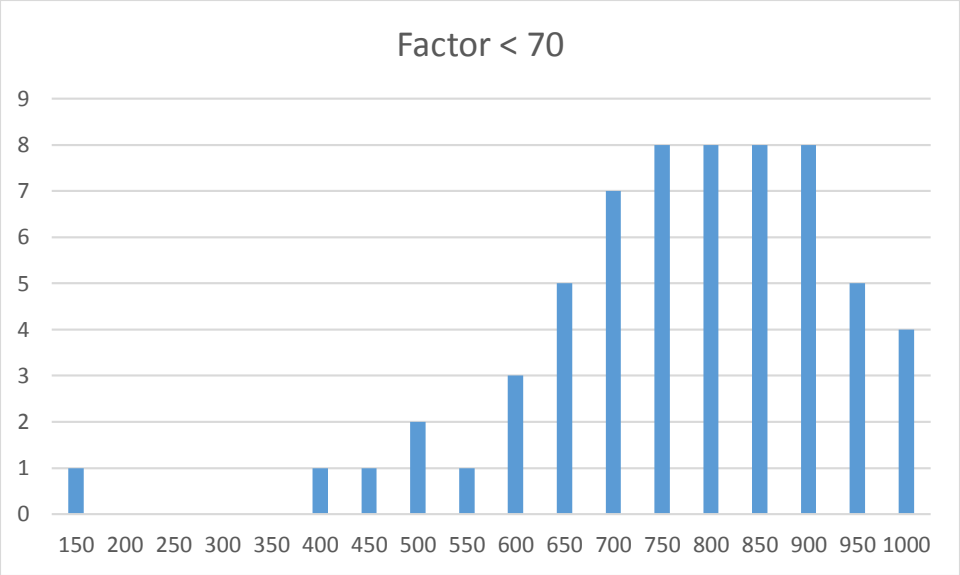

Figure 3. The histogram for the multipliers that have factors from 640 and up to 70.

Table 3 shows the mean and standard deviation values for the multipliers as grouped by ranges of factor values. The table shows the mean of the multipliers increasing with increasing range.

From	*To*	*Stat*	*Value*
30	40	Mean	322.5091
		Sdev	143.2858
40	50	Mean	456.6588
		Sdev	212.6668
50	60	Mean	635.4773
		Sdev	192.3316
60	70	Mean	745.4194
		Sdev	161.476

Table 3. The mean and standard deviation values for the multipliers as grouped by ranges of factor values.

A linear regression between the mean multiplier value (as the dependent variable) and the upper range value (as the independent variable) yields the following equation with a coefficient of determination, R^2, of 0.9925:

$$\text{Mean\_multiplier} = -256.136 + 14.47549 * \text{Upper\_Factor\_Range}$$

The above linear regression equation shows that higher factors are generated by higher values of the multipliers. You can invert the above equation and obtain the following linear relation:

Upper_FactorRange = 17.69445 + 0.069082 * Mean_multiplier

Comparing Algorithm A3 with other common PRNGS

How does the train A3 algorithm compare with common PRNGS, mostly used in generating random numbers for computer applications? I selected a number of commonly used PRNGS and applied calculations using the old factors and the new one. The common algorithms I used are:

- Two versions used by Apple computers.
- The 977*r algorithm,
- The 147*r algorithm
- The $(\pi+r)^5$ algorithm.
- The ANSI C algorithm.
- The BCPL algorithm.
- The Fishman LCGS algorithm.
- The Matlab PRNG.
- The Whichmann-Hill algorithm.
- The Numerical Recipes algorithm.
- The SimScript algorithm.
- The Super-Duper algorithm.
- The L'Ecuyer algorithm.
- The Borland C++ algorithm.
- The Borland Delphi algorithm.
- The Microsoft Visual C++ algorithm.
- The Microsoft Visual Basic 6 algorithm.
- The RANDU algorithm.

The definitions of most of the above PRNGs are found in Wikipedia. I recommend you consult Wikipedia for that information. Alternatively, you can look in the folders (downloadable from my web site) *PRNG Common Generator test Gen 1* or *RNG Common Generator test Gen 2* and inspect the various Matlab files that contain the code for the various PRNGs. I ran a Matlab code that supplied random seeds to the above algorithms as well as to the train A3 algorithm.

