Random Variate Generation and Testing in Interactive Environments

Josef Leydold and Wolfgang Hörmann

Department of Statistics and Mathematics, WU Wien, Austria
Boğaziçi University, Istanbul, Turkey

8th MCQMC 2008
Montréal, July 11, 2008

Overview

Interactive environments help you to

- detect coding errors in an implementation of a random variate generator;
- detect serious bug in a design of an algorithm;
- allow fast prototyping of simulation models.

The R programming language for statistical computing is well suited for this purpose.

Development of Algorithms

1. Design an algorithms using mathematical principles.
2. Theoretical Proof of correctness.
3. Implement proof-of-concept using assembler / **C** / Fortran.
4. Run an empirical test to detect programming errors.
5. Estimate memory consumption and generation times.
6. Maybe the algorithms is coded and used in a production environment.
7. Practitioners use the algorithm in their simulation using a **programming library** or some fancy **GUI** to build up a model and run simulations.

UNU.RAN

We have developed and implemented several (automatic) algorithm in a library called

UNU.RAN (**U**niversal **N**on**U**niform **Ran**dom variate generators),

available at `http://statmath.wu-wien.ac.at/unuran`.

Example: Maxwell Inflow Distribution

```
double lpdf (double x) { log(1-x)-x*x; }
double dlpdf (double x) { log(1-x)-x*x; }

int main(void) {
  UNUR_DISTR *distr;  /* distribution object */
  UNUR_PAR   *par;    /* parameter object    */
  UNUR_GEN   *gen;    /* generator object    */

  /* Create distribution object with required data */
  distr = unur_distr_cont_new();
  unur_distr_cont_set_logpdf(distr,  lpdf);
  unur_distr_cont_set_dlogpdf(distr, dlpdf);
  unur_distr_cont_set_domain(distr, -UNUR_INFINITY, 1);

  /* Choose a method: AROU */
  par = unur_arou_new(distr);

  /* Create the generator object */
  gen = unur_init(par);

  /* Sample */
  x = unur_sample(gen);
}
```

UNU.RAN String Interface

```
int main(void) {
  UNUR_GEN   *gen;    /* generator object    */

  /* Create the generator object */
  gen = unur_str2gen(
         "distr = cont; logpdf='log(1-x)-x*x'; domain=(-inf,1) & \
          method=arou");

  /* Sample */
  x = unur_sample(gen);
}
```

Interactive Environments

In opposition to this library / GUI approach interactive environments allow for

- fast **prototyping** of algorithms;
- **playing** around with the generator and look whether it behaves like a source of (non-uniform) randomness;

In addition they provide a **powerful language** to analyse random sequences.

Examples are **R**/S, **Root** , matlab, Mathematica, ...

The R Project for Statistical Computing

The **R Project** for Statistical Computing is well-suited for this purpose:

- many statistical tools
- can be easily extended
- large community provides add-on packages and support (more than 1000 packages on CRAN)
- open source (i.e. you can learn from others)
- comprehensive documentation (although a local Guru is helpful when you are new)
- `http://www.R-project.org`

An R Session

```
R version 2.7.0 (2008-04-22)
Copyright (C) 2008 The R Foundation for Statistical Computing
ISBN 3-900051-07-0

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

Built-in Gaussian Generator

- Create a Gaussian sample of size 1000

```
> X <- rnorm(n=1000)
```

- Show sample

```
> X[1:4]
```

```
[1] -0.08014195 -1.47207477 -1.72954408  0.00723830
```

- Transform into uniforms

```
> U <- pnorm(X)
> U[1:4]
```

```
[1] 0.46806218 0.07050034 0.04185588 0.50288764
```

- Get some help

```
> ?rnorm
```

Testing Gaussian Generator

- Run χ^2 goodness-of-fit test

```
> h <- hist(pnorm(X),plot=FALSE)
> chisq.test(h$counts)

Chi-squared test for given probabilities

data:  h$counts
X-squared = 5.26, df = 9, p-value = 0.811
```

- Run Kolmogorov-Smirnov test

```
> ks.test(X,pnorm)

One-sample Kolmogorov-Smirnov test

data:  X
D = 0.0329, p-value = 0.2287
alternative hypothesis: two-sided
```

Testing Gaussian Generator

- Run Shapiro-Wilk test of normality

```
> shapiro.test(X)

Shapiro-Wilk normality test

data:  X
W = 0.9985, p-value = 0.5854
```

- Run Anderson-Darling sample test

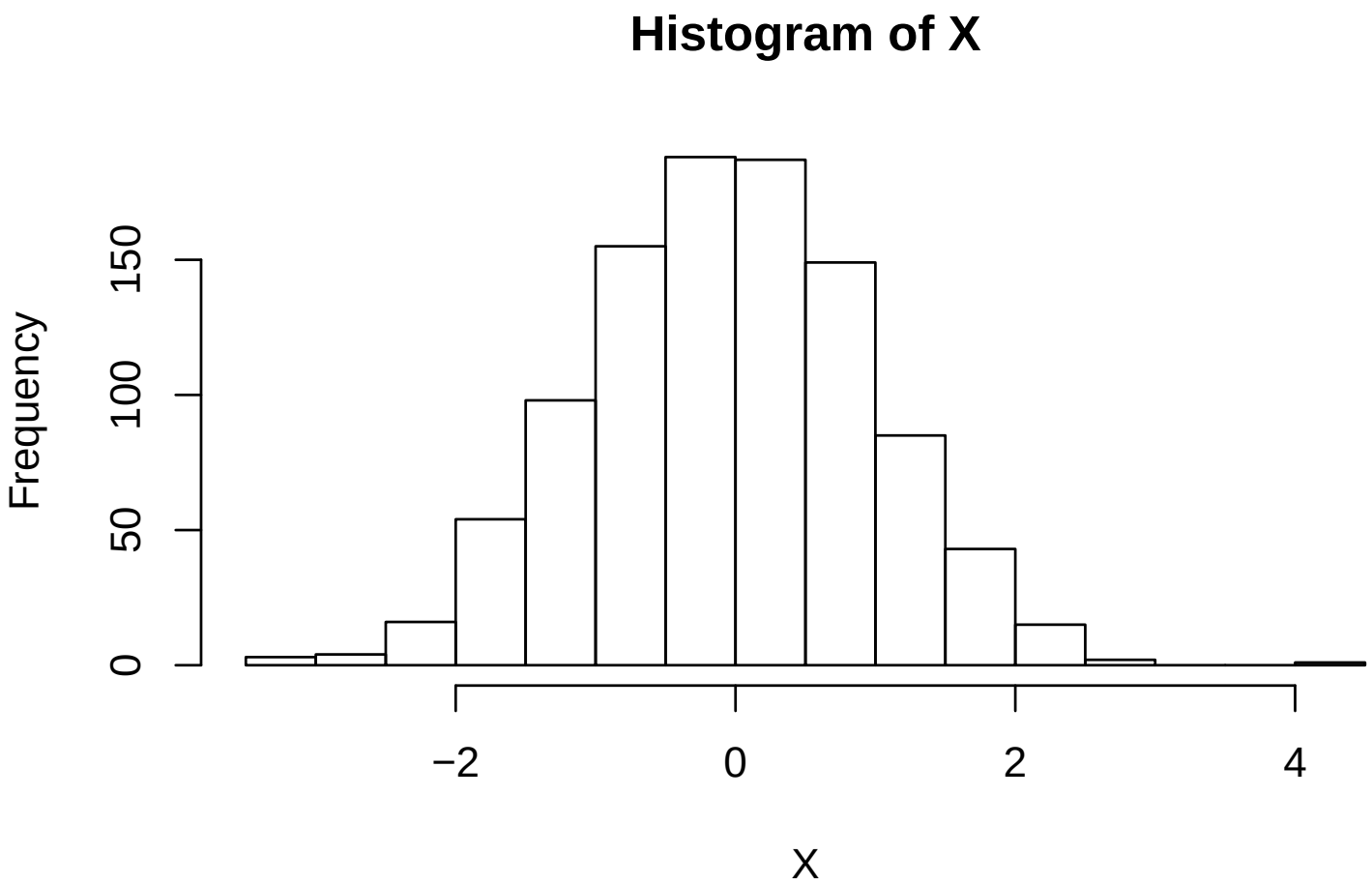
```
> library("adk")
> adk.test(pnorm(X), (1:1000-0.5)/1000)$adk

                     t.obs P-value extrapolation
not adj. for ties -0.34735 0.43139             1
adj. for ties     -0.34847 0.43173             1
```

Histogram

- Create histogram

```
> hist(X,breaks=25)
```



Fancy Histogram

- Create fancy histogram

```
> h <- hist(X, breaks=25, freq=FALSE,
+           border="dark green", col="light green",
+           xlab=NULL, ylab=NULL,
+           main="Distribution of random sample")
> xval <- seq(h$breaks[1],h$breaks[length(h$breaks)],
+          length.out=100)
> lines(xval, dnorm(xval), col="blue", lwd=2)
```

Fancy Histogram

- Create fancy histogram

```
> h <- hist(X, breaks=25, freq=FALSE,
+           border="dark green", col="light green",
+           xlab=NULL, ylab=NULL,
+           main="Distribution of random sample")
> xval <- seq(h$breaks[1],h$breaks[length(h$breaks)],
+          length.out=100)
> lines(xval, dnorm(xval), col="blue", lwd=2)
```

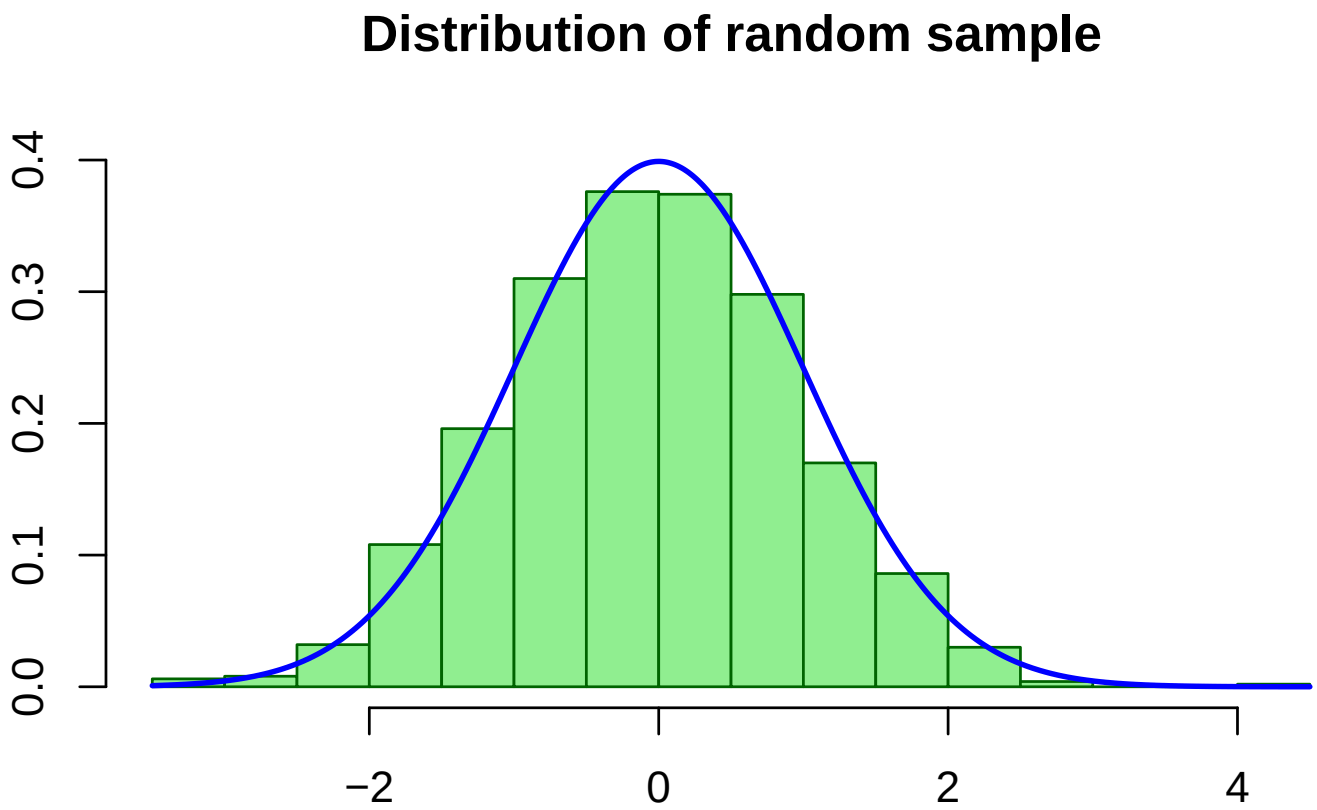
- Define function

