CDF/MEMO/STATISTICS/PUBLIC/8032

TRandom Pitfalls

Joel Heinrich—University of Pennsylvania

January 13, 2006

1 Introduction

The purpose of this note is to warn ROOT users about specific traps, defects and pitfalls associated with ROOT's basic random number class `TRandom`[1]. This class provides a uniform pseudorandom number generator, and various nonuniform generators: Gaussian, Poisson, binomial, Breit-Wigner, and others. The declaration of the class is:

```
class TRandom : public TNamed {

protected:
   UInt_t   fSeed;  //Random number generator seed

public:
   TRandom(UInt_t seed=65539);
   virtual ~TRandom();
   virtual  Int_t    Binomial(Int_t ntot, Double_t prob);
   virtual  Double_t BreitWigner(Double_t mean=0, Double_t gamma=1);
   virtual  Double_t Exp(Double_t tau);
   virtual  Double_t Gaus(Double_t mean=0, Double_t sigma=1);
   virtual  UInt_t   GetSeed() {return fSeed;}
   virtual  UInt_t   Integer(UInt_t imax);
   virtual  Double_t Landau(Double_t mean=0, Double_t sigma=1);
   virtual  Int_t    Poisson(Double_t mean);
   virtual  Double_t PoissonD(Double_t mean);
   virtual  void     Rannor(Float_t &a, Float_t &b);
   virtual  void     Rannor(Double_t &a, Double_t &b);
   virtual  void     ReadRandom(const char *filename);
   virtual  void     SetSeed(UInt_t seed=65539);
   virtual  Double_t Rndm(Int_t i=0);
   virtual  void     RndmArray(Int_t n, Float_t *array);
   virtual  void     RndmArray(Int_t n, Double_t *array);
   virtual  void     Sphere(Double_t &x, Double_t &y, Double_t &z, Double_t xlong);
   virtual  Double_t Uniform(Double_t x1=1);
   virtual  Double_t Uniform(Double_t x1, Double_t x2);
   virtual  void     WriteRandom(const char *filename);

   ClassDef(TRandom,1)  // Random number generators
};
```

The nonuniform generators all employ the uniform generator, so that any problems with the uniform generator affect all the nonuniform generators.

There may also be problems intrinsic to some of the individual nonuniform generators, the Poisson generator being one such case.

2 The Uniform Generator

The uniform pseudorandom number generator of the `TRandom` class uses an algorithm that is identical to CERNLIB's `RN32` generator. This is a multiplicative linear congruential generator. Such a generator takes an integer seed s_0 and produces a new seed s_1 via

$$s_1 = (as_0 + c) \bmod m$$

For the `TRandom` uniform generator (and CERNLIB's `RN32`), the choice is

$$a = 69069 \qquad c = 0 \qquad m = 2^{31}$$

The ROOT implementation is

```
 Double_t TRandom::Rndm(Int_t)
{
//  Machine independent random number generator.
//  Produces uniformly-distributed floating points between 0 and 1.
//  Identical sequence on all machines of >= 32 bits.
//  Periodicity = 10**8
//  Universal version (Fred James 1985).
//  generates a number in ]0,1]

   const Double_t kCONS = 4.6566128730774E-10;
   const Int_t kMASK24  = 2147483392;

   fSeed *= 69069;
   UInt_t jy = (fSeed&kMASK24); // Set lower 8 bits to zero to assure exact float
   if (jy) return kCONS*jy;
   return Rndm();
}
```

Note that the comment is incomplete: in addition to the low 8 bits, the high bit is also cleared before converting to double. This reduces m from 2^{32} to 2^{31}.

In this note we refer to the above algorithm as `RN32`. Unfortunately, `RN32` is of very low quality. The comments in the code mention that Fred James proposed the use of `RN32` in 1985. By 1990 he was recommending that it be abandoned:

> Old-fashioned generators like `RNDM`, `RN32`, ... should be archived and made available only upon special request for historical purposes or in situations where the user really wants a bad generator.

> The user who is not sure what he needs should not by default get a generator known to be deficient.[2]

`RN32` is deficient because it fails many randomness tests. Conceptually, randomness tests are just toy Monte Carlo simulations, employing the generator in question, whose answer, already known theoretically, is compared with the result of the simulation.

