Project 997 PRNGs Part 1
By
Namir C. Shammas

1/ INTRODUCTION

This study started for a presentation at the HHC 2022 conference in Nashville, Tennesee. That presentation focused on a specific set of equations that use fraction based uniform PRNGs. Once I returned from that conference, I decided to expand the study of integer based uniform PRNGs and more fraction based PRNGs. This Part 1 study focuses on the fraction based uniform PRNGs.

2/ THE HHC 2022 FOCUS

In the HHC 2022 presentation I pointed out that the following legacy algorithm that HP used in its stat pacs was previously shown in HHC 2017 to be the best legacy (and very simple) fraction based PRNG:

$$r_{n+1} = frac(997*r_n) \quad (2.1)$$

Where r_n represents the current random number, and r_{n+1} represents the new random number. The HHC 2022 presentation focused on a more general form of equation 2.1:

$$r_{n+1} = frac(a + b*r_n) \quad (2.2)$$

The presentation used the approach of optimizing the penalty factor (see the Appendix) to yield the optimum values for parameters *a* and *b* for the best PRNGs.

☛ The process of optimizing a PRNG includes optimizing the parameters for the PRNG AS WELL AS THE INITIAL SEED! The results are excellent. However, to use these optimized PRNGs you MUST use the optimized initial PRNG seeds. This restriction seems limiting for many users who prefer to choose their own initial seeds. That is why I use a two-step optimization process followed by using diverse random seeds calculations.

The study examines the following sets of statistics for the penalty factors for each set of random numbers generated:

- Mean and standard deviation.
- Number of observations.
- Minimum and maximum values, and the range. The maximum value gives an idea how good and bad the penalty factor can become.
- Confidence interval for mean at 95% confidence.
- Upper and lower confidence range of the mean. This is important in algorithm qualification, especially the upper limit which gives an idea about the *average worst mean* penalty factor.
- The values for *a* and *b* used in the algorithms.
- The best initial seed for the random numbers, *xrnd*. This value yields the minimum penalty factor for that algorithm.

The conference study looked at the following PRNGs:

- Three optimum models Alg1, Alg2, and Alg3 for x(i+1)=frac(a+b*x(i)).
- The algorithm frac(997*r) which is the baseline algorithm.
- The algorithm frac(π + 997*r).
- The algorithm frac(1003*r).
- The algorithm frac(π + 1003*r).
- The algorithm frac(110011*r).
- The algorithm frac(π + 110011*r).
- The algorithm frac(9977*r).
- The algorithm frac(π + 9977*r).
- The algorithm frac(9997*r).
- The algorithm frac(π + 9997*r).

The results for the baseline algorithm (equation 2.1) and its close variant appear in table 2.1:

	Frac(997*r)	Frac(π + 997*r)
	10-digit Rand seed	10-digit Rand seed
Mean	149.084063	147.5983023
Sdev	68.28816232	12.55323217

Min	104.7174548	105.0377792
Max	12,864.04	3103.329788
Range	12,759.32	2998.292009
Count	1000000	1000000
Conf	0.153061273	0.028136849
CI Upper	149.2371243	147.6264391
CI Lower	148.9310017	147.5701654
Xrnd	0.178091719	
A	0	0.141592654
B	997	9997

Table 2.1. The statistics for the baseline algorithm and its simple variant.

Adding π improves the baseline PRNG considerably.

☛ The study (in this part and subsequent parts) seeks values for the upper mean confidence interval THAT ARE LESS THAN 148.

The results for Alg1, Alg2, and Alg3 are:

	Alg1	Alg2	Alg3
	10-digit Rand seed	10-digit Rand seed	10-digit Rand seed
Mean	147.692	147.7691853	148.5250036
Sdev	12.73101	19.87514982	16.59391457
Min	105.3204	105.5931784	106.1338145
Max	4807.038	15,037.52	3,915.15
Range	4701.718	14,931.92	3,809.02
Count	1000000	1000000	1000000
Conf	0.028535	0.044548215	0.037193645
CI Upper	147.7205	147.8137335	148.5621972
CI Lower	147.6635	147.7246371	148.48781
Xrnd	0.184347	0.747325827	0.227149167
A	0.8502	0.047907	0.77587
B	1237	15701	26301

Table 2.2. The statistics for Alg1, Alg2, and Alg3.

The above results show that the Alg1 and Alg2 algorithms meet the selection criteria. The HHC 2022 study showed that Alg2 was more stable than Alg1.