Results Using the Old Factor

I ran the test for the algorithms three times and combined the results. Table 4 shows the factors for less than 30. The algorithm A3 came in the lead followed by the Apple2 PRNG. Only these two algorithms performed consistently well. The other algorithms altered ranks. Algorithm A3 showed a few factor values that exceeded 300. These results were filtered out for algorithm A3 and a few other algorithms.

Method	*Min*	*Max*	*Mean*	*Sdev*
Algorithm A3	3.97762	297.884	15.961	18.5666
Algorithm A3	4.07746	200.58	15.2848	15.5548
Algorithm A3	4.10719	296.466	15.7138	18.2244
Apple2	10.5513	71.0327	35.3722	10.7902
Apple2	10.595	100.181	35.7397	10.4135
Apple2	11.192	100.414	35.1174	10.6141
L`Ecuyer	26.6239	75.5872	46.8208	7.33189
Matlab rand	27.1777	77.131	46.4749	7.14993
Num Recipes	27.3584	82.8856	46.6458	7.57688
Super-Duper	28.0759	86.1288	46.9369	7.4413
Rng997	28.2246	76.7838	46.7941	7.38212
MS Visual C++	28.4778	76.6173	46.5856	7.16646
Super-Duper	28.5591	70.8764	46.6241	7.05973
Wichmann-Hill	28.6195	79.533	46.2561	7.25884
RANDU	28.7768	74.1421	46.8599	7.29605
L`Ecuyer	29.1564	81.6356	46.4845	7.24898
Rng997	29.203	75.2963	46.6025	7.16121
MS Visual Basic	29.2587	78.6755	46.7395	7.16813
Matlab rand	29.4367	74.9319	46.831	7.40842
Rng147	29.4547	79.6679	47.0178	7.09848
Wichmann-Hill	29.4869	73.0375	46.7985	7.22143
Matlab rand	29.6442	81.9051	46.788	7.02363
Num Recipes	29.9347	80.4611	46.5719	7.4477
Rng147	29.9533	80.251	47.3764	7.39952

Table 4. The results of comparing common PRNG algorithms using the old factor calculations.

Results Using the New Factor

I ran the test for the algorithms three times and combined the results. Table 5 shows the factors for less than 66. Once again, the train A3 algorithms and Apple2

came in the lead. The other algorithms altered ranks with factor values starting above 60. The results of tables 3 and 4 shows that there is at least a difference of 29 between the results of the old and new factor calculations. The factors of 60 seem to be the general minimum value and a good baseline value to use in examining other PRNGs not included in this paper. The train A3 algorithm and the Apple2 algorithm are able to produce random numbers with factors below 60.

Method	*Min*	*Max*	*Mean*	*Sdev*
Algorithm A3	33.4175	222.748	47.2444	15.4019
Algorithm A3	33.485	177.238	46.851	13.2886
Algorithm A3	33.5696	222.763	47.2347	15.6687
Apple2	40.0355	132.185	74.0165	13.226
Apple2	40.2205	104.165	74.2027	12.5738
Apple2	40.5181	103.624	74.2796	13.1178
Num Recipes	61.418	126.05	86.9245	9.54399
Rng997	62.7189	122.014	86.6891	8.78067
MS Visual C++	63.675	121.638	86.6604	9.3267
L`Ecuyer	64.1455	120.865	86.7525	9.21878
MS Visual C++	64.2018	126.732	86.9597	8.88438
RANDU	64.3915	123.35	87.2154	9.34769
MS Visual C++	64.5331	132.196	86.6505	9.21205
(Pi+Rand)^5	64.6438	153.929	87.3957	9.79243
Wichmann-Hill	64.836	129.587	86.7375	9.33777
Wichmann-Hill	65.1221	125.158	87.0651	9.41457
(Pi+Rand)^5	65.2292	125.35	87.6511	9.40748
RANDU	65.2373	126.73	86.4946	8.76665
Super-Duper	65.5379	124.056	86.7352	8.85417
MS Visual Basic	65.5402	133.234	86.9507	9.37381
Matlab rand	65.5488	132.019	86.6577	9.36276
Rng147	65.6712	127.342	87.7314	9.69041
Matlab rand	65.7448	130.09	87.0531	9.11979
Super-Duper	65.7695	123.072	86.7641	9.32998
Num Recipes	65.7901	129.761	86.8058	9.60342
(Pi+Rand)^5	65.9142	127.87	87.4296	9.26757
Wichmann-Hill	65.9473	127.094	86.8891	9.15363
Rng997	65.9636	125.505	87.2244	9.15307

Table 5. The results of comparing common PRNG algorithms using the new factor calculations.