A convenient collection of tests is the TestU01 library [3]. Appendix A shows the results of applying the battery `SmallCrush` and the battery `Crush` to `RN32`. The not very demanding `SmallCrush` ran 14 different randomness tests, taking ∼40 CPU seconds in total. Of the 14 tests, 11 tests failed badly. The more demanding `Crush` ran 94 randomness tests, taking less than 1 minute per test. Of the 94 tests, 81 tests failed badly.

It is important to note that this is not a new result; it has been known for at least 15 years that `RN32` is bad. What has changed is that computers are now fast enough to see the bad behavior in less than 1 second of CPU time.

2.1 the bad seed trap

When the seed s_0 is an odd integer, s_1 will also be odd. An odd seed yields a maximum period[1] of $2^{29} \simeq 5 \times 10^8$, which is dangerously short. To preserve randomness, it is important that the user not sample more than a tiny fraction of the period. Sampling a full period guarantees that every point on a regular grid is visited exactly the same number of times—the random fluctuations disappear completely.

The default seed for `TRandom` is odd, but class `TRandom` allows the user to specify his own seed (an `unsigned int`). It turns out that the seed *must* be odd to achieve the maximum period, but `TRandom` does not protect or warn the user about this. Each factor of 2 in the seed reduces the period by a factor of 2. For example:

```
CINT/ROOT C/C++ Interpreter version 5.15.94, June 30 2003
Type ? for help. Commands must be C++ statements.
Enclose multiple statements between { }.
root [0] TRandom r(1<<28);for(int i=0;i<10;++i) cout << r.Rndm() << '\n';
0.625
0.125
0.625
0.125
0.625
```

[1] There is a minor adjustment to the period (due to the deletion of zeros) which we neglect. The period mentioned in the comments of `TRandom::Rndm` seems to be missing a factor of 5.

```
0.125
0.625
0.125
0.625
0.125
```

So using 2^{28} for the seed results in a random sequence with period=2. It is therefore vital that the user stick to odd integer seeds, or risk serious degradation in quality. Because of this danger, a call to `TRandom::SetSeed(0)`, which sets the seed to the current machine clock time, is too dangerous to ever use.

The reader is encouraged to try to predict `TRandom`'s behavior for an initial seed equal to 2^{31}.

2.2 small number of bits

Because the low 8 bits are cleared, there are only 23 bits of precision provided by `TRandom::Rndm`, despite its declaration as returning a `double`. Consequently, the smallest value returned by `TRandom::Rndm` is $2^{-23} \simeq 10^{-7}$.

When the focus is on rare occurrences, some care must therefore be taken. For example, the inverse transform method of generating random Gaussian deviates would produce no deviates beyond $\sim 5\sigma$ using `TRandom::Rndm`.

2.3 `TRandom2` and `TRandom3`

The observant reader will have noticed that the `TRandom` member functions are declared `virtual`, indicating that `TRandom` is also intended to be used as a base class. The ROOT team has provided two classes, `TRandom2` and `TRandom3`, which are derived from `TRandom`, and override the `Rndm` member function. The purpose of `TRandom2` and `TRandom3` is to provide access to generators which are not as deficient as that used by `TRandom`. Both `TRandom2` and `TRandom3` fix all the problems listed above.

The `TRandom2` generator uses the same algorithm as CERNLIB's `RANECU` generator. This is far superior to `RN32`, and although not perfect, is suitable for most use. Appendix A shows the results of applying the battery `Crush` to `RANECU`. Out of 94 tests, there are only 2 definite failures

The `TRandom3` generator uses the "Mersenne Twister" algorithm, which is among the best available. Appendix A shows the results of applying the battery `Crush` to the Mersenne Twister. Out of 94 tests, there are no definite failures at all.

As is typical on most applications, the CPU time required to run the `Crush` suite of tests is about the same for `TRandom`, `TRandom2`, and `TRandom3`—most of the CPU time is spent in the testing, not the generation. One con-

cludes that there is no reason to ever use TRandom objects. TRandom3, being faster than TRandom2, and of better quality, is the obvious first choice.