The algorithms Frac(π+110011*r) and Frac(π+9997*r) also show promising results. The rest of the algorithms listed earlier did not.

	Frac(π+110011*r)	Frac(π+9997*r)
	10-digit Rand seed	10-digit Random seed
Mean	147.9442861	147.5983023
Sdev	12.32780436	12.55323217
Min	102.2384784	105.0377792
Max	305.09	3103.329788
Range	202.85	2998.292009
Count	1000000	1000000
Conf	0.027631574	0.028136849
CI Upper	147.9719176	147.6264391
CI Lower	147.9166545	147.5701654
Xrnd	0.113082105	0.346118233
A	0.141592654	0.141592654
B	11011	9997

Table 2.3. The statistics for the two most promising PRNG algorithms.

The best results show in the HHC 2022 presentation are:

- The minimum penalty factor belongs to frac(π+9997*r).
- Algorithm frac(0.8502+1237*r).
- The best "stable" algorithm is Alg2, frac(0.047907 + 15701*r).
- Algorithm frac(π +110011*r).

3/ More Fraction Based PRNGs Set 1

This section looks at simple PRNGs that use the following equation:

$$IX_{n+1} = c*IX_n \bmod (2^{32} - 1) \quad (3.1)$$

$$R = IX_{n+1} / (2^{32} - 1) \quad (3.2)$$

Where c is a user-specified integer. IX0 = fix(r*(2^{32} – 1)) where r is a uniform random number in the range of (0,1).

The leading MATLAB code for the PRNG function is (the rest of the code calculates the penalty fact as shows in the Appendix):

```
function factor = rng997Gen1(maxElems, c, nDigits)
%UNTITLED2 Summary of this function goes here
  rng('shuffle','twister');
  x=zeros(maxElems,1);
  M = 2^32-1;
  IX = fix(M*rand);
  for i=1:maxElems
    IX = mod(c*IX,M);
    x(i) = IX / M;
  end
  x = round(x,nDigits);
  factor=calcFactor(x,false);
  if isnan(factor), factor=65535; end
end
...
```

Tables 3.1 and 3.2 show the results of using equations 3.1 and 3.2. The best value of c is 11111 (with a few trimmed data points). The other values of c that yield and upper confidence for the mean that is below 148 are 9989 (with a few trimmed data points), 1111, 9989, 1117, 997, 10101, and 991.

Value of c	11111	9989	11111	9989	1117
Mean	147.3378	147.430392	147.465569	147.5643456	147.7576412
Sdev	11.54862	11.6367443	33.9142082	32.62516826	13.65851225
Min	104.2365	106.21614	104.236482	106.2161398	104.2685806
Max	225.8453	217.884544	12836.7515	9398.619012	2942.310275
Range	121.6088	111.668405	12732.515	9292.402872	2838.041694
Count	999982	999980	1000000	1000000	1000000
Conf	0.025885	0.02608289	0.0760154	0.073126141	0.030614227
CI Upper	147.3637	147.456475	147.541584	147.6374717	147.7882555
CI Lower	147.3119	147.404309	147.389553	147.4912194	147.727027

Table 3.1. The first part of the results of using equations 3.1 and 3.2.

	997	10101	991	97	911003
Mean	147.7669774	147.7720797	147.8418031	148.8210072	170.8995089
Sdev	13.60391603	13.16059889	12.39322772	11.827355	17.20632593
Min	105.4219022	101.4632728	104.7943481	107.3007602	116.9106083
Max	3150.915315	1625.409076	1504.060586	231.2129813	3542.587509
Range	3045.493413	1523.945804	1399.266238	123.9122211	3425.676901
Count	1000000	1000000	1000000	999986	1000000
Conf	0.030491854	0.029498202	0.027778214	0.026510051	0.038566306
CI Upper	147.7974692	147.8015779	147.8695813	148.8475172	170.9380752
CI Lower	147.7364855	147.7425815	147.8140249	148.7944971	170.8609426

Table 3.2. The second part of the results of using equations 3.1 and 3.2.

4/ More Fraction Based PRNGs Set 2

This section looks at double-random number generators (DRNG). These are generators that create two random numbers and return the second one. I present four flavors of these double-random number generators.

All these DRNGs use the array [991 997 1117 9989 11111] as the source of constants. The algorithms use sequences made up of the element of the matrix of the above arrays (except the diagonal elements are ignored).

