rvgtest

Error Detection in Non-Uniform Random Variate Generators

Josef Leydold

Institute for Statistics and Mathematics, WU Vienna, Austria

Version 0.7.4 – Feb 26, 2014

Abstract

Non-uniform random variate generators are of fundamental importance in Monte-Carlo methods and stochastic simulation. They are based on the assumption that a source of uniformly distributed random numbers is available that produces real and truely random numbers. In practice, however, we have to use pseudo-random numbers which are generated by means of floating point numbers. Thus there are deviations from the theoretical framework caused by round-off errors which are due to the limitations of floating point arithmetic. In addition algorithm have to developed and implemented. As human can err coding errors or mistakes in the design of algorithm may cause defects in a generated random sample. It is of paramount importance to detect such errors using statistical tools and categorize which tests are sensitive to which errors.

rvgtest is a collection of routines for testing non-uniform random variate generators. They hopefully should help to detect all kinds of error. Thus it provides routines for goodness-of-fit tests that are based on histograms and routines to estimate approximation errors for numerical inversion.

Table of Contents

1 Introduction 2

2 Goodness-of-Fit Tests based on Histograms 5

3 Approximation Errors in Numeric Inversion 14

References 21

1 Introduction

Non-uniform random variate generation is a small field of research between mathematics, statistics and computer science. It plays a very crucial role in every stochastic simulation. A lot of research has been done to evolve more efficient and robust algorithms to generate non-uniform random variates. These algorithms usually transform a sequence of independent uniformly distributed random numbers on $(0, 1)$ into a sequence of independent random variates of the desired distribution.

The theory of non-uniform random variate generation is based on the assumption that a source of uniformly distributed random numbers is available that produces real and truely random numbers. In practice, however, we have to use pseudo-random numbers which are generated by means of floating point numbers. Thus there are deviations from the theoretical framework which may or may not have influence on the result of a Monte Carlo simulation.

In addition algorithm have to developed and implemented. As human can err, coding errors or improper design of algorithms may cause defects in a generated random sample. It is of paramount importance to detect such errors using statistical tools and categorize which tests are sensitive to which errors.

1.1 Possible Kind of Errors

There are various reasons why a particular random variate generator in ones computing environment may produce a defective random sample. There are three mean reasons for such defects:

(1) Implementation errors – Mistakes in computer programs.

(2) Errors in the design of algorithms – The proof for theorem that claims the correctness of the algorithm is wrong.

(3) Limitations of floating point arithmetic and round-off errors in implementations of these algorithms in real world computers.

Error of type 1 is of course common to any computer program. An example for an error of type 2 is the *Kinderman-Ramage* generator [6] for the standard normal distribution. Although it has been used in several programs for statistical computing for many years there has been an small error deviation caused by missing line in the statement of the algorithm, see [8] for details.

Errors of type 3 can arise in distributions with extreme properties. E.g., it is no problem to generate a sample of beta distributed random variates with shape parameters 3 and 0.05 using `rbeta(n=1, shape1=3, shape2=0.05)`. However, the following piece of code should plot the histogram of a uniformly distributed random variate:

Example

```
> X <- rbeta(10000, shape1=3, shape2=0.05)
> U <- pbeta(X, shape1=3, shape2=0.05)
> hist(U)
```

However, this is not the case as Fig. 1 shows. It is obvious that a goodness-of-fit must fail. The reason for this numeric result is given by the limitations of floating point arithmetic. Thus the beta distribution degenerates into a discrete distribution near one as can be seen by following example. There are not floating point numbers between $1 - 2^{-53}$ and 1.