Since any use of the TRandom class, other than as a base class, risks disaster, it would be best if the ROOT team would convert the TRandom class to *pure virtual*. That is, change the line in the declaration to

```
virtual Double_t Rndm(Int_t i=0) = 0; // pure virtual
```

so that it would *only* function as a base class. For now, users should be instructed to only use TRandom3.

3 TRandom::Poisson

Class TRandom's member function TRandom::Poisson, not being overridden by TRandom2 or TRandom3, is common to the base and the two derived classes. The code for TRandom::Poisson is

```
Int_t TRandom::Poisson(Double_t mean)
{
// Generates a random integer N according to a Poisson law.
// Coded from Los Alamos report LA-5061-MS
// Prob(N) = exp(-mean)*mean^N/Factorial(N)
//
   Int_t n;
   if (mean <= 0) return 0;
     // use a gaussian approximation for large values of mean
   if (mean > 88) {
      n = Int_t(Gaus(0,1)*TMath::Sqrt(mean) + mean +0.5);
      return n;
   }
   Double_t expmean = TMath::Exp(-mean);
   Double_t pir = 1;
   n = -1;
   while(1) {
      n++;
      pir *= Rndm();
      if (pir <= expmean) break;
   }
   return n;
}
```

A Gaussian approximation to the Poisson distribution is used for mean μ greater than 88. At $\mu = 88$, this is actually a very poor approximation to a true Poisson distribution. Appendix B shows that the approximation at $\mu = 88$ is good to only $\sim 5\%$ within $\pm 2\sigma$, and beyond that it degrades very quickly. For example, at $n = 50$, 4σ below a mean of 88, the Gaussian approximation produces 3.5 times too many events.

The reason for this is that the Poisson distribution is quite asymmetric (its third moment is equal to μ), while the Gaussian is symmetric about its mean. It is not sufficient to just match the first two moments; to achieve a good approximation, matching at least the first three moments[2] seems necessary in practice. Consequently, the Gaussian approximation to the Poisson remains inaccurate even for $\mu \gg 88$.

Because of this problem, `TRandom::Poisson`, `TRandom2::Poisson`, and `TRandom3::Poisson` are all unsuitable for general use. Poisson generators that are both correct and efficient are readily available. For example, the Numerical Recipes[4] version is reliable. CLHEP's `RandPoisson`[5] also uses the Numerical Recipes algorithm. However, CERNLIB's `RNPSSN` is identical to `TRandom::Poisson`, and must also be rejected.

It is important that any generator provided for general use not make unwarranted assumptions about the user's requirements. If the user is sure that the Gaussian approximation is sufficient for his application, he is free to call the Gaussian generator in his own code. If he is not sure, he should not get the Gaussian approximation anyway by default.

4 Recommendations

- Do *not* use ROOT class `TRandom`.
- Use the ROOT class `TRandom3` instead. `TRandom2` is also usable, but not quite as good as `TRandom3`.
- Do *not* use ROOT's `TRandom::Poisson`, `TRandom::PoissonD`, `TRandom2::Poisson`, `TRandom2::PoissonD`, `TRandom3::Poisson`, `TRandom3::PoissonD`.
- Do *not* use CERNLIB's Poisson generator `RNPSSN`.
- Use the Numerical Recipes Poisson generator[4], or CLHEP's `RandPoisson`[5] instead.

Appendix A: TestU01 Results

RN32—`TRandom`

The summary log from the battery SmallCrush[3], a suite of not very stringent statistical tests. Out of 14 tests, only 3 passed. The results of the 11 tests that failed are listed below.