4.1/ Fraction based PRNGs Subset 1

The first type of double-random number generators uses the following equations:

$$z = \text{frac}(c1 * r_n) \quad (4.1.1)$$

$$r_{n+1} = \text{frac}(c2 * z) \quad (4.1.2)$$

Where c1 and c2 are values in the array [991 997 1117 9989 11111], such that c1 and c2 are never equal. The leading MATLAB code for the function that calculates the penalty factor based on the above equations is:

```
function factor = rng997Gen1(maxElems, c1, c2, nDigits)
%UNTITLED2 Summary of this function goes here
  rng('shuffle','twister');
  x=zeros(maxElems,1);
  x(1) = rand;
  for i=2:maxElems
     z = round(mod(x(i-1)*c1,1),nDigits);
     x(i) = round(mod(z*c2,1),nDigits);
  end
  factor=calcFactor(x,false);
  if isnan(factor), factor=65535; end
end
```

The parameters c1 and c2 are the values of the multipliers found in the array [991 997 1117 9989 11111].

The next tables show the results of the penalty factor statistics. The results are sorted in ascending order using the upper mean confidene values.

	991, 11111	11111, 991	1117, 997	997, 1117
Mean	142.4532898	143.3695766	145.719052	145.7960721
Sdev	286.406196	396.9037711	24.02019841	85.96511402
Min	41.369725	42.258258	56.708475	56.202139
Max	97302.00976	101616.3272	4416.397409	82955.99999

Range	**97260.64003**	**101574.0689**	**4359.688934**	**82899.79785**
Count	**999948**	**999944**	**1000000**	**999999**
Conf	**0.64196832**	**0.889646106**	**0.053838938**	**0.192682537**
CI Upper	**143.0952581**	**144.2592227**	**145.7728909**	**145.9887546**
CI Lower	**141.8113215**	**142.4799305**	**145.6652131**	**145.6033896**

Table 4.1.1. The results of the penalty factor statistics.

Table 4.1.1 shows very good results where the the upper mean confidene values are below 146! The results are symmetric in that the first two results columns belong to 991 and 1111. Likewise, the results of the last two columns belong to 997 and 1117.

	991, 9989	**9989, 991**	**997, 9989**	**9989, 997**
Mean	**147.0158838**	**147.0299846**	**147.7189768**	**147.7191093**
Sdev	**14.54305956**	**15.05729811**	**11.93994337**	**11.90360118**
Min	**52.94438813**	**53.173338**	**49.068753**	**49.359399**
Max	**2404.118974**	**4308.649974**	**963.00152**	**1106.448268**
Range	**2351.174586**	**4255.476636**	**913.932767**	**1057.088869**
Count	**1000000**	**1000000**	**1000000**	**1000000**
Conf	**0.032596853**	**0.033749469**	**0.026762222**	**0.026680764**
CI Upper	**147.0484807**	**147.063734**	**147.745739**	**147.74579**
CI Lower	**146.983287**	**146.9962351**	**147.6922146**	**147.6924285**

Table 4.1.2. The results of the penalty factor statistics (cont.).

	9989, 1117	**1117, 9989**	**997, 991**	**9989, 11111**
Mean	**147.7340121**	**147.7341885**	**147.7364101**	**147.73849**
Sdev	**11.82301291**	**12.71400539**	**11.86566209**	**11.8120017**
Min	**57.9027**	**54.947202**	**58.9998914**	**46.124159**
Max	**684.660836**	**4358.652492**	**643.7539051**	**435.826581**
Range	**626.758136**	**4303.70529**	**584.7540137**	**389.702422**
Count	**1000000**	**1000000**	**1000000**	**1000000**
Conf	**0.026500133**	**0.028497206**	**0.026595727**	**0.02647545**
CI Upper	**147.7605122**	**147.7626857**	**147.7630059**	**147.764965**

CI Lower	147.7075119	147.7056913	147.7098144	147.712014

Table 4.1.3. The results of the penalty factor statistics (cont.).

	991, 997	1111, 9989	11111, 997	997, 11111
Mean	147.738892	147.7393818	147.7450841	147.757196
Sdev	12.0216592	11.82405729	11.80790676	11.7993041
Min	58.436788	44.501768	77.781553	76.005645
Max	2044.72284	689.099431	703.023756	593.761305
Range	1986.28605	644.597663	625.242203	517.75566
Count	1000000	1000000	1000000	1000000
Conf	0.02694538	0.026502474	0.026466274	0.02644699
CI Upper	147.765837	147.7658843	147.7715504	147.783643
CI Lower	147.711947	147.7128794	147.7186178	147.730749

Table 4.1.4. The results of the penalty factor statistics (cont.).