```
> myhist <- function(x,pdf) {
+    h <- hist(x,breaks=25,freq=FALSE,border="dark green",
+              col="light green",xlab=NULL,ylab=NULL,
+              main="Distribution of random sample")
+    xval <- seq(h$breaks[1],h$breaks[length(h$breaks)],leng
+    lines(xval,pdf(xval),col="blue",lwd=2)
+ }
```

Fancy Histogram

- Create histogram

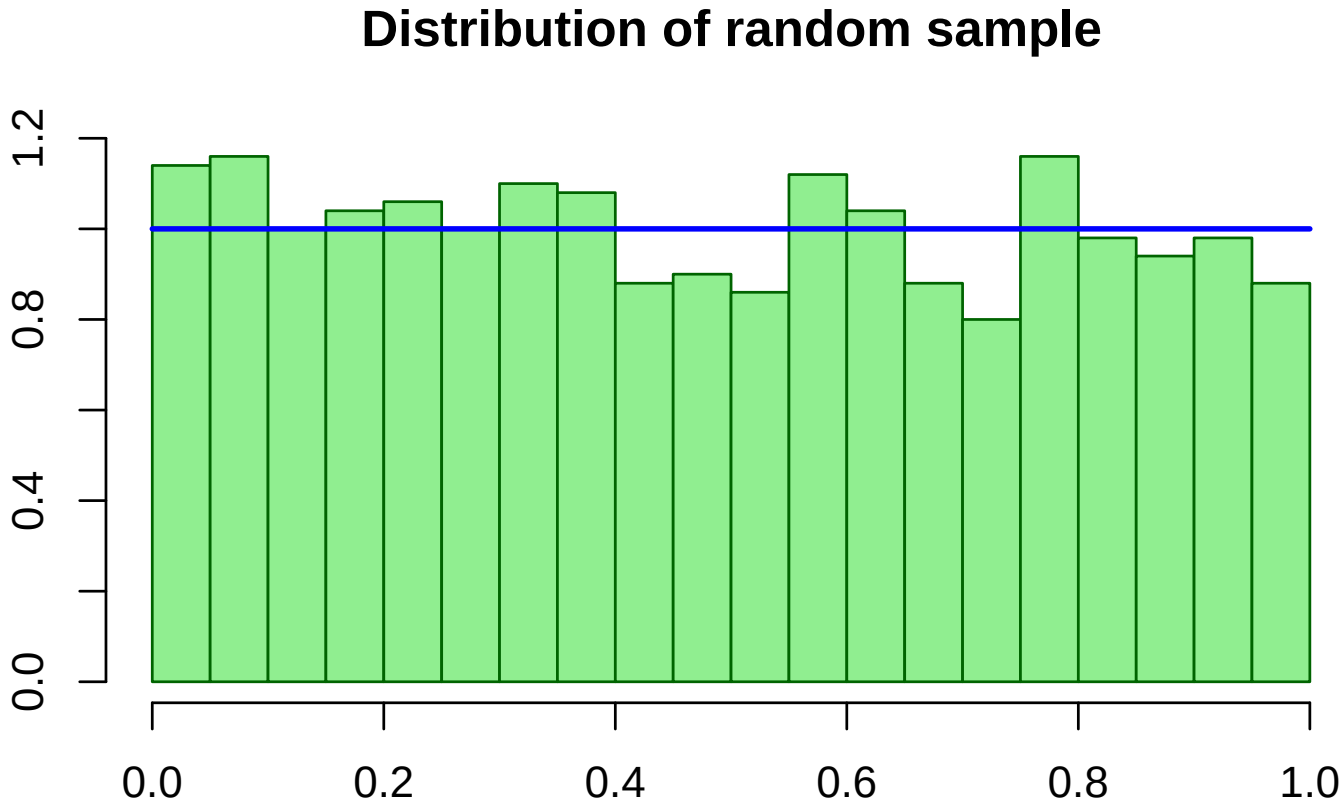
```
> myhist(X, dnorm)
```



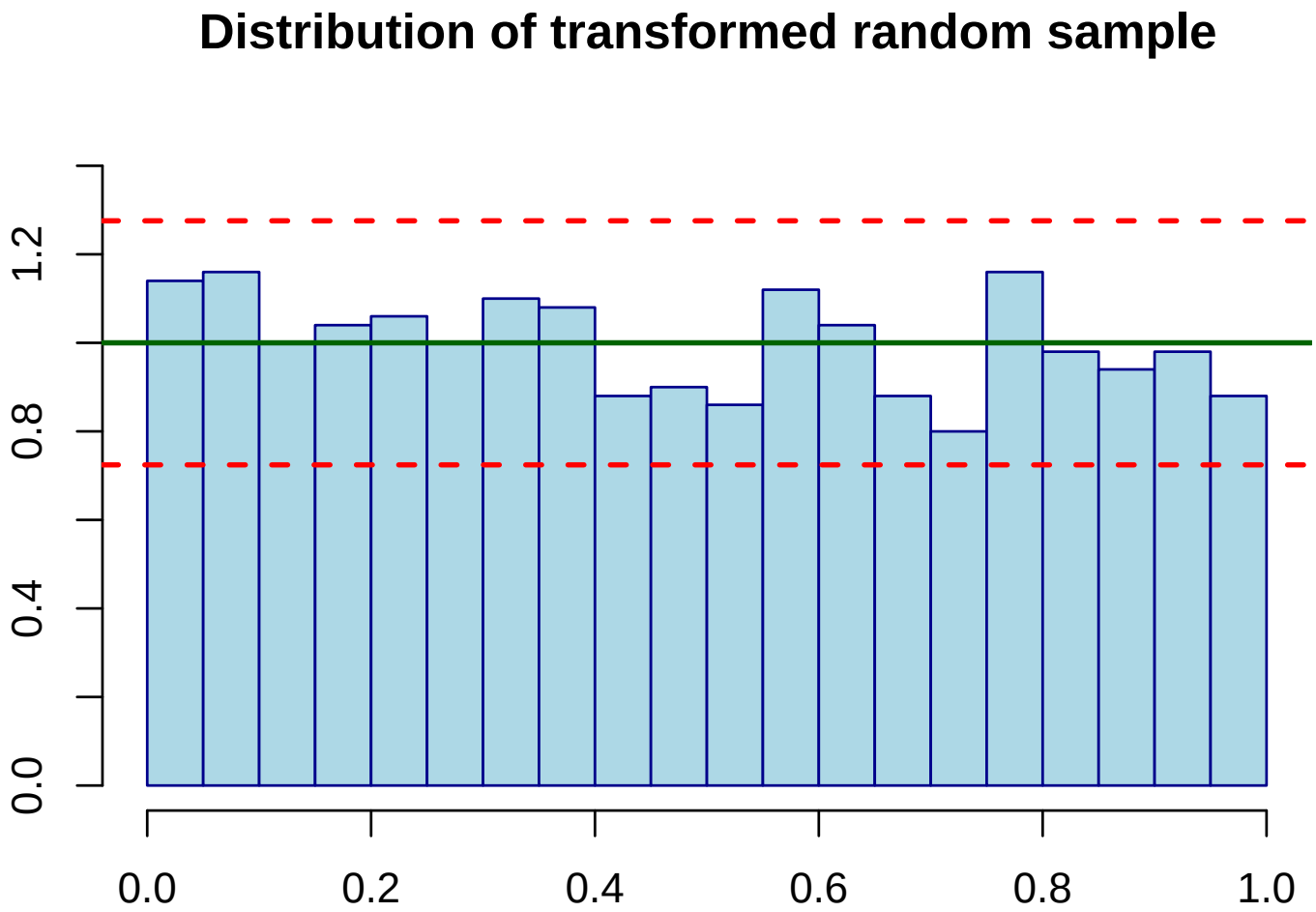
Transform to Uniforms

- Create histogram

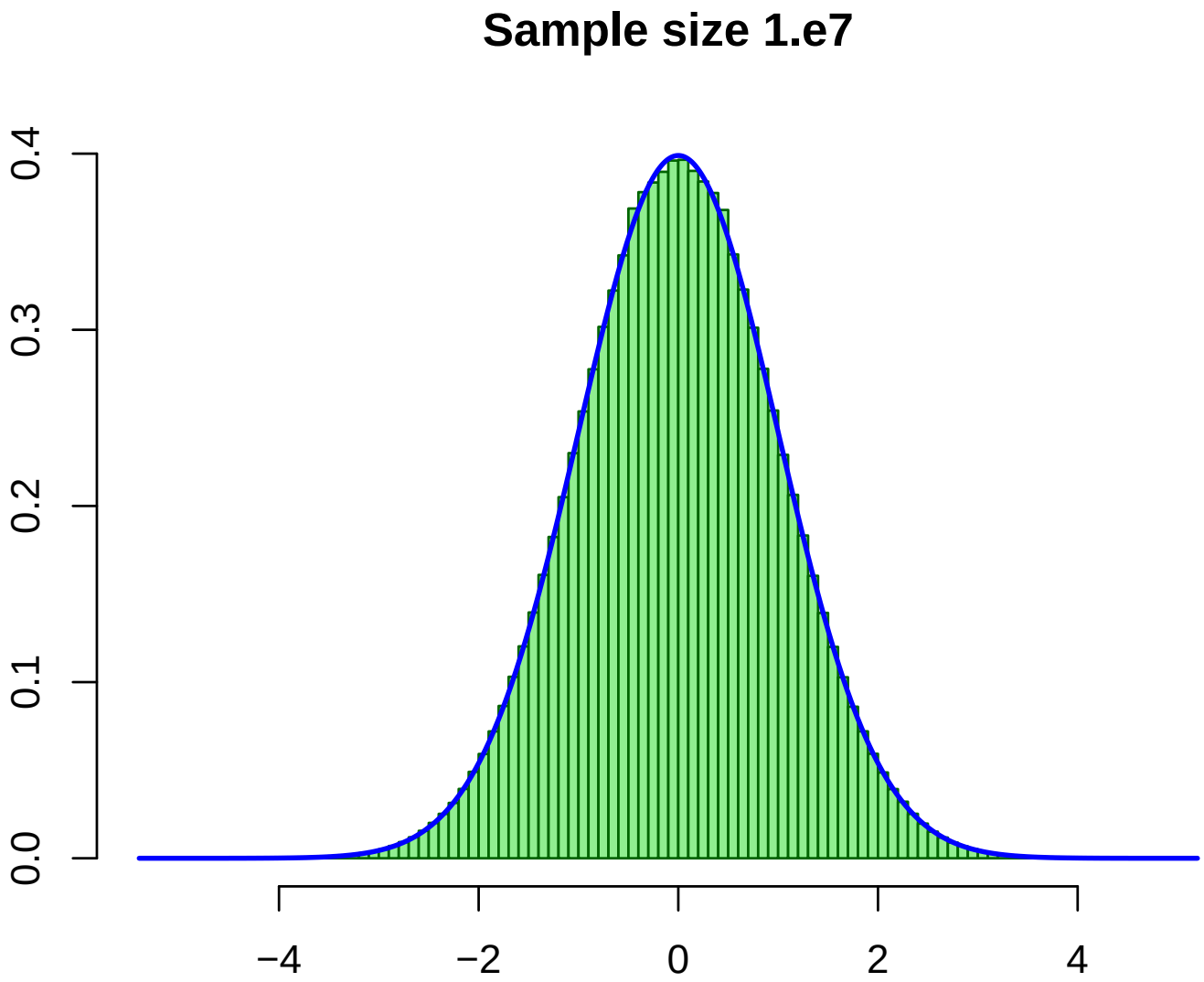
```
> myhist(pnorm(X), dunif)
```



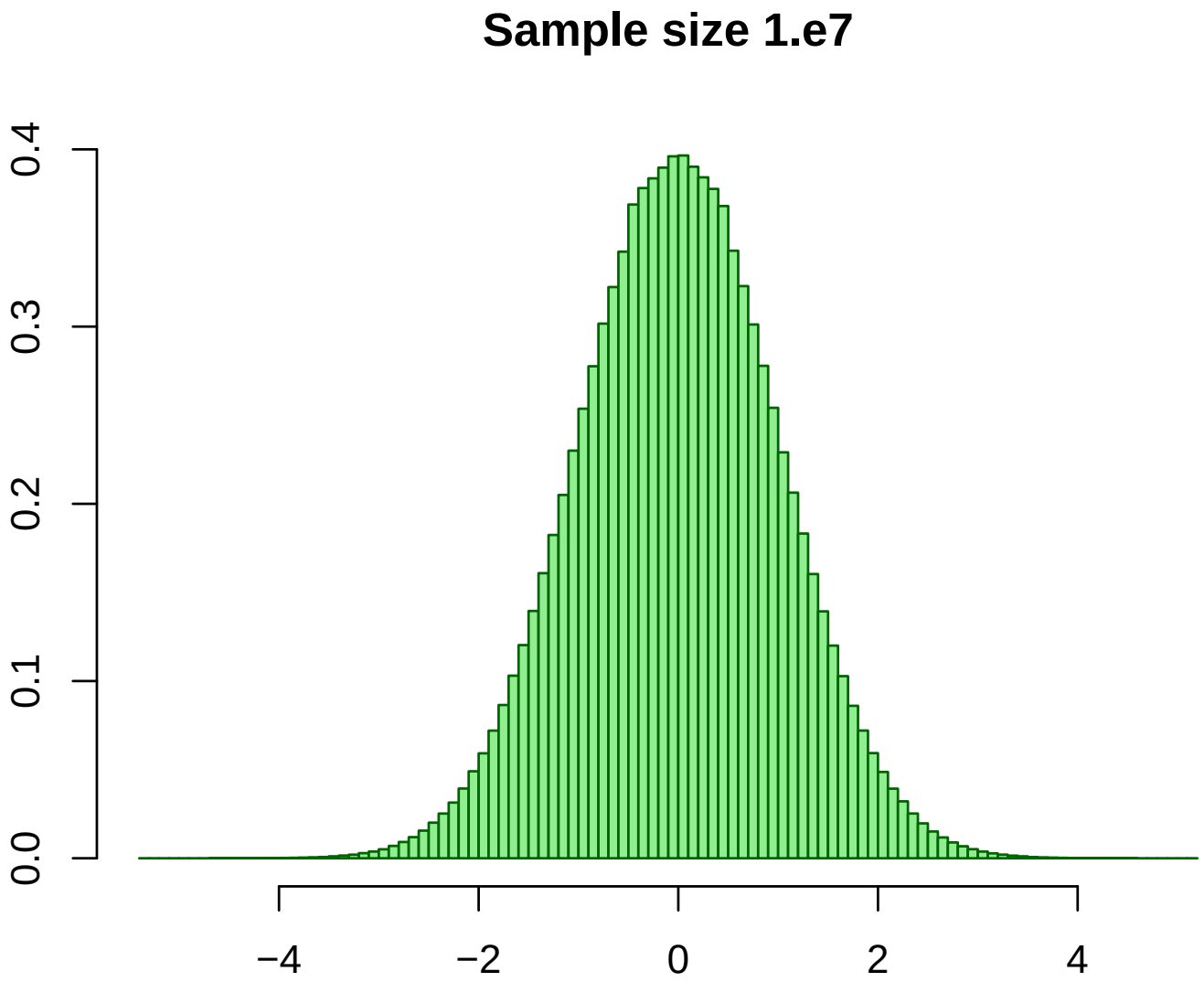
Transform to Uniforms



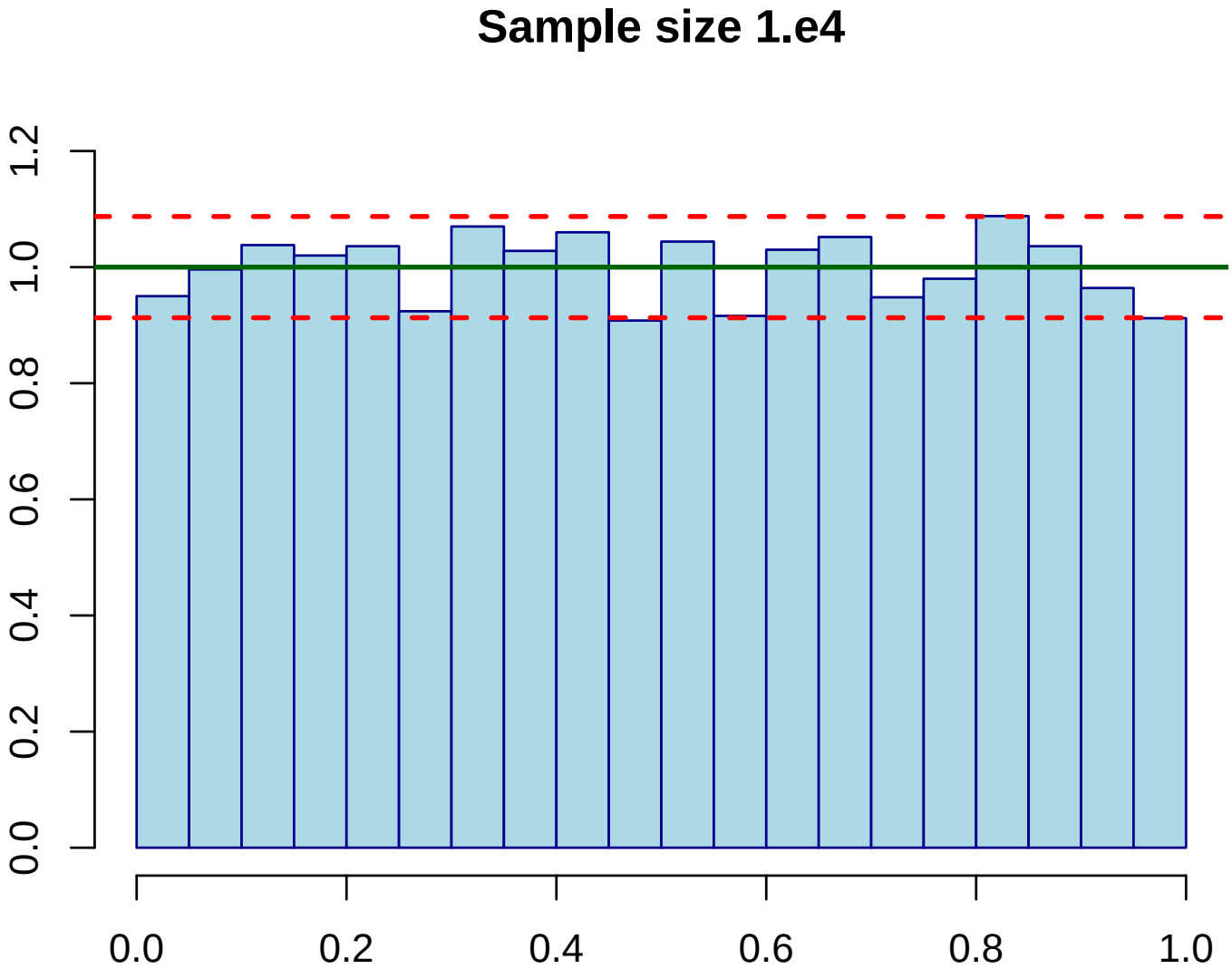
Increase Sample Size