[2]The classic Pearson scheme requires matching the first four moments.

```
============== Summary results of SmallCrush ==============

 Generator:        RN32
 Number of tests:  14
 Total CPU time:   00:00:36.95
 The following tests gave p-values outside [0.01, 0.99]:
 (eps  means a value < 1.0e-300):
 (eps1 means a value < 1.0e-15):

       Test                          p-value
 ----------------------------------------------
  1  BirthdaySpacings                  eps
  2  Collision                       1 - eps1
  3  Gap                              1.2e-6
  6  MaxOft                            eps
  8  MatrixRank                        eps
  9  HammingIndep                      eps
 10  RandomWalk1 H                     eps
 11  RandomWalk1 M                     eps
 12  RandomWalk1 J                     eps
 13  RandomWalk1 R                     eps
 14  RandomWalk1 C                     eps
 ----------------------------------------------
 All other tests were passed
```

The summary log from the battery Crush[3], a suite of stringent statistical tests. Out of 94 tests, only 11 definitely passed. The results of the remaining 83 tests are listed below. Of those, 2 tests (numbers 37 and 43) are borderline, and 81 tests are definite failures.

```
============== Summary results of Crush ==============

 Generator:        RN32
 Number of tests:  94
 Total CPU time:   01:14:31.60
 The following tests gave p-values outside [0.01, 0.99]:
 (eps  means a value < 1.0e-300):
 (eps1 means a value < 1.0e-15):

       Test                          p-value
 ----------------------------------------------
  1  SerialOver (t = 2)              1 - eps1
  3  Collision (t = 1)               1 - eps1
  4  CollisionOver (t = 2)           1 - eps1
  5  MultinomialBitsOver             1 - eps1
  6  MultinomialBitsOver               eps
  7  MultinomialBitsOver             1 -  2.3e-9
  8  MultinomialBitsOver               eps
  9  MultinomialBitsOver               eps
 10  BirthdaySpacings (t = 2)          eps
 11  BirthdaySpacings (t = 3)          eps
 12  BirthdaySpacings (t = 4)          eps
 13  BirthdaySpacings (t = 7)          eps
 14  BirthdaySpacings (t = 7)          eps
 15  BirthdaySpacings (t = 8)          eps
 16  BirthdaySpacings (t = 8)          eps
 17  ClosePairs NP (t = 2)             eps
 18  ClosePairs mNP (t = 2)            eps
 19  ClosePairs NP (t = 4)          1.9e-165
 20  ClosePairs mNP (t = 4)         1.9e-165
 21  ClosePairs mNP1 (t = 4)           eps
```

```
22  ClosePairs mNP2 (t = 4)        1 - eps1
23  ClosePairsBitMatch (t = 2)     1 - eps1
24  ClosePairsBitMatch (t = 4)     1 - eps1
25  SimpPoker (d = 16)              6.1e-5
26  SimpPoker (d = 16)                eps
27  SimpPoker (d = 64)             2.2e-16
28  SimpPoker (d = 64)                eps
30  CouponCollector                   eps
31  CouponCollector                 1.2e-5
32  CouponCollector                   eps
34  Gap                               eps
35  Gap                               eps
36  Gap                               eps
37  Run of U01                     0.9968
38  Permutation                    1 - eps1
40  MaxOft (t = 5)                1.9e-165
41  MaxOft (t = 20)                   eps
43  SampleMean                     0.9996
45  AppearanceSpacings                eps
47  WeightDistrib (r = 8)             eps
48  WeightDistrib (r = 16)            eps
49  WeightDistrib (r = 27)            eps
51  MatrixRank (30 x 30)              eps
52  MatrixRank (300 x 300)            eps
53  MatrixRank (1200 x 1200)          eps
54  MatrixRank (30 x 30)              eps
55  MatrixRank (300 x 300)            eps
56  MatrixRank (1200 x 1200)          eps
57  GCD                               eps
58  RandomWalk1 H (L = 90)            eps
59  RandomWalk1 M (L = 90)            eps
60  RandomWalk1 J (L = 90)            eps
61  RandomWalk1 R (L = 90)            eps
62  RandomWalk1 C (L = 90)            eps
63  RandomWalk1 H (L = 1000)          eps
64  RandomWalk1 M (L = 1000)          eps
65  RandomWalk1 J (L = 1000)          eps
66  RandomWalk1 R (L = 1000)          eps
67  RandomWalk1 C (L = 1000)          eps
68  RandomWalk1 H (L = 10000)         eps
69  RandomWalk1 M (L = 10000)         eps
70  RandomWalk1 J (L = 10000)         eps
71  RandomWalk1 R (L = 10000)         eps
72  RandomWalk1 C (L = 10000)         eps
74  LempelZiv                     1.9e-165
75  Fourier1                          eps
76  Fourier3                          eps
77  LongestHeadRun                    eps
78  PeriodsInStrings               1 - 1.1e-15
79  HammingWeight2                    eps
80  HammingCorr (L = 30)              eps
81  HammingCorr (L = 300)             eps
82  HammingCorr (L = 1200)            eps
83  HammingIndep (L = 30)             eps
84  HammingIndep (L = 30)             eps
85  HammingIndep (L = 300)            eps
86  HammingIndep (L = 300)            eps
87  HammingIndep (L = 1200)           eps
88  HammingIndep (L = 1200)           eps
89  Run of bits                       eps
90  Run of bits                       eps
93  AutoCor (d = 30)              1.9e-165
94  AutoCor (d = 10)              1.9e-165
```

```
 ------------------------------------------------
 All other tests were passed
```