	991, 1117	1117, 991	1117, 11111	1111, 1117
Mean	147.774633	147.77636	147.77636	147.7830005
Sdev	11.8499896	11.8463069	11.8463069	11.94057025
Min	54.6301811	54.6189992	54.6189992	52.769769
Max	910.027694	693.63608	693.63608	1329.481435
Range	855.397513	639.017081	639.017081	1276.711666
Count	1000000	1000000	1000000	1000000
Conf	0.0265606	0.02655234	0.02655234	0.026763627
CI Upper	147.801193	147.802912	147.802912	147.8097641
CI Lower	147.748072	147.749808	147.749808	147.7562369

Table 4.1.5. The results of the penalty factor statistics (cont.).

The results in Tables 4.1.2 through 4.1.5 are good as they show values for the upper mean confidence values below 148.

4.2/ Fraction based PRNGs Subset 2

The second type of double-random number generators uses the following equations:

$z = \text{frac}(c1 * r_n)$ (4.2.1)

$r_{n+1} = \text{frac}(\pi + c2*z)$ (4.2.2)

Where c1 and c2 are integers in the array [991 997 1117 9989 11111], such that c1 and c2 are never equal. The leading MATLAB code for the function that calculates the penalty factor based on the above equations is:

```
function factor = rng997Gen2(maxElems, c1, c2, nDigits)
%UNTITLED2 Summary of this function goes here
  rng('shuffle','twister');
  x=zeros(maxElems,1);
  x(1) = rand;
  for i=2:maxElems
     z = round(mod(x(i-1)*c1,1),nDigits);
     x(i) = round(mod(pi+z*c2,1),nDigits);
  end
  factor=calcFactor(x,false);
  if isnan(factor), factor=65535; end
end
```

The parameters c1 and c2 are the values of the multipliers found in the array [991 997 1117 9989 11111].

The next tables show the results of the penalty factor statistics. The results are sorted in ascending order using the upper mean confidene values.

	1117, 997	997, 1117	11111, 991	991, 11111
Mean	146.521817	146.541178	146.601915	146.6090653
Sdev	16.9882487	17.3839264	12.26152792	12.26999382
Min	75.477558	71.475384	100.239467	99.645322
Max	1211.67764	1610.94431	218.840164	219.821854
Range	1136.20008	1539.46893	118.600697	120.176532
Count	1000000	1000000	1000000	1000000
Conf	0.03807751	0.03896438	0.027483022	0.027501998
CI Upper	146.559894	146.580143	146.629398	146.6365673
CI Lower	146.483739	146.502214	146.5744319	146.5815633

Table 4.2.1. The results of the penalty factor statistics.

Table 4.2.1 shows very good results where the the upper mean confidene values are below 147! Compared to the results of Table 4.1.1 we can conclude that using the value of π did not improve the PRNGs in equations 4.2.1 and 4.2.2.

	991, 9989	9989, 991	11111, 1117	1117, 9989

Mean	**146.9901676**	**147.013034**	**147.7259206**	**147.728387**
Sdev	**14.96411904**	**17.000439**	**11.86231135**	**11.7503705**
Min	**55.797042**	**57.094894**	**50.370471**	**92.222338**
Max	**5208.597622**	**9122.27429**	**1455.45141**	**698.544704**
Range	**5152.80058**	**9065.1794**	**1405.080939**	**606.322366**
Count	**1000000**	**1000000**	**1000000**	**1000000**
Conf	**0.033540617**	**0.03810483**	**0.026588217**	**0.02633731**
CI Upper	**147.0237082**	**147.051138**	**147.7525088**	**147.754724**
CI Lower	**146.956627**	**146.974929**	**147.6993323**	**147.702049**

Table 4.2.2. The results of the penalty factor statistics.