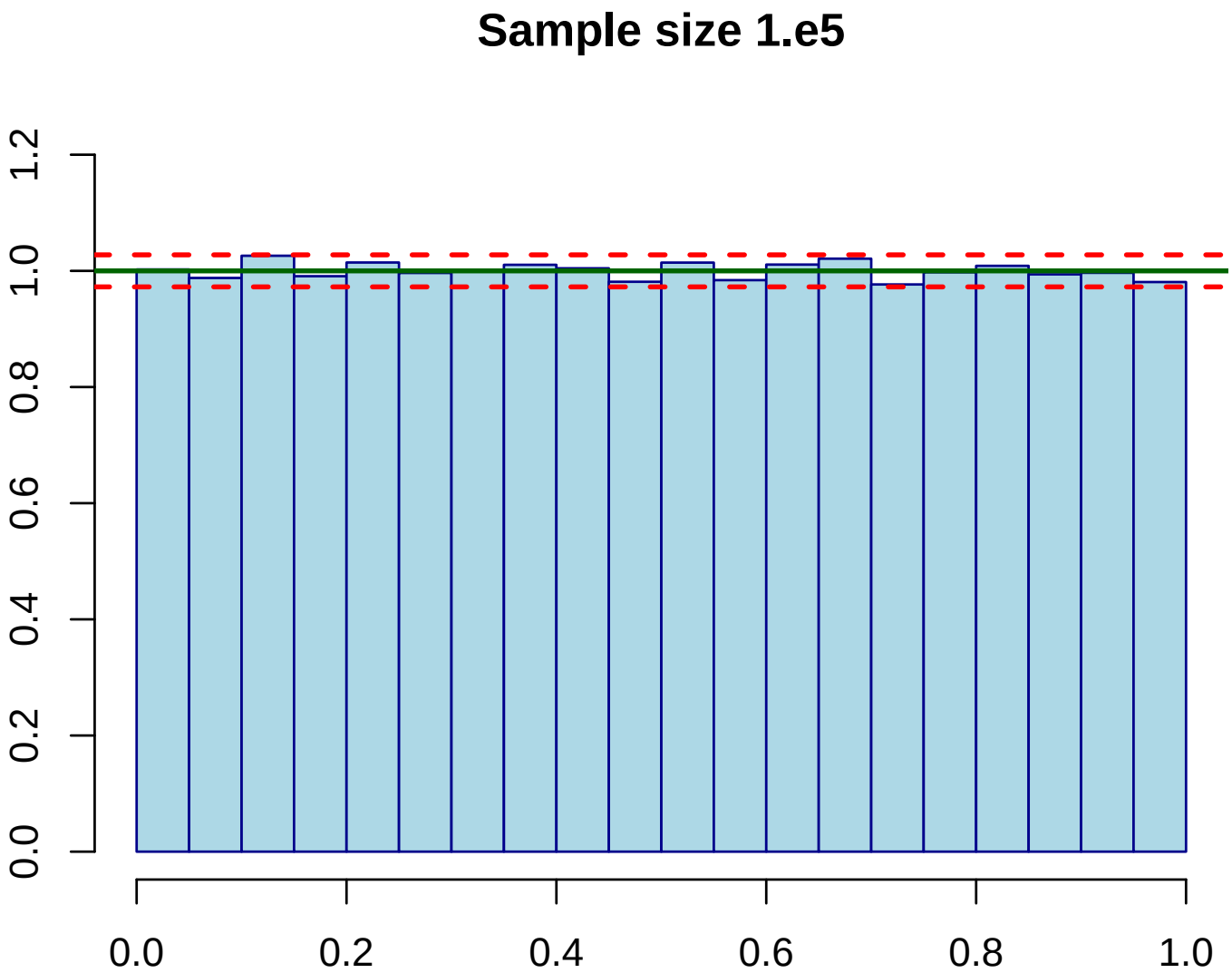
Increase Sample Size



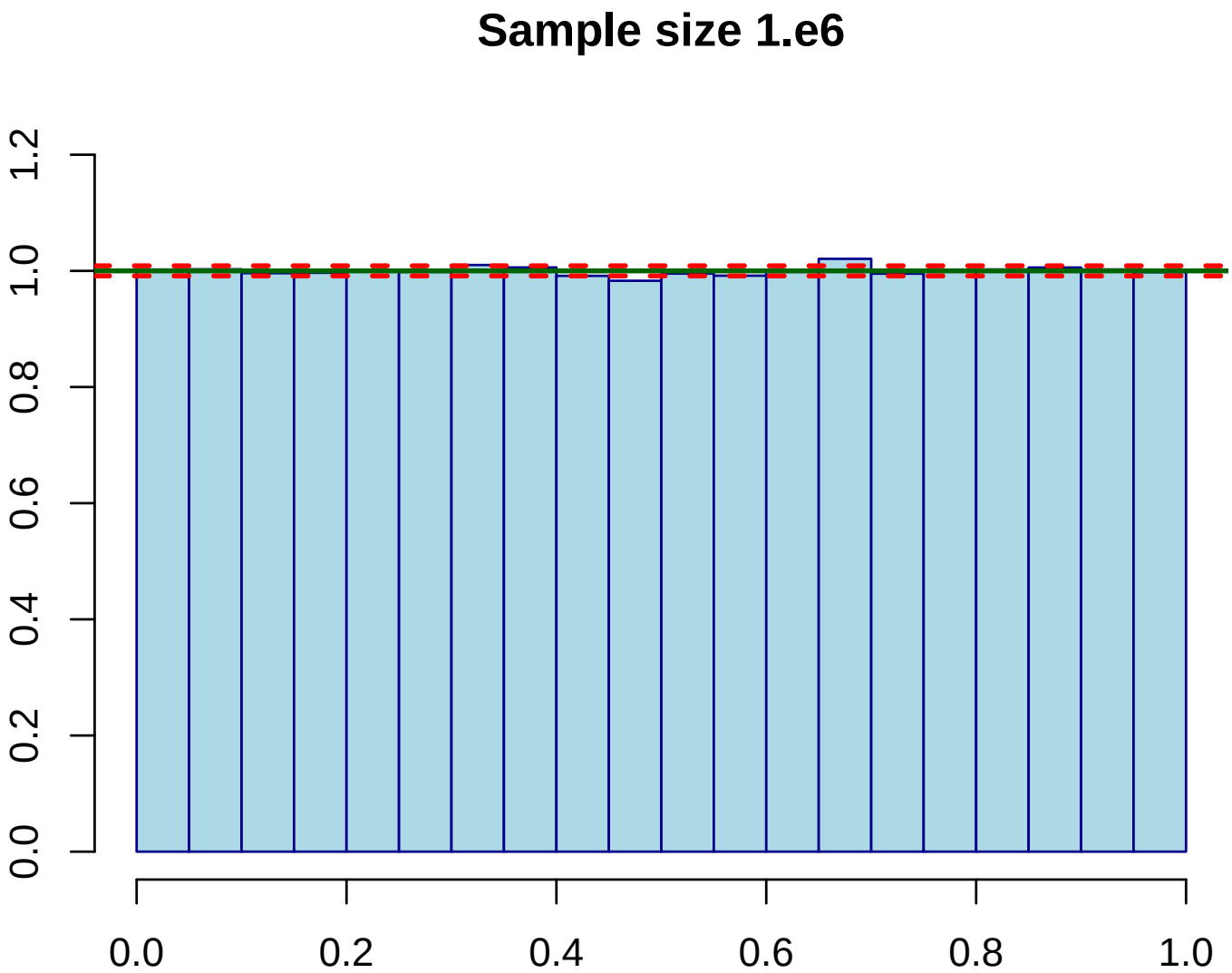
Increase Sample Size



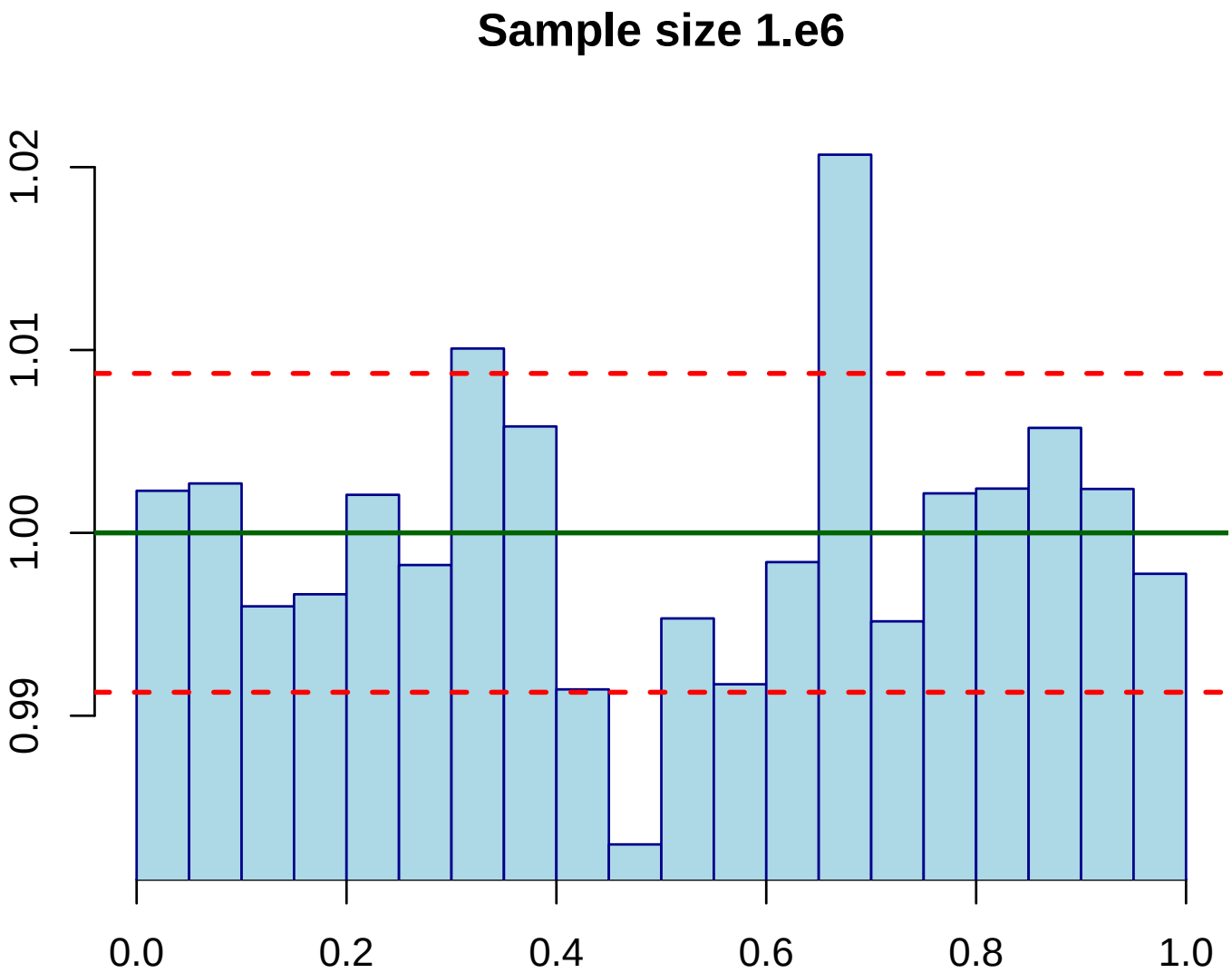
Increase Sample Size



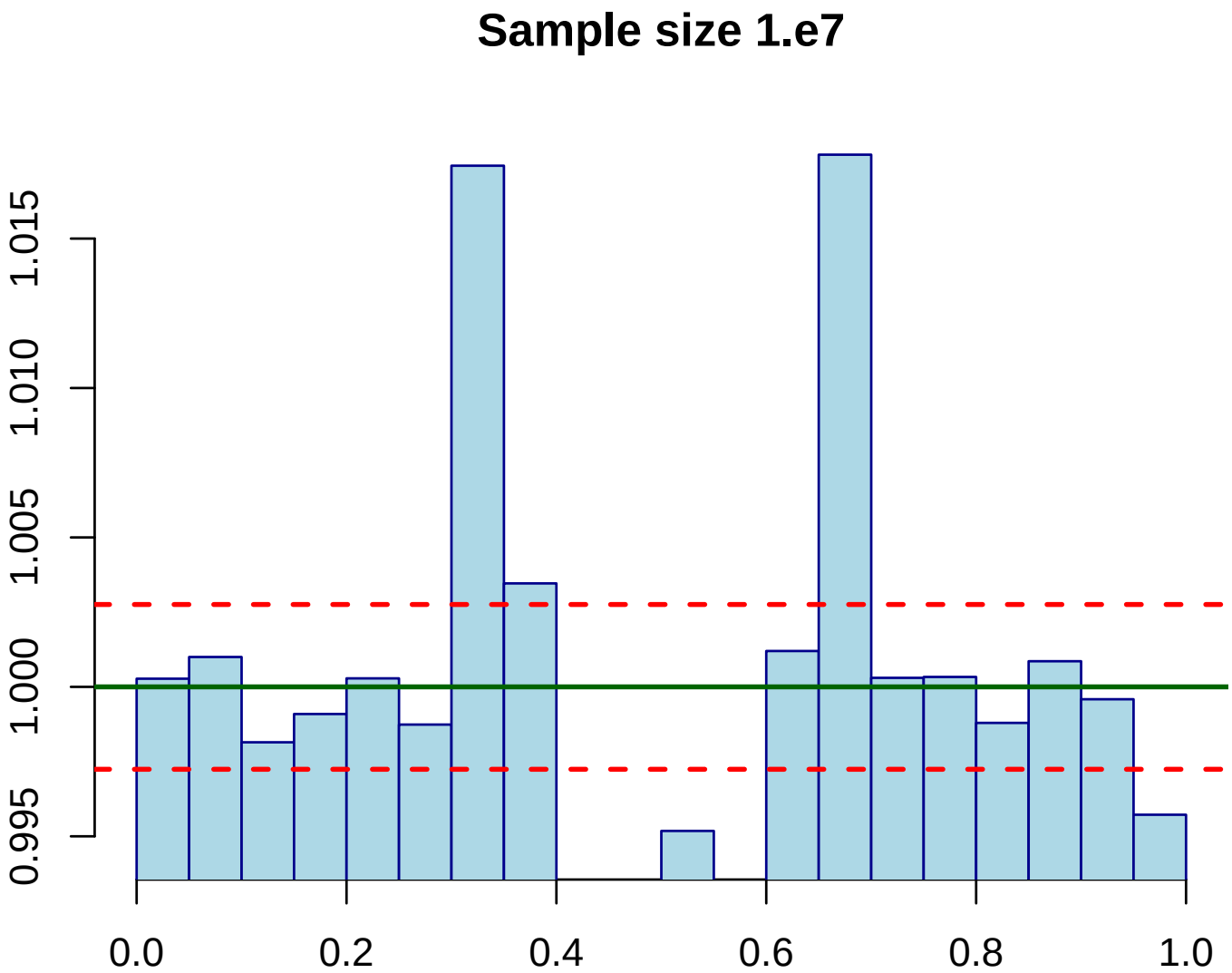
Increase Sample Size



Looking Closer



Looking Closer



Test with Increasing Sample Sizes

- Run χ^2 goodness-of-fit test