RANECU—TRandom2

The summary log from the battery Crush[3], a suite of stringent statistical tests. Out of 94 tests, only 2 definitely failed. The p-value for test number 18 below is probably just a statistical fluctuation.

```
============== Summary results of Crush ==============

 Generator:        RANECU
 Number of tests:  94
 Total CPU time:   01:37:01.10
 The following tests gave p-values outside [0.01, 0.99]:
 (eps  means a value < 1.0e-300):
 (eps1 means a value < 1.0e-15):

       Test                          p-value
 ------------------------------------------------
 12  BirthdaySpacings (t = 4)       5.7e-23
 18  ClosePairs mNP (t = 2)          2.7e-3
 59  RandomWalk1 M (L = 90)         1 -  6.3e-5
 ------------------------------------------------
 All other tests were passed
```

Mersenne Twister—TRandom3

The summary log from the battery Crush[3], a suite of stringent statistical tests. Out of 94 tests, there are no definite failures. The marginal one listed below is probably just a statistical fluctuation.

```
============== Summary results of Crush ==============

 Generator:        Mersenne Twister
 Number of tests:  94
 Total CPU time:   01:23:50.31
 The following tests gave p-values outside [0.01, 0.99]:
 (eps  means a value < 1.0e-300):
 (eps1 means a value < 1.0e-15):

       Test                          p-value
 ------------------------------------------------
 29  CouponCollector                 0.9998
 ------------------------------------------------
 All other tests were passed
```

Appendix B: Gaussian vs Poisson

We compare a true Poisson to the Gaussian approximation used by `TRandom::Poisson`, for the case $\mu = 88$. The Poisson probability of obtaining n, given mean μ, is

$e^{-\mu}\mu^n/n!$, tabulated in the third column. The fourth column is the probability of obtaining n from `TRandom::Poisson` using the Gaussian approximation to that probability, given by

$$\frac{1}{\sqrt{2\pi\mu}} \int_{n-1/2}^{n+1/2} e^{-0.5(x-\mu)^2/\mu} dx$$

The fifth column is the ratio of the two probabilities at a given n (Gaussian approximation over Poisson), and is a measure of the quality of the Gaussian approximation. Beyond $\sim 2\sigma$, the quality of the Gaussian approximation used by `TRandom::Poisson` quickly degrades.