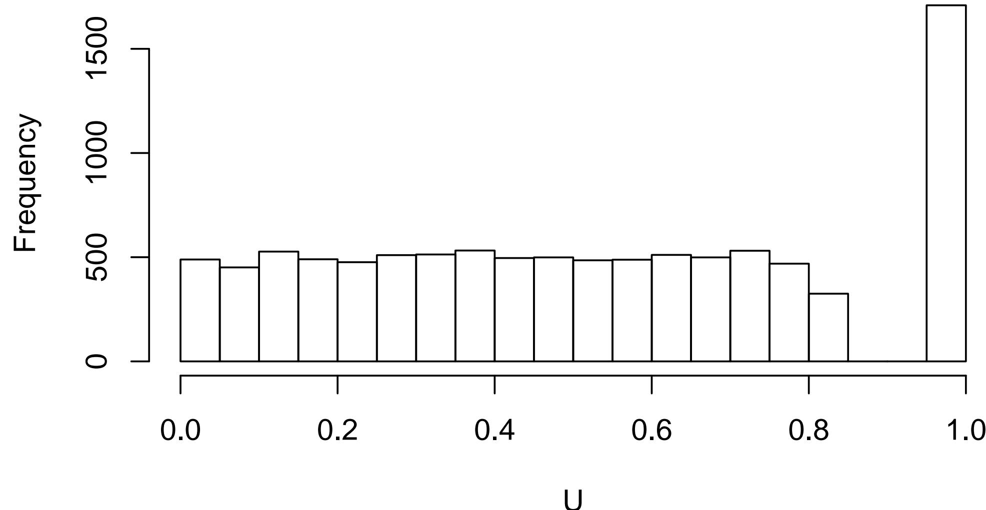


Figure 1: Histogram of a sample of beta distributed random variates with shape parameters 3 and 0.05

Example

```
> x <- c(1-2^(-52), 1-2^(-53), 1-2^(-54), 1-2^(-54))
> pbeta(x, shape1=3, shape2=0.05)

[1] 0.8224850 0.8285318 1.0000000 1.0000000

> 1-x

[1] 2.220446e-16 1.110223e-16 0.000000e+00 0.000000e+00
```

Although this is an extreme example (and one can thick of even worse) this error is of the "nice" type. The limitations of floating point arithmetic should be well-known to everybody who is working with numeric methods[1]. It cannot be avoided[2] but these limitations can taken into consideration when a particular representation of model is created.

1.2 Detecting Errors

Other errors can be quite nasty as they are hard to detect. Since we are dealing with "random" numbers there is not a unique exact solution. It is more the opposite. There will be deviation from the exact distribution. Even further, there must be deviation from the perfect

[1] Overton [7] gives a good introduction into the IEEE floating point arithmetic that is implemented in all modern computers.

[2] There exist libraries that implement multiple precision arithmetic. However, they only shift the boundary when computation became inexact. They do not solve the problem.

distribution. Nobody would trust a dice when 60 throws would give exactly 10 times each face. So we need statistical method to detect deviation of the empirical distribution from the theoretical distribution that are too large, that is, that cannot just happen by chance. The smaller the deviations are the larger the sample size must be. For example, to detect the error in the Kinderman-Ramage generator it requires a sample of size 10^7 which was much too large when the generator was developed, see [8]. Nowadays one can create and test such a sample within a few seconds. This shows that the term "exact generator" is also heavily linked to the available computing environment and changes with the evolution of computer hardware.

It is the purpose of package rvgtest to provide tools to find possible errors in RVGs. However, observing a defect in (the implementation of) a pseudo-random variate generator by purely statistical tools may require a large sample size which that exceeds the memory when hold in a single array in R. (Nevertheless, there is some chance that this defect causes an error in a particular simulation with a moderate sample size.) Hence we have implemented routines that can run tests on very large sample sizes (which are only limited by the available runtimes).

Currently there are two tool-sets for testing random variate generators for *univariate distributions*:

- Tests based on histograms for all kinds of RVGs.
- Estimating errors of RVGs that are based on numerical inversion.

1.3 Non-uniform Random Variate Generation

The interested reader may find much more information on non-uniform random variate generation by the books of Devroye [2] and Hörmann, Leydold, and Derflinger [5].