```
                1.e4      1.e5          1.e6          1.e7
p-value 0.03573423 0.4961224 1.046281e-05 1.500107e-89
```

- Run Kolmogorov-Smirnov test

```
                1.e4      1.e5         1.e6          1.e7
p-value 0.5586853 0.5722032 0.009953565 3.152891e-09
```

The Kinderman-Ramage Algorithm

The above error appeared in R prior to version 1.7.
Then the default Gaussian generator was the algorithm by Kinderman and Ramage [3] but one line of code was missing.

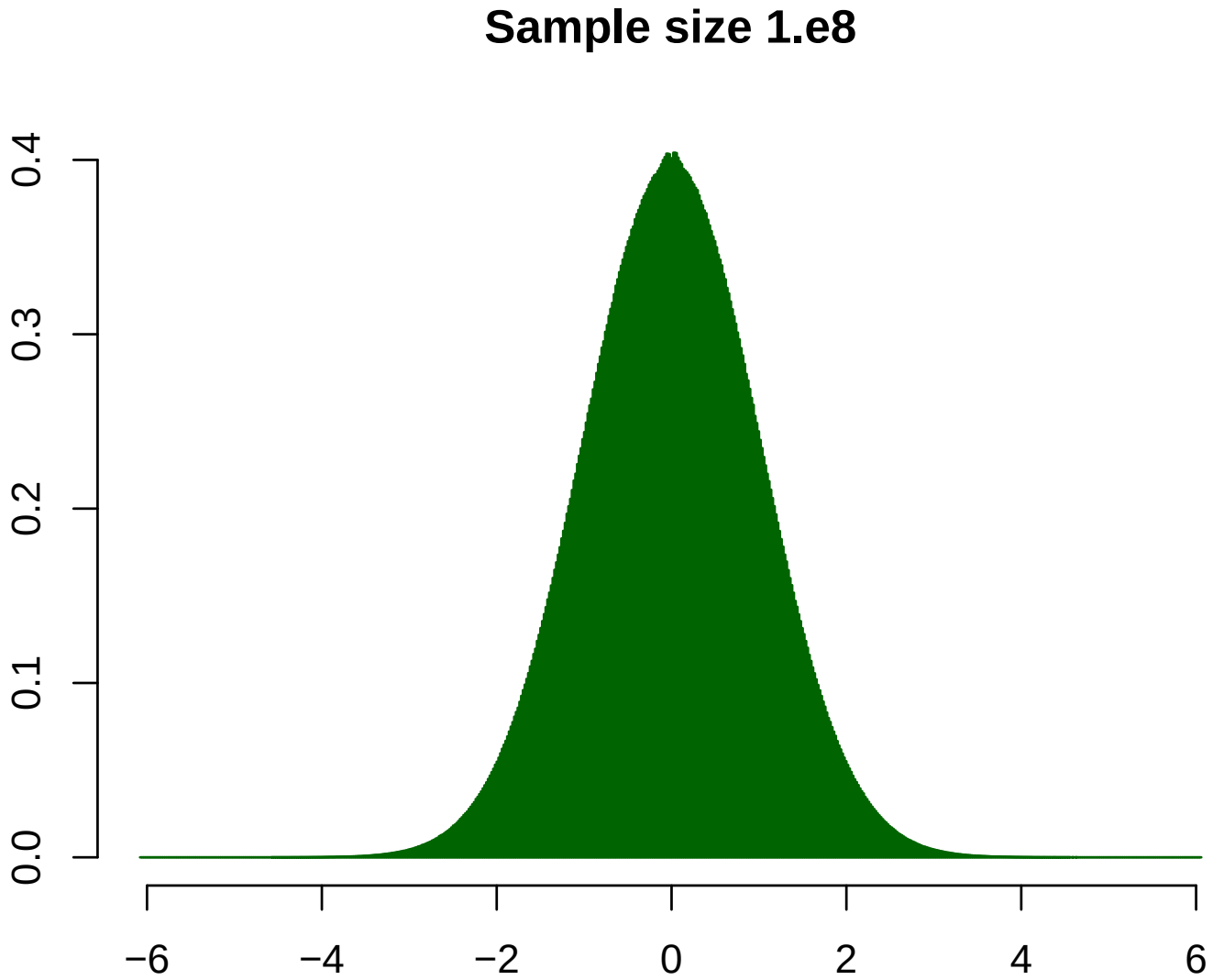
This generator was said to be the fastest Gaussian random variate generator and is implemented in several programs and libraries.

Meanwhile the default Gaussian generator is the quantile function (inversion method). One can play with buggy generator by setting the global generator using

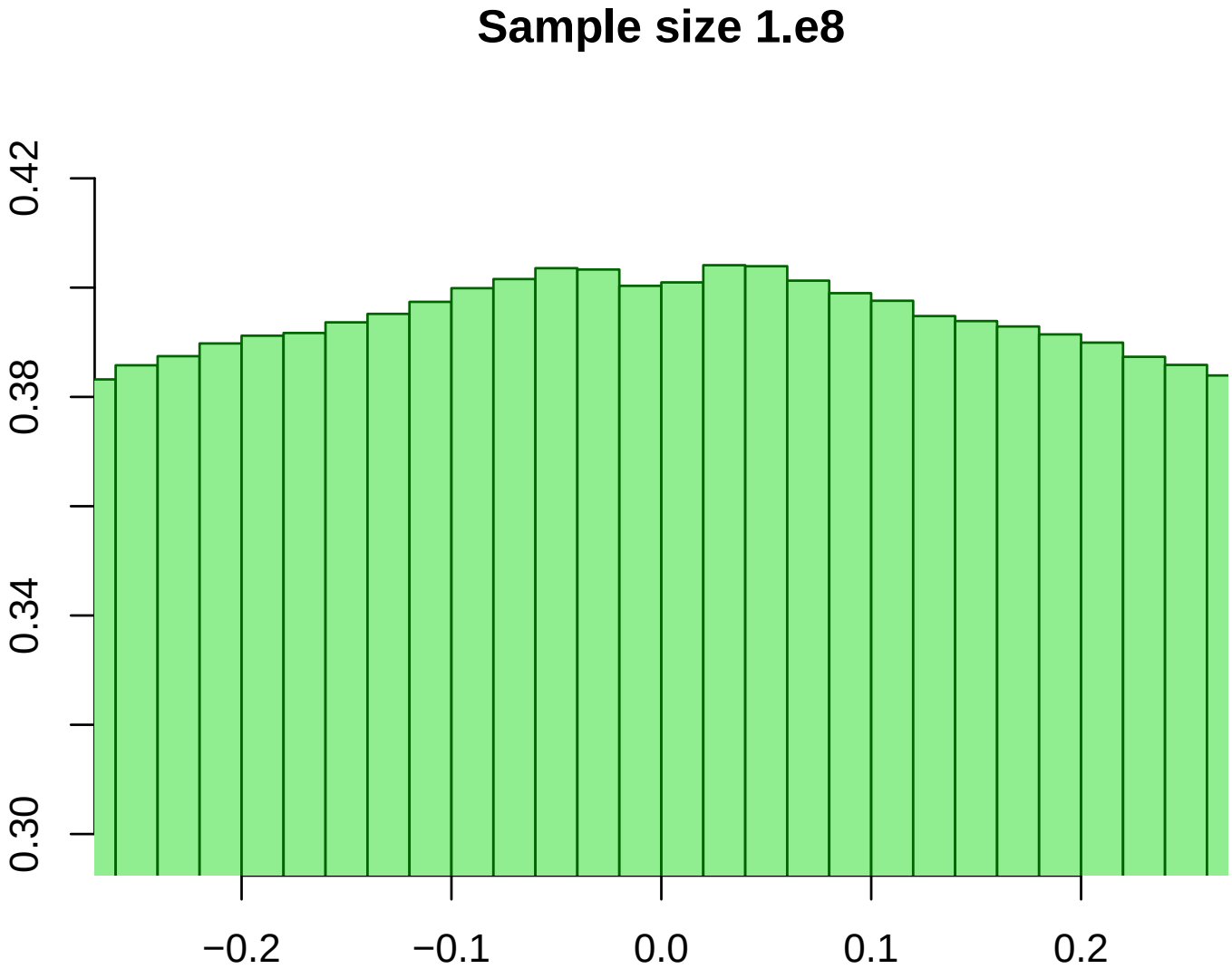
```
> RNGkind(normal.kind="Buggy Kinderman-Ramage")
```

So what happens when we fix this bug and use the originally published algorithm?

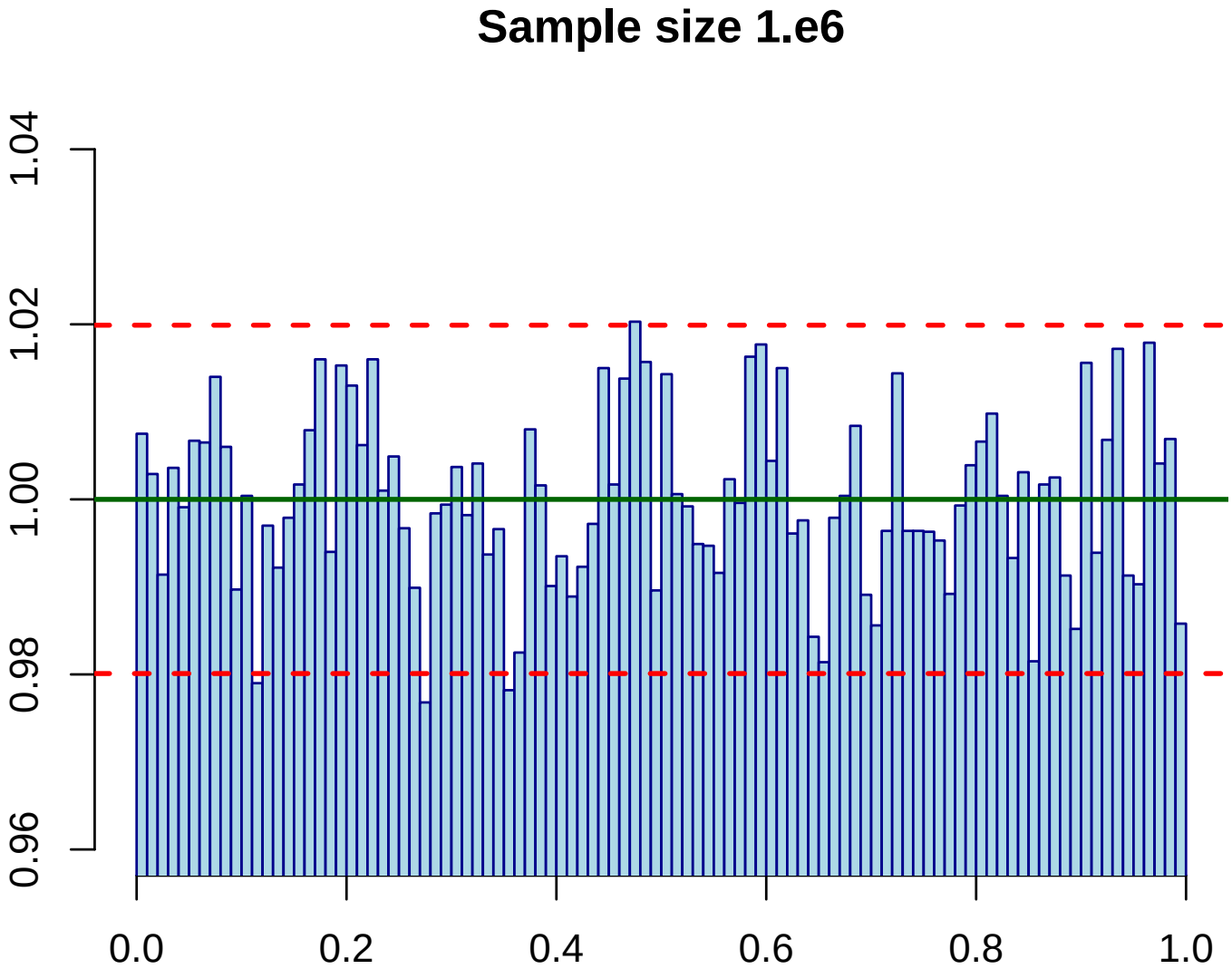
The Kinderman-Ramage Generator



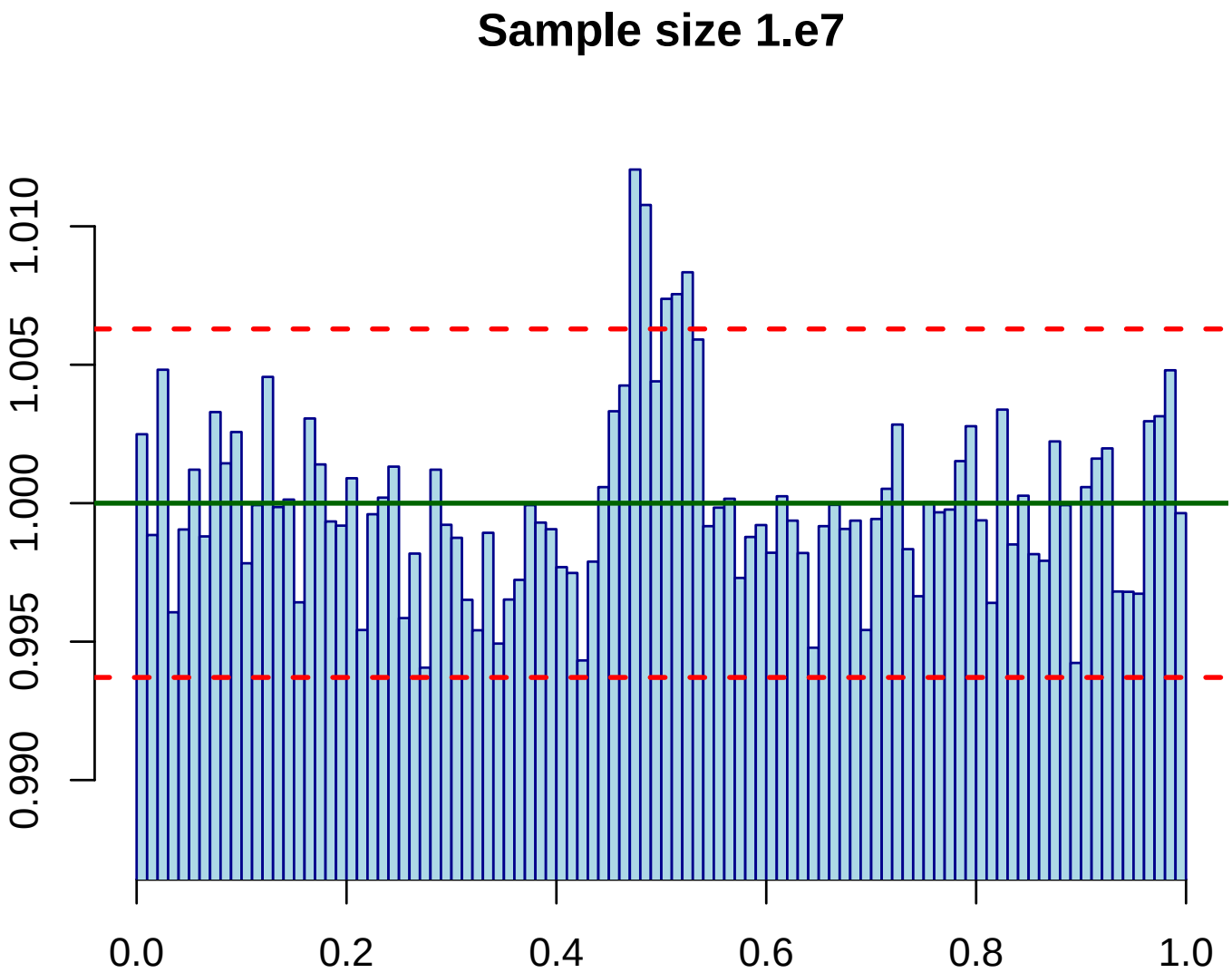
The Kinderman-Ramage Generator



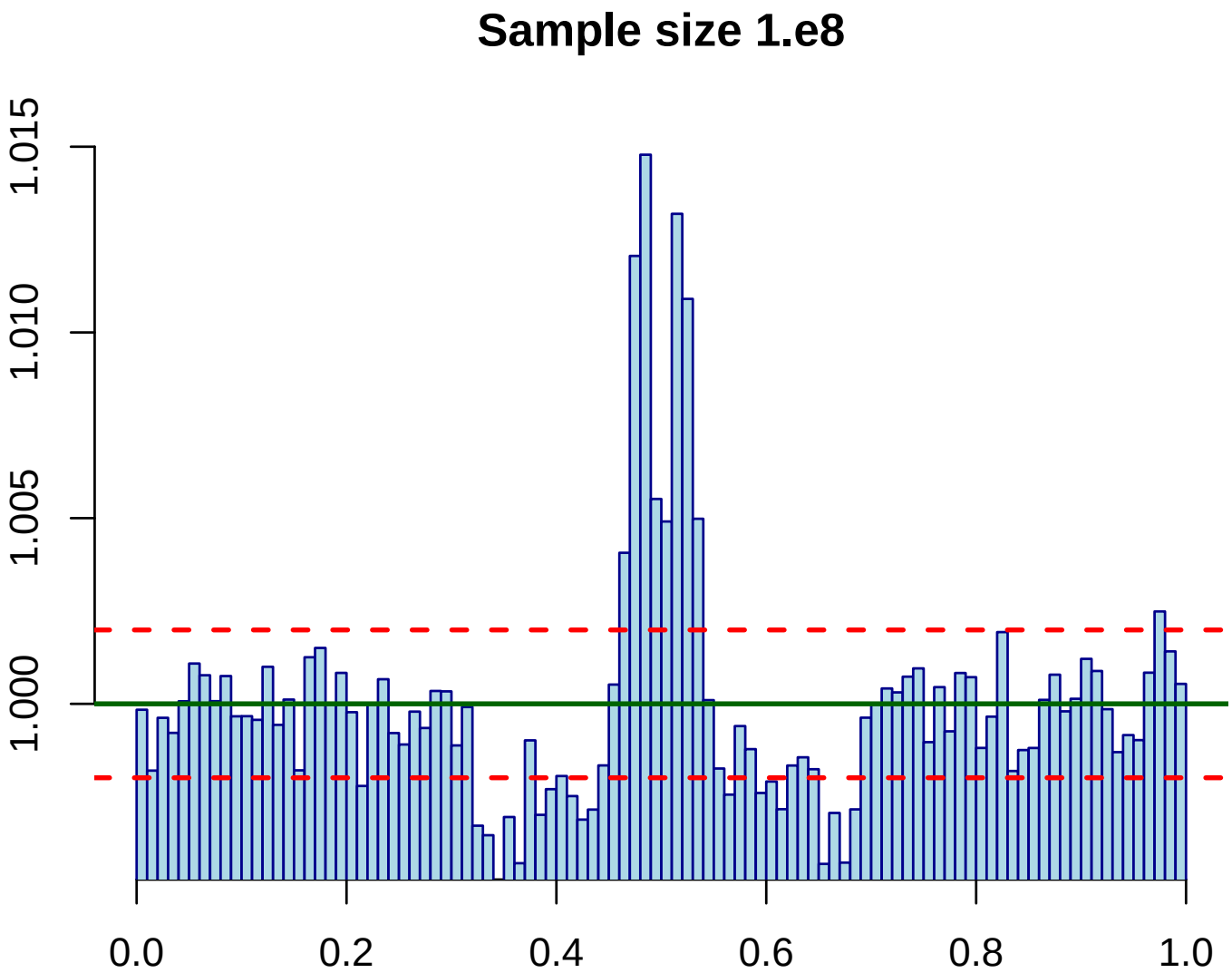
The Kinderman-Ramage Generator



The Kinderman-Ramage Generator



The Kinderman-Ramage Generator



The Kinderman-Ramage Algorithm

- Run χ^2 goodness-of-fit test