Among the PRNG algorithms that did well, showing two or three entries, in Table 5 are:

- Numerical Recipes algorithm.
- The (997*r) algorithm.
- MS Visual C++ algorithm.
- RANDU algorithm, which is supposed to be inferior and faulty!
- (Pi+r)^5 algorithm.
- Wichmann-Hill algorithm.
- Supper-Duper algorithm.
- Matlab's rand() function.

Sample Matlab Code

To avoid having you wade through hundreds of web pages of Matlab code, I present a sample of two Matlab functions to show you the new calculations for the factor values. Here is the code for function rngSimpleVerA3Gen2 which shows the use of the new factor calculations with algorithm A3:

```
function factor =
rngSimpleVerA3Gen2(maxElems,multiplier,shift,initSeed,
bShowResults)
%UNTITLED2 Summary of this function goes here

  if ~exist('bShowResults','var') || isempty(bShowResults)
    bShowResults=false;
  end
%   Detailed explanation goes here
  fprintf('RNG version A3 Gen 2 with multiplier %g and shift %g
and initial seed = %g\n', multiplier,shift,initSeed);
  x=zeros(maxElems,1);
  if abs(frac(1000*initSeed))<1e-7
    initSeed=(frac(1000*frac(initSeed)) + 0.35711)/1000;
  end
  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,bShowResults);
  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 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');
    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
  Kplus=sqrt(n)*diffMaxPlus;
  Kminus=sqrt(n)*diffMaxMinus;
End
```

The nested function chs calculates the change-of-sign statistic. The nested function KStest calculates the Kolmogorov-Smirnov statistics. Please examine the code in these functions to get a good idea on the exact calculations they perform.

And here is the code for function rngSimpleGrabA3Gen2:

```
function [factor,minFactor,lastInitSeed,x] =
rngSimpleGrabA3Gen2(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
  if abs(frac(1000*initSeed))<1e-7
    initSeed=(frac(1000*frac(initSeed)) + 0.35711)/1000;
  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)
% Calculate the factor statistic for the array of random
nnumbers x.

  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)));
  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+10*chs(x);
  factor = factor + 10 *(Kplus + Kminus);
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
```



```
  Kplus=sqrt(n)*diffMaxPlus;
  Kminus=sqrt(n)*diffMaxMinus;
End
```

The rest of the Matlab files can be downloaded as file prng.zip from my web site. The files are grouped in folders by category of code. The folders that deal with the older version of the factors are labeled *Generation 1* or *Gen 1*. The folders that handle the updated factors calculations are labeled *Generation 2* or *Gen 2*.

CONCLUSION

Even with the updated method of calculating the factor for the PRNG, the train A3 algorithm remains ahead of the commonly used PRNGs. The results in this article affirm those in part 2, using the updated expressions for calculating the factor values.

PROLOGUE MARCH 1, 2015

After publishing the trilogy articles I kept tinkering with a few new ways to efficiently generate random numbers. I stumbled on the following method that worked well in Matlab. The newer version(s) of Matlab support a function called randperm which takes an argument n and returns an array of n integers where the values in the range (1, n) are placed in a random order. If you take the elements of the integer-value array and divide them by n+1, you get an array of uniformly distributed random numbers between 0 and 1. Of course, this approach is using a special kind of random number generator, so in a sense we are cheating. However, the results are very good. Here is the Matlab code fraction in the heart of the random number generation:

```
rng('shuffle','twister');
x=randperm(maxElems)/(maxElems+1);
factor=calcFactor(x,bShowResults);
```