mu=88

n	(n-mu)/sig	Poisson	Gaussian	G/P	n	(n-mu)/sig	Poisson	Gaussian	G/P
0	-9.381	6.055e-39	3.447e-21	5.693e+17	88	0.000	4.249e-02	4.251e-02	1.000e+00
1	-9.274	5.328e-37	9.308e-21	1.747e+16	89	0.107	4.201e-02	4.227e-02	1.006e+00
2	-9.168	2.344e-35	2.485e-20	1.060e+15	90	0.213	4.108e-02	4.155e-02	1.012e+00
3	-9.061	6.877e-34	6.560e-20	9.540e+13	91	0.320	3.972e-02	4.039e-02	1.017e+00
4	-8.954	1.513e-32	1.712e-19	1.132e+13	92	0.426	3.799e-02	3.882e-02	1.022e+00
5	-8.848	2.663e-31	4.418e-19	1.659e+12	93	0.533	3.595e-02	3.688e-02	1.026e+00
6	-8.741	3.905e-30	1.127e-18	2.887e+11	94	0.640	3.366e-02	3.465e-02	1.030e+00
7	-8.635	4.909e-29	2.844e-18	5.792e+10	95	0.746	3.118e-02	3.219e-02	1.032e+00
8	-8.528	5.400e-28	7.092e-18	1.313e+10	96	0.853	2.858e-02	2.956e-02	1.034e+00
9	-8.421	5.280e-27	1.749e-17	3.312e+09	97	0.959	2.593e-02	2.684e-02	1.035e+00
10	-8.315	4.647e-26	4.264e-17	9.176e+08	98	1.066	2.328e-02	2.410e-02	1.035e+00
11	-8.208	3.717e-25	1.028e-16	2.765e+08	99	1.173	2.070e-02	2.139e-02	1.033e+00
12	-8.102	2.726e-24	2.450e-16	8.986e+07	100	1.279	1.821e-02	1.877e-02	1.031e+00
13	-7.995	1.845e-23	5.773e-16	3.128e+07	101	1.386	1.587e-02	1.629e-02	1.026e+00
14	-7.888	1.160e-22	1.345e-15	1.159e+07	102	1.492	1.369e-02	1.397e-02	1.021e+00
15	-7.782	6.805e-22	3.098e-15	4.553e+06	103	1.599	1.170e-02	1.185e-02	1.013e+00
16	-7.675	3.743e-21	7.056e-15	1.885e+06	104	1.706	9.897e-03	9.939e-03	1.004e+00
17	-7.569	1.937e-20	1.589e-14	8.201e+05	105	1.812	8.294e-03	8.242e-03	9.936e-01
18	-7.462	9.472e-20	3.537e-14	3.735e+05	106	1.919	6.886e-03	6.757e-03	9.812e-01
19	-7.355	4.387e-19	7.787e-14	1.775e+05	107	2.025	5.663e-03	5.477e-03	9.671e-01
20	-7.249	1.930e-18	1.695e-13	8.780e+04	108	2.132	4.614e-03	4.389e-03	9.511e-01
21	-7.142	8.089e-18	3.647e-13	4.509e+04	109	2.239	3.725e-03	3.478e-03	9.335e-01
22	-7.036	3.235e-17	7.759e-13	2.398e+04	110	2.345	2.980e-03	2.724e-03	9.142e-01
23	-6.929	1.238e-16	1.632e-12	1.318e+04	111	2.452	2.363e-03	2.110e-03	8.932e-01
24	-6.822	4.539e-16	3.394e-12	7.478e+03	112	2.558	1.856e-03	1.616e-03	8.706e-01
25	-6.716	1.598e-15	6.980e-12	4.369e+03	113	2.665	1.446e-03	1.224e-03	8.464e-01
26	-6.609	5.408e-15	1.419e-11	2.624e+03	114	2.772	1.116e-03	9.161e-04	8.209e-01
27	-6.503	1.763e-14	2.853e-11	1.618e+03	115	2.878	8.540e-04	6.781e-04	7.940e-01
28	-6.396	5.539e-14	5.669e-11	1.023e+03	116	2.985	6.479e-04	4.963e-04	7.660e-01
29	-6.289	1.681e-13	1.114e-10	6.628e+02	117	3.091	4.873e-04	3.591e-04	7.369e-01
30	-6.183	4.931e-13	2.164e-10	4.390e+02	118	3.198	3.634e-04	2.569e-04	7.069e-01
31	-6.076	1.400e-12	4.158e-10	2.970e+02	119	3.305	2.687e-04	1.817e-04	6.762e-01
32	-5.970	3.849e-12	7.896e-10	2.051e+02	120	3.411	1.971e-04	1.271e-04	6.448e-01
33	-5.863	1.026e-11	1.483e-09	1.445e+02	121	3.518	1.433e-04	8.786e-05	6.131e-01
34	-5.756	2.657e-11	2.753e-09	1.036e+02	122	3.624	1.034e-04	6.007e-05	5.810e-01
35	-5.650	6.680e-11	5.053e-09	7.565e+01	123	3.731	7.396e-05	4.060e-05	5.489e-01
36	-5.543	1.633e-10	9.171e-09	5.617e+01	124	3.838	5.249e-05	2.713e-05	5.169e-01
37	-5.437	3.883e-10	1.646e-08	4.238e+01	125	3.944	3.695e-05	1.793e-05	4.852e-01
38	-5.330	8.993e-10	2.920e-08	3.247e+01	126	4.051	2.581e-05	1.171e-05	4.538e-01
39	-5.223	2.029e-09	5.121e-08	2.524e+01	127	4.157	1.788e-05	7.565e-06	4.230e-01
40	-5.117	4.464e-09	8.882e-08	1.990e+01	128	4.264	1.229e-05	4.831e-06	3.930e-01
41	-5.010	9.582e-09	1.523e-07	1.590e+01	129	4.371	8.387e-06	3.051e-06	3.637e-01
42	-4.904	2.008e-08	2.582e-07	1.286e+01	130	4.477	5.677e-06	1.904e-06	3.354e-01