	9989, 1117	**9989, 997**	**997, 11111**	**11111, 997**
Mean	**147.733482**	**147.73479**	**147.736369**	**147.7413108**
Sdev	**11.8850783**	**11.967004**	**11.8124728**	**11.81092722**
Min	**90.907176**	**52.558883**	**56.086426**	**56.388226**
Max	**1817.66065**	**1491.2001**	**944.473394**	**740.586953**
Range	**1726.75348**	**1438.6413**	**888.386968**	**684.198727**
Count	**1000000**	**1000000**	**1000000**	**1000000**
Conf	**0.02663925**	**0.0268229**	**0.02647651**	**0.026473044**
CI Upper	**147.760121**	**147.76161**	**147.762845**	**147.7677838**
CI Lower	**147.706843**	**147.70796**	**147.709892**	**147.7148377**

Table 4.2.3. The results of the penalty factor statistics.

	997, 991	**997, 9989**	**991, 997**	**1117, 11111**
Mean	**147.74131**	**147.74067**	**147.741497**	**147.7420627**
Sdev	**11.858166**	**12.222815**	**12.0096608**	**11.78837889**
Min	**58.29748**	**52.665324**	**55.738003**	**49.432003**
Max	**462.05887**	**2970.7743**	**1762.91446**	**468.997888**
Range	**403.76139**	**2918.109**	**1707.17646**	**419.565885**
Count	**1000000**	**1000000**	**1000000**	**1000000**
Conf	**0.0265789**	**0.0273963**	**0.02691849**	**0.026422505**
CI Upper	**147.76789**	**147.76807**	**147.768415**	**147.7684852**

CI Lower	147.71474	147.71328	147.714578	147.7156402

Table 4.2.4. The results of the penalty factor statistics.

	9989, 11111	11111, 9989	991, 1117	1117, 991
Mean	147.745387	147.749548	147.7706799	147.7818959
Sdev	12.4322964	11.9318643	11.80701308	11.82352587
Min	53.018094	49.82348	49.62004	51.390634
Max	3968.32541	1567.68114	489.839992	291.162331
Range	3915.30732	1517.85766	440.219952	239.771697
Count	1000000	1000000	1000000	1000000
Conf	0.02786578	0.02674411	0.026464271	0.026501283
CI Upper	147.773253	147.776292	147.7971441	147.8083972
CI Lower	147.717521	147.722804	147.7442156	147.7553946

Table 4.2.5. The results of the penalty factor statistics.

4.3/ Fraction based PRNGs Subset 3

The third type of double-random number generators uses the following algorithm:

If $r_n < 0.5$ then

$$z = \text{frac}(c1 * r_n) \quad (4.3.1)$$

$$r_{n+1} = \text{frac}(c2 * z) \quad (4.3.2)$$

else

$$z = \text{frac}(c2 * r_n) \quad (4.3.3)$$

$$r_{n+1} = \text{frac}(c1 * z) \quad (4.3.4)$$

end if

Where c1 and c2 are integers in the array [991 997 1117 9989 11111], such that c1 and c2 are never equal. The leading MATLAB code for the function that calculates the penalty factor based on the above equations is:

```
function factor = rng997Gen3(maxElems, c1, c2, nDigits)
%UNTITLED2 Summary of this function goes here
  rng('shuffle','twister');
  x = zeros(maxElems,1);
  x(1) = rand;
  for i=2:maxElems
    if x(i-1) <= 0.5
```



```
      z = round(mod(x(i-1)*c1,1),nDigits);
      x(i) = round(mod(z*c2,1),nDigits);
    else
      z = round(mod(x(i-1)*c2,1),nDigits);
      x(i) = round(mod(z*c1,1),nDigits);
    end
  end
  factor=calcFactor(x,false);
  if isnan(factor), factor=65535; end
end
```

The next tables show the results of the penalty factor statistics. The results are sorted in ascending order using the upper mean confidene values.

	991, 11111	997, 1117	991, 9989	991, 997
Mean	142.5524994	145.7277177	146.9920276	147.7277126
Sdev	288.7813175	23.01036757	14.4650768	11.86339256
Min	41.690705	58.06775	53.818746	58.151892
Max	99601.33217	4105.612417	2627.380164	472.455684
Range	99559.64146	4047.544667	2573.561418	414.303792
Count	999953	1000000	1000000	1000000
Conf	0.647290444	0.051575501	0.032422063	0.02659064
CI Upper	143.1997898	145.7792932	147.0244496	147.7543032
CI Lower	141.9052089	145.6761422	146.9596055	147.7011219

Table 4.3.1. The results of the penalty factor statistics.

Table 4.3.1 shows very good results where the the upper mean confidene values for the two left data columns are below 146! In fact, the results of the left most data columns is the best so far.