```
              1.e5      1.e6      1.e7           1.e8
p-value 0.822784 0.4299325 0.1891353 8.561413e-154
```

The error occured because a rejection statement was missing in the proposed algorithm. However, in 1976 the authors of the paper would have needed at least 8.5 hours of pure CPU time on a mainframe machine for just generating a sample of size 10^8.

A “Perfect” Slice Sampler

Perfect sampler turn MCMC sampler into exact sampling algorithms.

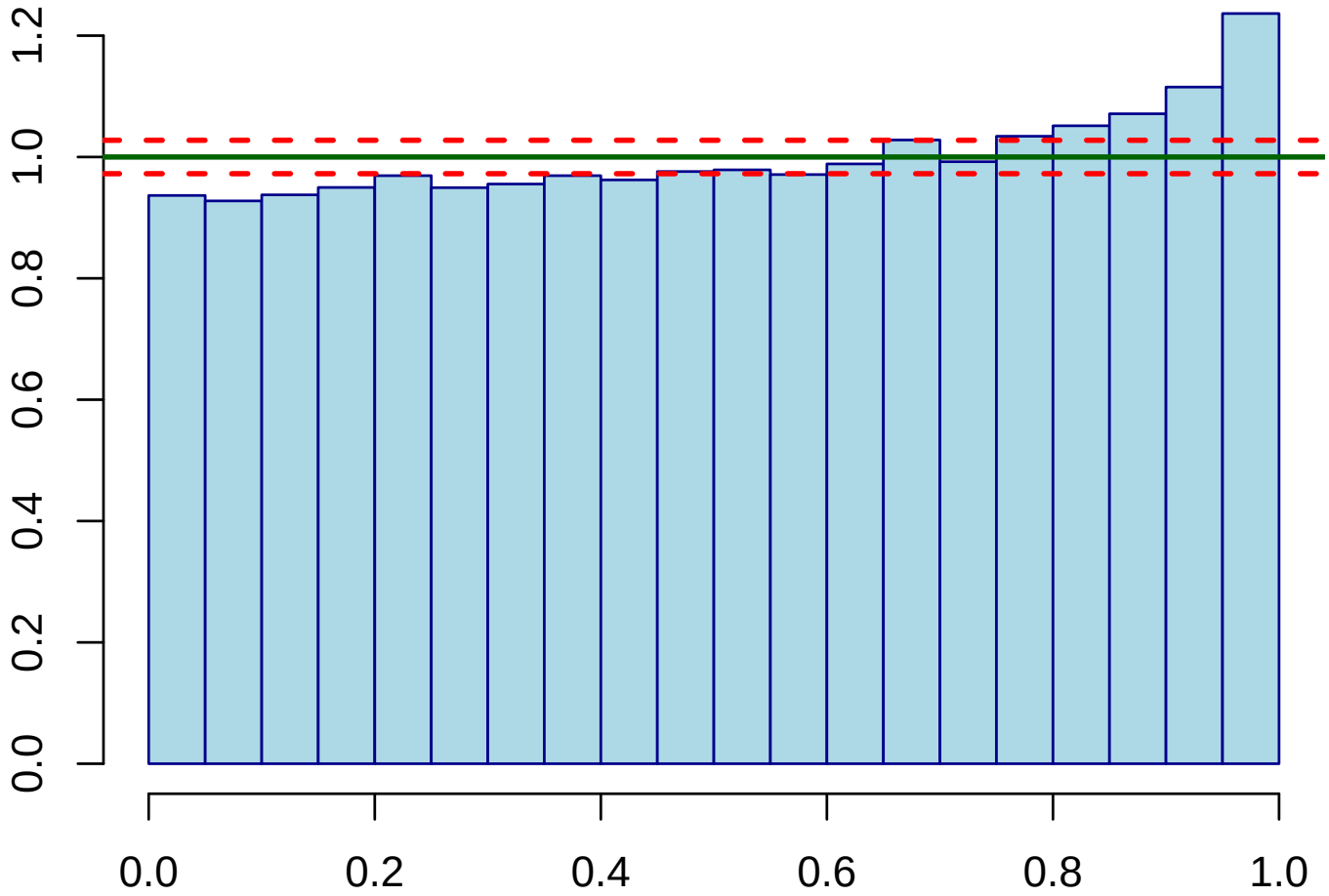
A popular method is **Coupling from the past** (CFTP) where two extremal chains are generated by means of the same sequence of random numbers. When the two chains coalesce, the state is returned.

A variant based on the slice sampler stores all states of the two chains. Thus a modification of the algorithm has been proposed that need only store the sequence of the uniform random numbers.

We ran the algorithm on a (truncated) exponential distribution.

A “Perfect” Slice Sampler

- Sample size = 10^5, $t_0 = 10$



Computational Limits

- F distribution with $\nu_1 = 10$ and $\nu_2 = 0.005$ degrees of freedom

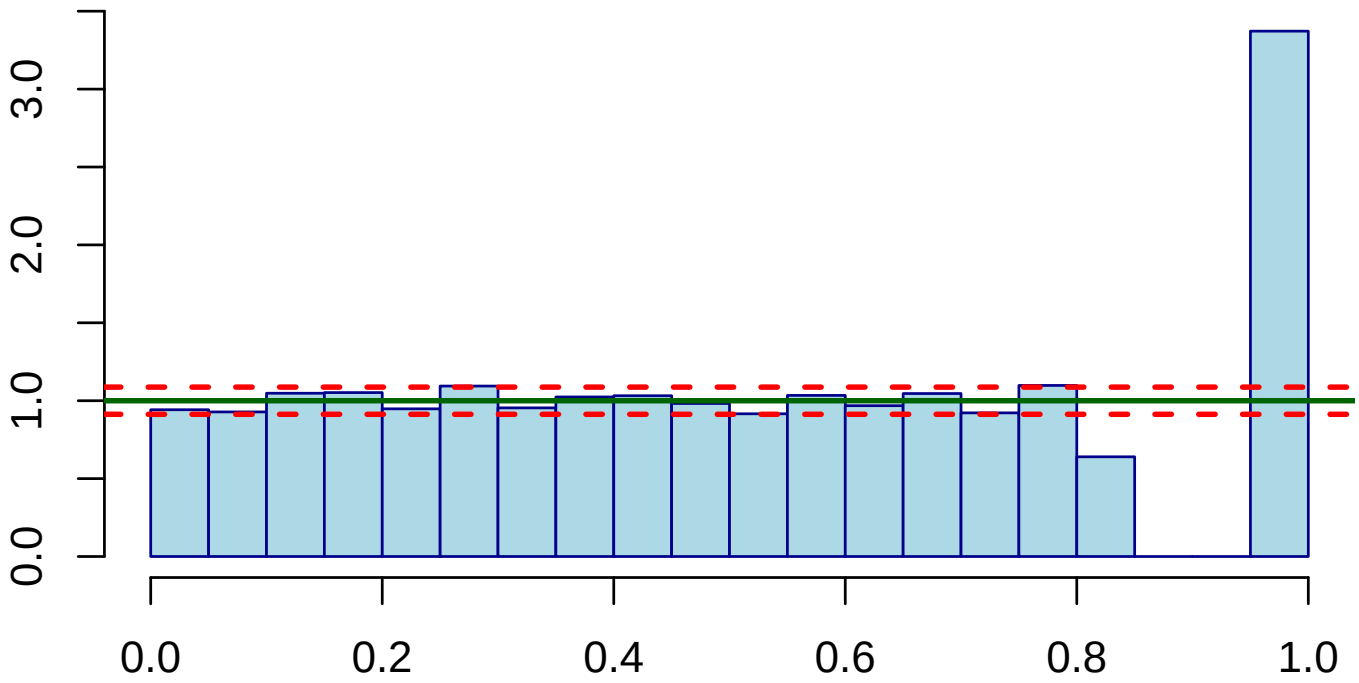
```
> X <- rf(1.e4, df1=10, df2=0.005)
> h <- hist(pf(X,10,0.005),breaks=100)
> chisq.test(h$counts)