43	-4.797	4.109e-08	4.328e-07	1.054e+01	131	4.584	3.814e-06	1.175e-06	3.082e-01
44	-4.690	8.217e-08	7.174e-07	8.730e+00	132	4.690	2.542e-06	7.174e-07	2.821e-01
45	-4.584	1.607e-07	1.175e-06	7.315e+00	133	4.797	1.682e-06	4.328e-07	2.573e-01
46	-4.477	3.074e-07	1.904e-06	6.195e+00	134	4.904	1.105e-06	2.582e-07	2.337e-01
47	-4.371	5.756e-07	3.051e-06	5.300e+00	135	5.010	7.201e-07	1.523e-07	2.115e-01
48	-4.264	1.055e-06	4.831e-06	4.578e+00	136	5.117	4.660e-07	8.882e-08	1.906e-01
49	-4.157	1.895e-06	7.565e-06	3.992e+00	137	5.223	2.993e-07	5.121e-08	1.711e-01
50	-4.051	3.335e-06	1.171e-05	3.512e+00	138	5.330	1.909e-07	2.920e-08	1.530e-01
51	-3.944	5.755e-06	1.793e-05	3.115e+00	139	5.437	1.208e-07	1.646e-08	1.362e-01
52	-3.838	9.739e-06	2.713e-05	2.786e+00	140	5.543	7.595e-08	9.171e-09	1.207e-01
53	-3.731	1.617e-05	4.060e-05	2.511e+00	141	5.650	4.740e-08	5.053e-09	1.066e-01
54	-3.624	2.635e-05	6.007e-05	2.279e+00	142	5.756	2.938e-08	2.753e-09	9.371e-02
55	-3.518	4.216e-05	8.786e-05	2.084e+00	143	5.863	1.808e-08	1.483e-09	8.202e-02
56	-3.411	6.626e-05	1.271e-04	1.918e+00	144	5.970	1.105e-08	7.896e-10	7.147e-02
57	-3.305	1.023e-04	1.817e-04	1.776e+00	145	6.076	6.705e-09	4.158e-10	6.201e-02
58	-3.198	1.552e-04	2.569e-04	1.655e+00	146	6.183	4.041e-09	2.164e-10	5.356e-02
59	-3.091	2.315e-04	3.591e-04	1.551e+00	147	6.289	2.419e-09	1.114e-10	4.605e-02
60	-2.985	3.395e-04	4.963e-04	1.462e+00	148	6.396	1.438e-09	5.669e-11	3.941e-02
61	-2.878	4.898e-04	6.781e-04	1.384e+00	149	6.503	8.496e-10	2.853e-11	3.358e-02
62	-2.772	6.952e-04	9.161e-04	1.318e+00	150	6.609	4.984e-10	1.419e-11	2.847e-02
63	-2.665	9.711e-04	1.224e-03	1.260e+00	151	6.716	2.905e-10	6.980e-12	2.403e-02
64	-2.558	1.335e-03	1.616e-03	1.210e+00	152	6.822	1.682e-10	3.394e-12	2.019e-02
65	-2.452	1.808e-03	2.110e-03	1.167e+00	153	6.929	9.672e-11	1.632e-12	1.687e-02
66	-2.345	2.410e-03	2.724e-03	1.130e+00	154	7.036	5.527e-11	7.759e-13	1.404e-02
67	-2.239	3.166e-03	3.478e-03	1.099e+00	155	7.142	3.138e-11	3.647e-13	1.162e-02
68	-2.132	4.097e-03	4.389e-03	1.071e+00	156	7.249	1.770e-11	1.695e-13	9.574e-03
69	-2.025	5.225e-03	5.477e-03	1.048e+00	157	7.355	9.922e-12	7.787e-14	7.848e-03