	997, 9989	9989, 11111	1117, 9989	997, 11111
Mean	147.728692	147.737474	147.7381929	147.7413793
Sdev	12.45502987	11.91668438	11.94483851	11.79054007
Min	49.882359	45.230354	53.731633	75.951167
Max	3419.868765	1647.123117	1281.087524	396.057603
Range	3369.986406	1601.892763	1227.355891	320.106436
Count	1000000	1000000	1000000	1000000
Conf	0.027916738	0.026710089	0.026773194	0.026427349

CI Upper	147.7566087	147.764184	147.7649661	147.7678066
CI Lower	147.7007752	147.7107639	147.7114197	147.7149519

Table 4.3.2. The results of the penalty factor statistics.

	1117, 1111	991, 1117
Mean	147.7504584	147.777089
Sdev	11.79041323	12.40453866
Min	51.94002	53.940579
Max	448.912386	3495.453545
Range	396.972366	3441.512966
Count	1000000	1000000
Conf	0.026427064	0.027803567
CI Upper	147.7768855	147.8048926
CI Lower	147.7240313	147.7492855

Table 4.3.3. The results of the penalty factor statistics.

The remaining results in the above tables show good values for the upper mean values that are less than 148.

4.4/ Fraction based PRNGs Subset 4

The fourth type of double-random number generators uses the following algorithm:

If $r_n < 0.5$ then

$$z = \text{frac}(\pi + c1*r_n) \quad (4.4.1)$$

$$r_{n+1} = \text{frac}(\pi + c2*z) \quad (4.4.2)$$

else

$$z = \text{frac}(\pi + c2*r_n) \quad (4.4.3)$$

$$r_{n+1} = \text{frac}(\pi + c1*z) \quad (4.4.4)$$

end if

Where c1 and c2 are integers in the array [991 997 1117 9989 11111], such that c1 and c2 are never equal. The leading MATLAB code for the function that calculates the penalty factor based on the above equations is:

```
function factor = rng997Gen3(maxElems, c1, c2, nDigits)
```



```
%UNTITLED2 Summary of this function goes here
  rng('shuffle','twister');
  x = zeros(maxElems,1);
  x(1) = rand;
  for i=2:maxElems
    if x(i-1) <= 0.5
      z = round(mod(x(i-1)*c1,1),nDigits);
      x(i) = round(mod(z*c2,1),nDigits);
    else
      z = round(mod(x(i-1)*c2,1),nDigits);
      x(i) = round(mod(z*c1,1),nDigits);
    end
  end
  factor=calcFactor(x,false);
  if isnan(factor), factor=65535; end
end
```

The next tables show the results of the penalty factor statistics. The results are sorted in ascending order using the upper mean confidene values.

	1117, 9989	1117, 11111	991, 997	9989, 11111
Mean	147.7414968	148.5321536	149.0897552	149.25322
Sdev	12.00966084	49.19675086	72.33458022	66.21821203
Min	55.738003	107.404609	106.015059	106.214569
Max	1762.914462	28764.25649	16025.27426	9198.689357
Range	1707.176459	28656.85189	15919.25921	9092.474788
Count	1000000	1000000	1000000	1000000
Conf	0.026918487	0.110269732	0.162130925	0.148421681
CI Upper	147.7684153	148.6424233	149.2518861	149.4016417
CI Lower	147.7145783	148.4218839	148.9276242	149.1047983

Table 4.4.1. The results of the penalty factor statistics.

Table 4.4.1 shows disappointing results. Only the leftmost data colum has an upper mean value below 148. The remaining columns have upper mean values above 148. It gets worse in the remaining two tables!

	997, 11111	997, 9989	991, 1117	991, 9989
Mean	149.25322	150.0478361	150.7026971	158.471012
Sdev	66.21821203	72.75344309	70.99917403	366.5291446
Min	106.214569	103.74546	107.79708	105.055159
Max	9198.689357	49614.3818	10836.16752	94797.67601
Range	9092.474788	49510.63634	10728.37044	94692.62085

Count	1000000	1000000	1000000	1000000
Conf	0.148421681	0.163069766	0.159137742	0.821539425
CI Upper	149.4016417	150.2109059	150.8618349	159.2925514
CI Lower	149.1047983	149.8847663	150.5435594	157.6494726

Table 4.4.2. The results of the penalty factor statistics.