Chi-squared test for given probabilities

data:  h$counts
X-squared = 26335.66, df = 99, p-value < 2.2e-16
```

Computational Limits

- F distribution with $\nu_1 = 10$ and $\nu_2 = 0.005$ degrees of freedom




```
> pf(1e300, df1=10, df2=0.005)
[1] 0.824609
```

Computational Limits

- Beta distribution with shape parameters 3 and 0.01

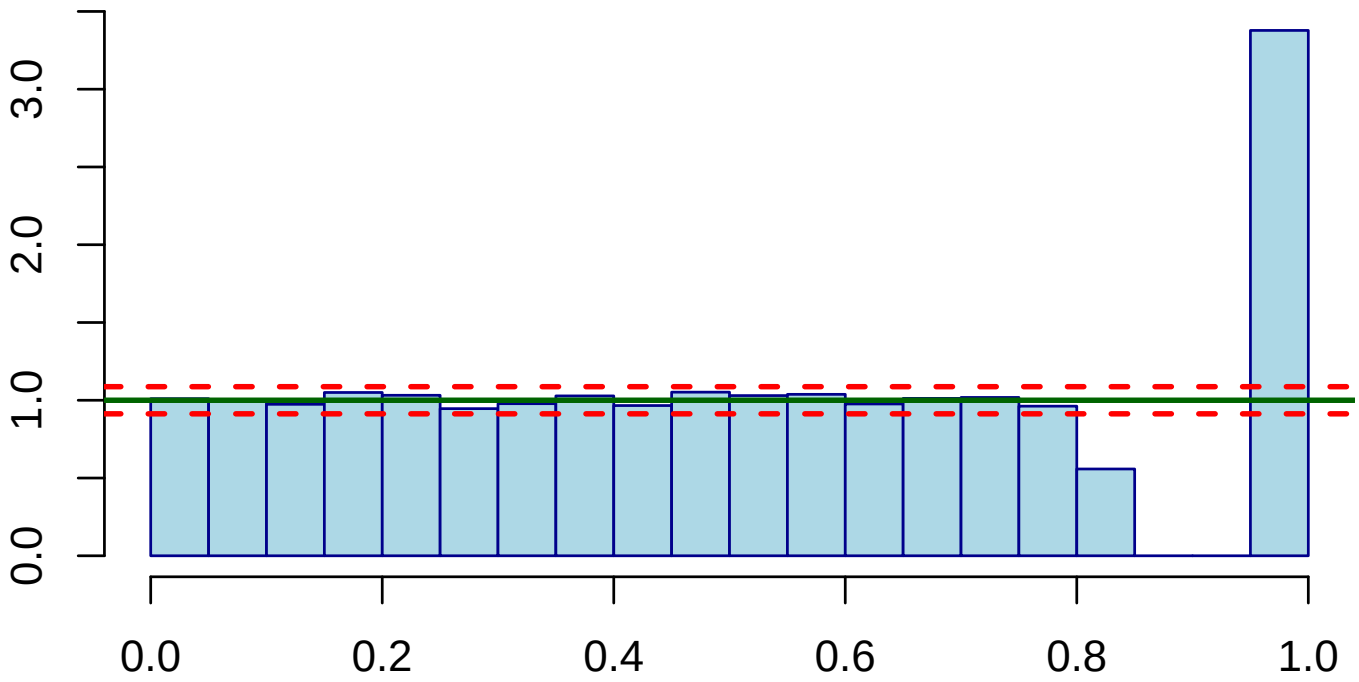
```
> X <- rbeta(1e4, shape1=3, shape2=0.05)
> h <- hist(pbeta(X,3,0.05),breaks=100)
> chisq.test(h$counts)

Chi-squared test for given probabilities

data:  h$counts
X-squared = 26707.86, df = 99, p-value < 2.2e-16
```

Computational Limits

- Beta distribution with shape parameters 3 and 0.01




```
> pbeta(c(1-2^(-52), 1-2^(-53), 1-2^(-54)),
+       shape1=3, shape2=0.05)
[1] 0.8224850 0.8285318 1.0000000
```

Extending R by Your Own Code

R can load shared libraries. Thus

- Time critical code can be implemented in C / C++ / Fortran / (Java) / ...
- The results of the code can be easily compared with the prototype code.
- Production-ready code can be included, used and/or checked within the R framework.
- External libraries can be used by means of quite simple wrapper functions.
- External code can be collected in packages for easier use.

Uniform Random Numbers

- Package `rstream`

implements a framework were random number generators are treated as objects. This allows for different in/dependent streams of random numbers from different sources.
In particular it provides an interface to one of L'Ecuyer's pseudo-random number generators with multiple substreams.

```
> library(rstream)
> rs <- new("rstream.lecuyer")
> rstream.sample(rs,4)
[1] 0.8008996 0.2685858 0.4814772 0.3634042
> rstream.reset(rs); rstream.antithetic(rs) <- TRUE
> rstream.sample(rs,4)
[1] 0.1991004 0.7314142 0.5185228 0.6365958
```

Runuran

Package `Runuran`

provides an interface to our UNU.RAN library.

```
> library(Runuran)
> lpdf <- function(x) { log(1-x)-x^2 }
> dlpdf <- function(x) { -1/(1-x)-2*x }
> distr <- unuran.cont(pdf=lpdf,dpdf=dlpdf,islog=TRUE,
+                      lb=-Inf,ub=1)
> urg <- unuran.new(distr,"arou")
> unuran.sample(urg,4)
[1] -0.10216088  0.08524869  0.16917772  0.23213163
```

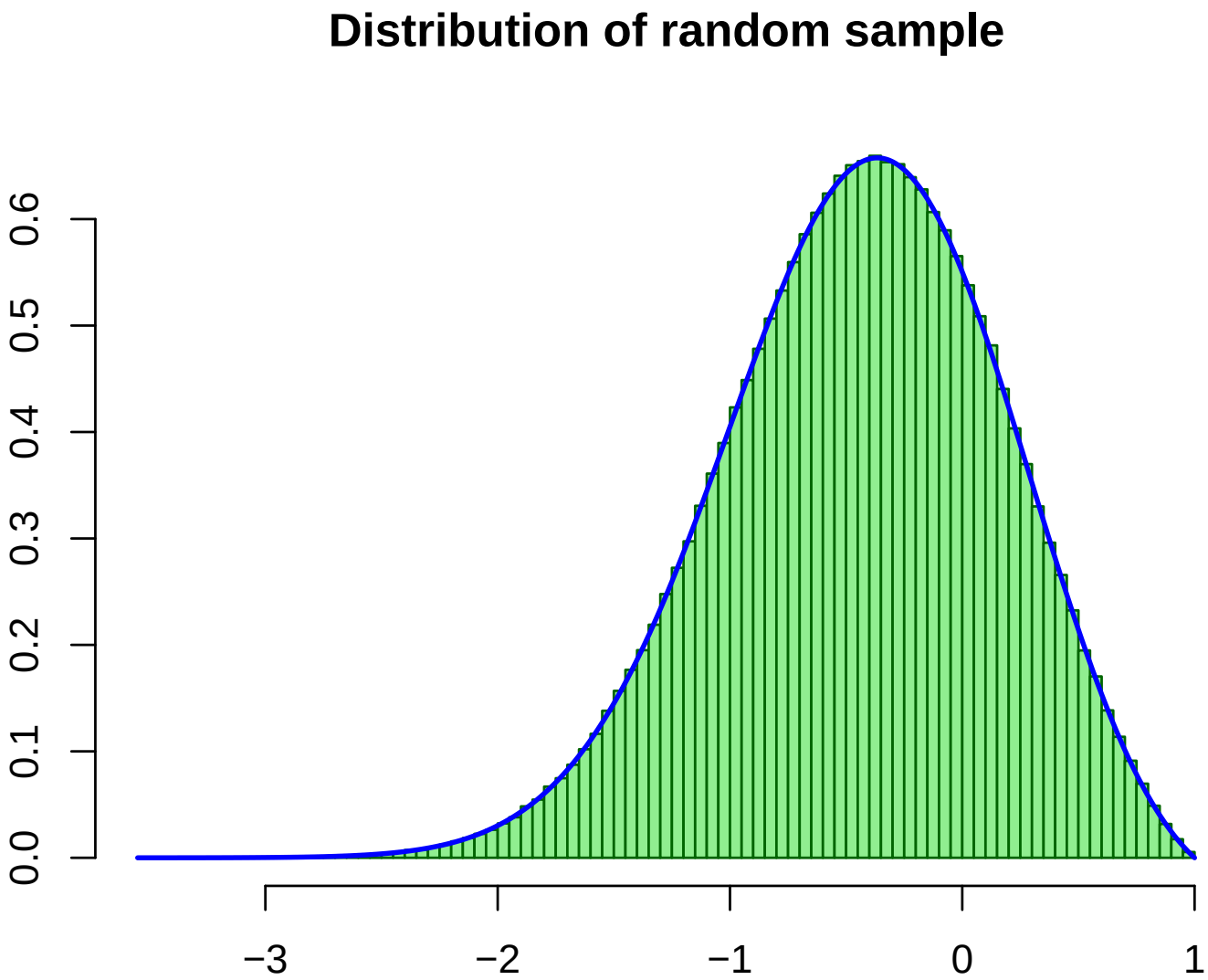
Runuran

Package `Runuran`

provides an interface to our UNU.RAN library.

```
> library(Runuran)
> lpdf <- function(x) { log(1-x)-x^2 }
> dlpdf <- function(x) { -1/(1-x)-2*x }
> distr <- unuran.cont(pdf=lpdf,dpdf=dlpdf,islog=TRUE,
+                      lb=-Inf,ub=1)
> urg <- unuran.new(distr,"arou")
> unuran.sample(urg,4)
[1] -0.10216088  0.08524869  0.16917772  0.23213163
> X <- unuran.sample(urg,1e6)
> nnpdf <- function(x) {exp(lpdf(x))}
> c <- 1/integrate(nnpdf,-100,1)$value
> pdf <- function(x) {c*exp(lpdf(x))}
> myhist(X, pdf)
```

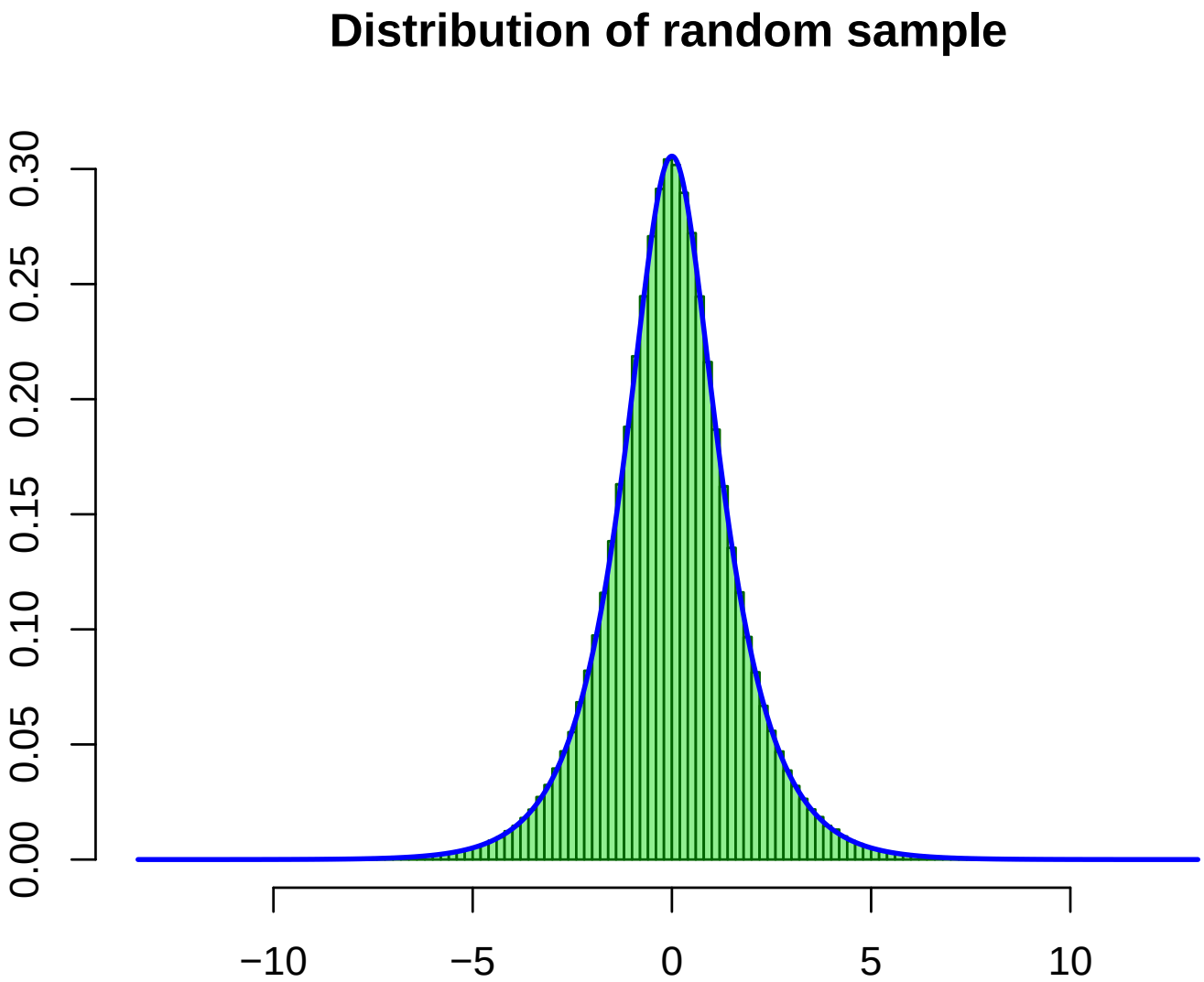
Maxwell Inflow Distribution



Hyperbolic Distribution