70	-1.919	6.569e-03	6.757e-03	1.029e+00	158	7.462	5.526e-12	3.537e-14	6.402e-03
71	-1.812	8.141e-03	8.242e-03	1.012e+00	159	7.569	3.058e-12	1.589e-14	5.195e-03
72	-1.706	9.950e-03	9.939e-03	9.989e-01	160	7.675	1.682e-12	7.056e-15	4.195e-03
73	-1.599	1.200e-02	1.185e-02	9.881e-01	161	7.782	9.194e-13	3.098e-15	3.370e-03
74	-1.492	1.426e-02	1.397e-02	9.795e-01	162	7.888	4.994e-13	1.345e-15	2.693e-03
75	-1.386	1.674e-02	1.629e-02	9.731e-01	163	7.995	2.696e-13	5.773e-16	2.141e-03
76	-1.279	1.938e-02	1.877e-02	9.686e-01	164	8.102	1.447e-13	2.450e-16	1.693e-03
77	-1.173	2.215e-02	2.139e-02	9.657e-01	165	8.208	7.716e-14	1.028e-16	1.332e-03
78	-1.066	2.499e-02	2.410e-02	9.643e-01	166	8.315	4.091e-14	4.264e-17	1.042e-03
79	-0.959	2.783e-02	2.684e-02	9.643e-01	167	8.421	2.156e-14	1.749e-17	8.114e-04
80	-0.853	3.062e-02	2.956e-02	9.654e-01	168	8.528	1.129e-14	7.092e-18	6.281e-04
81	-0.746	3.326e-02	3.219e-02	9.676e-01	169	8.635	5.879e-15	2.844e-18	4.837e-04
82	-0.640	3.570e-02	3.465e-02	9.707e-01	170	8.741	3.043e-15	1.127e-18	3.704e-04
83	-0.533	3.785e-02	3.688e-02	9.745e-01	171	8.848	1.566e-15	4.418e-19	2.821e-04
84	-0.426	3.965e-02	3.882e-02	9.790e-01	172	8.954	8.013e-16	1.712e-19	2.137e-04
85	-0.320	4.105e-02	4.039e-02	9.839e-01	173	9.061	4.076e-16	6.560e-20	1.610e-04
86	-0.213	4.200e-02	4.155e-02	9.893e-01	174	9.168	2.061e-16	2.485e-20	1.206e-04
87	-0.107	4.249e-02	4.227e-02	9.948e-01	175	9.274	1.037e-16	9.308e-21	8.980e-05

References

[1] `root.cern.ch/root/html/TRandom.html`

[2] Fred James, "A review of pseudorandom number generators", Computer Physics Communications, **60** (1990) page 329.

[3] Pierre L'Ecuyer and Richard Simard, "TestU01, A Software Library in ANSI C for Empirical Testing of Random Number Generators", `www.iro.umontreal.ca/~simardr/testu01/tu01.html`

[4] William H. Press *et al.*, "Numerical Recipes", 2nd edition, (Cambridge University Press, Cambridge, 1992), §7.3
lib-www.lanl.gov/numerical/bookcpdf/c7-3.pdf

[5] CLHEP—A Class Library for High Energy Physics
wwwasd.web.cern.ch/wwwasd/lhc++/clhep/
wwwasd.web.cern.ch/wwwasd/lhc++/clhep/manual/RefGuide/Random/RandPoisson.html