The second statement does all the random number generation. Here is the code for my version of randperm (in case you are using an older version of Matlab) which I call myrandperm:

```
function x=myrandperm(n)
  x=1:n;
  while n>2
    j=1+fix(n*rand(1,1));
    temp=x(n);
```



```
    x(n)=x(j);
    x(j)=temp;
    n=n-1;
  end
end
```

Table 6 shows the results of using Matlab's randperm function and my own myrandperm function. The minimum value for the factors seem to rank third after the train A3 algorithm and the Apple2 algorithm. The standard deviation is small. You can consider the random permutation method as a good algorithm since it is easy to implement and each random number does not depend on the previous random number. In the case of function myrandperm, the code makes n calls to the Matlab rand function. The justification is that the random number produced have a better quality than those by function rand itself.

Method	Min	Max	Mean	Sdev
RandPerm1	49.0842	61.7664	55.1647	1.98798
RandPerm2	49.0117	61.5947	55.1353	2.02506

Table 6. The results of random permutation methods.

Finally, I present a different approach that uses array shuffling without invoking random number functions. The basic approach for the method that I present is to take an ordered array and shuffle it to yield as much as possible random order in the array elements. My first trial shuffled the array of integers in a manner similar to shuffling cards. The approach involved a few tricks and a lot of repetition to increase the randomness of the array elements. The second approach divides the array into ten buckets or pages, and then shuffles neighboring and distant buckets. This approach has the advantage that the number of repetition for the shuffling is less than my first attempt. Here is the Matlab code for the function that calculates the factor using the ten-bucket shuffling approach:

```
function factor = rngRandPerm4cGen2(maxElems,bShowResults)

if ~exist('bShowResults','var') || isempty(bShowResults)
    bShowResults=false;
  end
  if bShowResults, fprintf('Algorithm RandPerm4 rng test\n');
end
  x=1:maxElems;
  x=unsort(x)/(maxElems+1);
  factor=calcFactor(x,bShowResults);
  if isnan(factor), factor=1e99; end
```



```
end

function x=unsort(x)
  N=length(x);
  delta=10;
  n=fix(N/delta);
  spacing=fix(delta)/2:-1:1;
  for j=1:7
    for k=1:length(spacing)
      spc=spacing(k);
      for i=1:delta-1
        i1=1+(i-1)*n;
        i2=i1+n-1;
        i3=i1+spc*n;
        i4=i3+n-1;
        if i4>N, break; end
        if i1~=i3
          y=[x(i1:i2),x(i3:i4)];
          y=shuffle(y);
          x(i1:i2)=y(1:n);
          x(i3:i4)=y(n+1:2*n);
        end
      end
    end
  end
  %plot(x)
end

function x=shuffle(x)

  n=length(x);
  m=fix(n/2);
  primeArr=[1,primes(m)];
  for ii=1:14
    for k=1:length(primeArr)
      aprime=primeArr(k);
      for i=1:aprime:n-aprime
        t=x(i);
        x(i)=x(i+aprime);
        x(i+aprime)=t;
      end
    end
  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 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');
    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(n,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
  Kplus=sqrt(n)*diffMaxPlus;
  Kminus=sqrt(n)*diffMaxMinus;
end
```

The loop that generates the random numbers calls function unsort, shown in red text. This function divides the array x into ten buckets and then uses array y to temporarily merge the data from any two buckets. The function in turn calls function shuffle, also shown in red text, to perform shuffling the array y. The function unsort then writes the data from the array y back to the source buckets. The function shuffle repeats the shuffling process of any two buckets for 14 times. This value is optimum for Matlab and for generating 100,000 random numbers.

You can easily alter the returned parameters of function rngRandPerm4cGen2 by replacing factor with the array [factor, x], where x is the array of pseudo-random numbers.

The array shuffling algorithm generates a factor of 51.0668. While this value is slightly higher than the methods that use random number generation functions, it stands apart for not using such functions. Thus, the array shuffling method comes in third place, if you exclude the other methods that I presented in this section.

DOCUMENT HISTORY

Date	*Version*	*Comments*
2/28/2015	1.00.00	Initial release.
3/1/2015	1.01.00	Added the ***Prologue*** section.