```
> lpdf <- function(x) { -sqrt(1+x^2) }
> dlpdf <- function(x) { -x/sqrt(1+x^2) }
> distr <- unuran.cont(pdf=lpdf,dpdf=dlpdf,islog=TRUE)
> urg <- unuran.new(distr,"arou")
> X <- unuran.sample(urg,1e6)
```

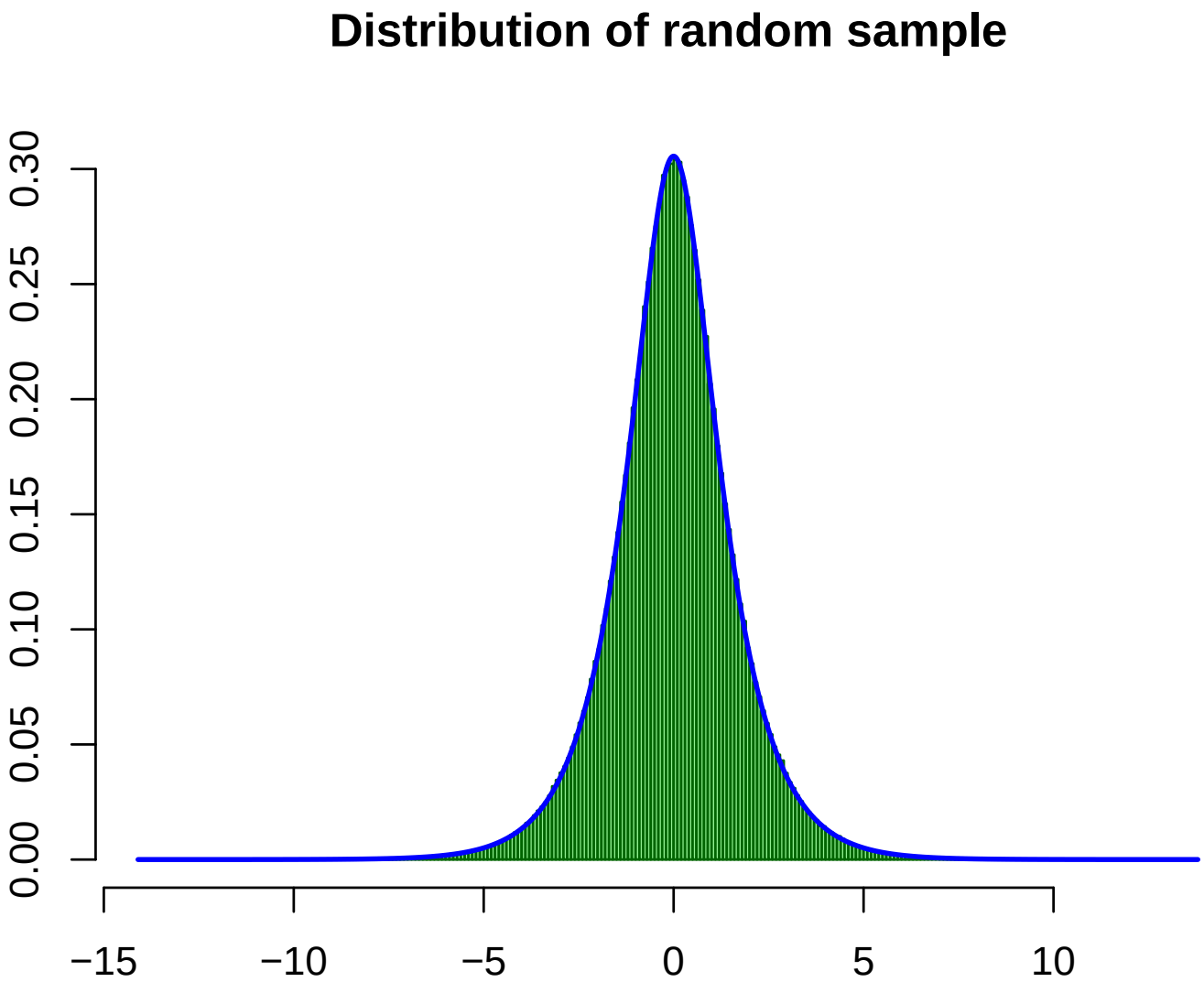
Hyperbolic Distribution



Hyperpolic Distribution by Inversion

```
> lpdf <- function(x) { -sqrt(1+x^2) }
> distr <- unuran.cont(pdf=lpdf,islog=TRUE)
> urg <- unuran.new(distr,"pinv")
> X <- unuran.sample(urg,1e6)
```

Hyperpolic Distribution by Inversion



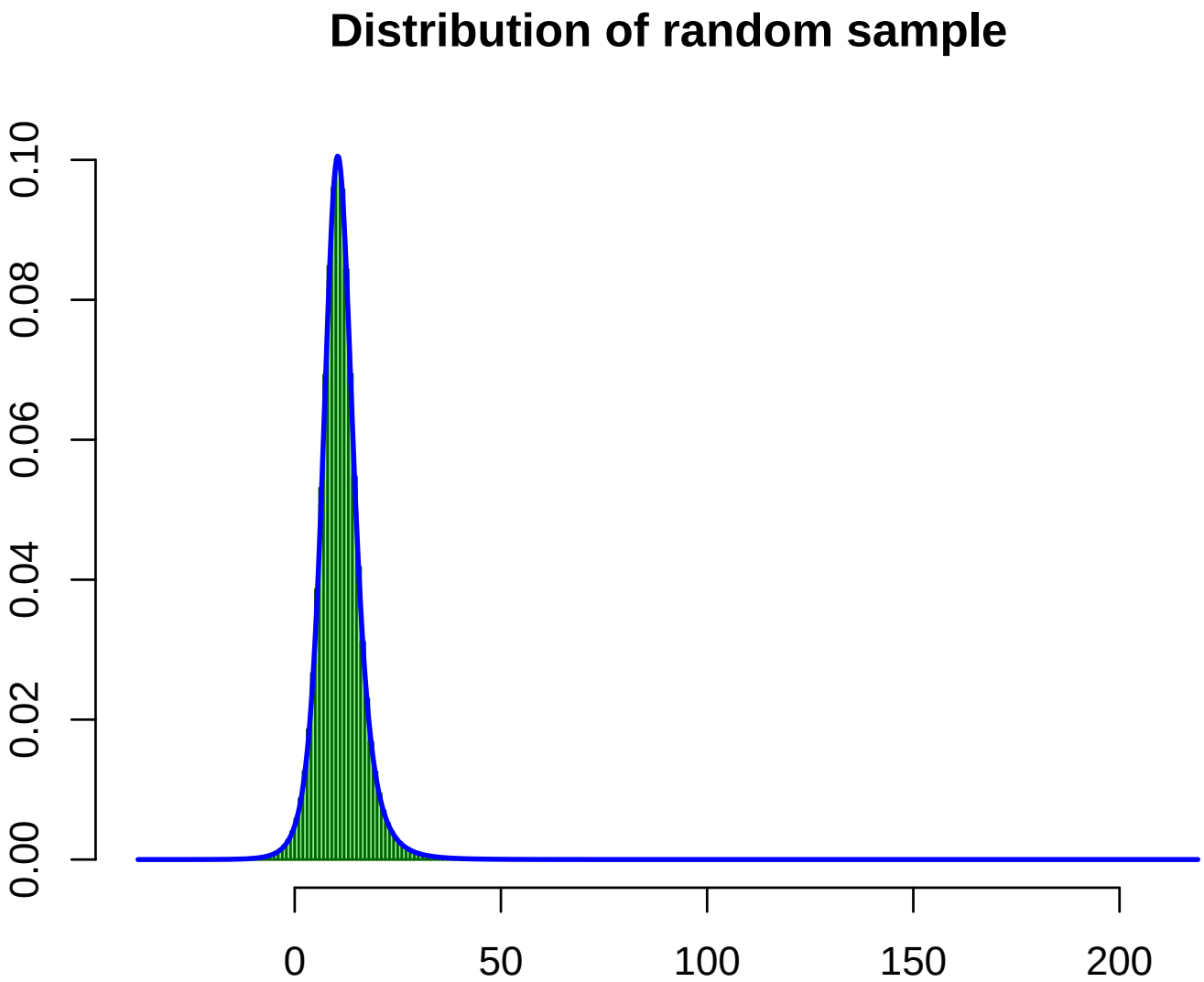
Generalized Hyperpolic Distribution

```
> library(ghyp)

  ghyp, 2007, Institute of Data Analysis and Process Design, GPL

> gh <- ghyp(lambda = -2, alpha.bar = 0.5,
+            mu = 10, sigma = 5, gamma = 1)
> pdf <- function(x) { dghyp(x,gh) }
> distr <- unuran.cont(pdf=pdf,islog=FALSE)
> urg <- unuran.new(distr,"pinv")
> X <- unuran.sample(urg,1e6)
```

Generalized Hyperpolic Distribution



Conclusion

Before using a new non-uniform random variate generator use an interactive environment to

- visualize the resulting samples;
- run various tests;
- play around with different parameters.

The R programming language for statistical computing is well suited for this purpose.

Conclusion

Before using a new non-uniform random variate generator use an interactive environment to

- visualize the resulting samples;
- run various tests;
- play around with different parameters.

The R programming language for statistical computing is well suited for this purpose.

Beware:
You can never verify the correctness of a particular random variate generator!

Thank You

References I

[1] *Root. An object-oriented data analysis framework*. CERN, Geneve, Switzerland. URL `http://root.cern.ch/`.

[2] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2008. URL `http://www.R-project.org`. ISBN 3-900051-07-0.

[3] A. J. Kinderman and J. G. Ramage. Computer generation of normal random variables. *J. Am. Stat. Assoc.*, 71(356):893–898, 1976.

[4] Günter Tirler, Peter Dalgaard, Wolfgang Hörmann, and Josef Leydold. An error in the Kinderman-Ramage method and how to fix it. *Computational Statistics and Data Analysis*, 47(3):433–440, 2004. doi: doi:10.1016/j.csda.2003.11.019.

[5] James G. Propp and David B. Wilson. Exact sampling with coupled Markov chains and applications to statistical mechanics. *Random Structures and Algorithms*, 9:223–252, 1996.

References II

[6] G. Casella, K. L. Mengersen, C. P. Robert, and D. M. Titterington. Perfect slice samplers for mixture distributions. *Journal of the Royal Statistical Society B*, 64:777–790, 2002.

[7] Anne Philippe and Christian P. Robert. Perfect simulation of positive Gaussian distributions. *Statistics and Computing*, 13:179–186, 2003.

[8] Christian P. Robert and George Casella. *Monte Carlo Statistical Methods*. Springer Verlag, 2 edition, 2004.

[9] Pierre L'Ecuyer and Josef Leydold. xxxx. *R News*, x(x):x–x, 2005.

[10] Pierre L'Ecuyer, Richard Simard, E. Jack Chen, and W. David Kelton. An object-oriented random-number package with many long streams and substreams. *Operations Research*, 50(6):1073–1075, 2002.

[11] Josef Leydold and Wolfgang Hörmann. *UNU.RAN – A Library for Universal Non-Uniform Random Variate Generation*. Department of Statistics and Mathematics, WU Wien, A-1090 Wien, Austria, 2000–2007. available at `http://statmath.wu-wien.ac.at/unuran/`.

References III

[12] Josef Leydold and Wolfgang Hörmann. Runuran – *R interface to the UNU.RAN random variate generators*. Department of Statistics and Mathematics, WU Wien, A-1090 Wien, Austria, 2007. available at `http://cran.r-project.org/`.

[13] Wolfgang Hörmann, Josef Leydold, and Gerhard Derflinger. *Automatic Nonuniform Random Variate Generation*. Springer-Verlag, Berlin Heidelberg, 2004.