	991, 11111	997, 1117
Mean	160.6130305	251.9056575
Sdev	42.8437274	736.8921854
Min	105.714223	107.806424
Max	3001.951281	94195.9157
Range	2896.237058	94088.10928
Count	1000000	1000000
Conf	0.096030047	1.651672154
CI Upper	160.7090606	253.5573297
CI Lower	160.5170005	250.2539854

Table 4.4.3. The results of the penalty factor statistics.

The above tables show that the algorithm used in this section does not satisfy the expectations for a good PRNG.

5/ CONCLUSION

The next two tables show the best ten PRNGS in this study. The tables contain references to the equations used by the PRNG algorithms.

Coeff	991, 11111	991, 11111	11111, 991	1117, 997	997, 1117
Equations	4.1.1, 4.1.2	4.3.1/4.3.4	4.1.1, 4.1.2	4.1.1, 4.1.2	4.3.1/4.3.4
Mean	142.4532898	142.5524994	143.37	145.719	145.728
Sdev	286.406196	288.7813175	396.904	24.0202	23.0104
Min	41.369725	41.690705	42.2583	56.7085	58.0678
Max	97302.00976	99601.33217	101616	4416.4	4105.61
Range	97260.64003	99559.64146	101574	4359.69	4047.54
Count	999948	999953	999944	1000000	1000000

Conf	0.64196832	0.647290444	0.88965	0.05384	0.05158
CI Upper	143.0952581	143.1997898	144.259	145.773	145.779
CI Lower	141.8113215	141.9052089	142.48	145.665	145.676

Table 5.1. The best PRNGs in this study.

Coeff	997, 1117	1117, 997	997, 1117	11111, 991	991, 11111
Equations	4.1.1, 4.1.2	4.2.1, 4.2.2	4.2.1, 4.2.2	4.2.1, 4.2.2	4.2.1, 4.2.2
Mean	145.796	146.522	146.541	146.601915	146.609
Sdev	85.9651	16.9882	17.3839	12.26152792	12.27
Min	56.2021	75.4776	71.4754	100.239467	99.6453
Max	82956	1211.68	1610.94	218.840164	219.822
Range	82899.8	1136.2	1539.47	118.600697	120.177
Count	999999	1000000	1000000	1000000	1000000
Conf	0.19268	0.03808	0.03896	0.027483022	0.0275
CI Upper	145.989	146.56	146.58	146.629398	146.637
CI Lower	145.603	146.484	146.502	146.5744319	146.582

Table 5.2. The best PRNGs in this study.

The best algorithm is:

$z = \text{frac}(991 * r_n)$ (5.1)

$r_{n+1} = \text{frac}(11111 * z)$ (5.2)

APPENDIX-THE PENALTY FACTOR

There are several tests for measuring the randomness of a sequence of uniformly distributed random numbers. The most famous battery of tests is the diehard test. I have devised my own test which calculates a penalty factor. The lower this factor the better the sequence of random numbers generated. The values for a calculated penalty factor depend on the count of random numbers generated. This study is based on consistently generating sequences of 1000,000 random numbers. Lowering the count of random numbers generated tends to increase the values for the factors. One reason is the random numbers may appear slightly more auto correlated when there are fewer of them. The values for the factor depend on the following statistics related to the random numbers generated:

- The mean value.
- The standard deviation.
- The maximum and minimum autocorrelations taken for 1 to 1000 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 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 statistics. 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 consecutive signs constantly and systematically alternating between positive and negative. However, real-world PRNGs will have 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 penalty factor using:

$$\text{Factor} = 1000\,[|\text{mean} - 0.5| + |\text{sdev} - 1/\sqrt{12}|] + 100\,(\text{max\_autoCorrel} - \text{min\_autoCorrel})) + 100 \cdot \text{AutoCorrSum} + \text{ChiSqr10} + \text{ChiSqr20} / 2 + 10 \cdot \text{chsStat} + 10\,(K_+ + K_-) \quad (2)$$

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

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

Thus, the calculated penalty factor measures the following:

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

- The closeness of the cumulative distribution of the numbers generated to the ideal cumulative distribution.

Here is the source code for the functions that calculate the penalty factor. These functions appear in the function that calculates the factor for a set of random numbers generated by a specific algorithm.

```
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 consequtive positive
% abd 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 happpens 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
```

DOCUMENT HISTORY

Date	*Version*	*Comments*
2/10/2023	1.00.00	Initial release.