2 Goodness-of-Fit Tests based on Histograms

A frequently used method for testing univariate distributions is based on the following strategy: Draw a sample, compute a histogram and run a goodness-of-fit test on the resulting frequency table. Often the random sample is transformed into a sample of uniformly distributed random numbers by means of the distribution function. This allows for visual inspection of the empirical distribution and tests for uniformity can be applied for all distributions.

Generating the random sample and (even worse) the transformation into the uniform scale is the most expensive step. Moreover, (extremely) large sample sizes may exceed the physical memory of the computer and results an extremely long run times due to swapping of memory. If the table size even exceeds the virtual memory, then computation is aborted.

Thus we have implemented a three step procedure:

1. Create a table of bin frequencies that hold the information of huge random samples. In order to save memory only small samples are drawn at once. This also provides some information about the sample size at which a possible defect becomes statistical significant.

2. Perform some goodness-of-fit tests.

3. Visualize the frequency table (histogram) as well the results of the GoF tests (plot p-values against sample size).

The advantages of this procedure are obvious:

- Huge total sample sizes are possible (only limited by available runtime but not by memory).

- We can run multiple tests on the same random sample. Thus we can compare the power of different tests.

- Inspect data and compare visually.

2.1 An Example

Here is a simple example (see the corresponding help pages for details, the plots are shown in Figs. 3 and 2):

Example

```
> ##   Normal distribution
> ## ......................................................................
> #
> ## Create frequency table.
> ## We use a sample of 20 times 10^5 random variates.
> ft <- rvgt.ftable(n=1e5, rep=20, rdist=rnorm, qdist=qnorm)
> #
> ## We can visualize this table by drawing the histogram
> plot(ft)
> #
> ## Perform a chi-square goodness-of-fit test
> tr <- rvgt.chisq(ft)
> tr


rvgtest - test:
   type           = chisq
   sample size    = 2e+06
   # break points = 101
   p-value        = 0.4517403

> ## We can visualize the p-values for increasing sample size
> plot(tr)
```

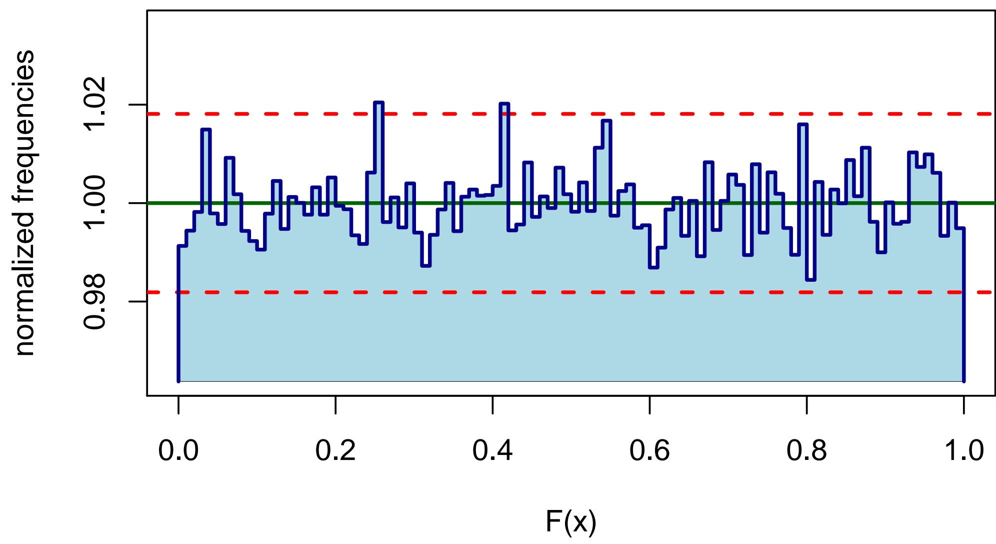


Figure 2: Histogram of a sample of normal distributed random variates. The green line shows the expected value for each bin height, the red dashed lines show the boundary of the confidence interval.

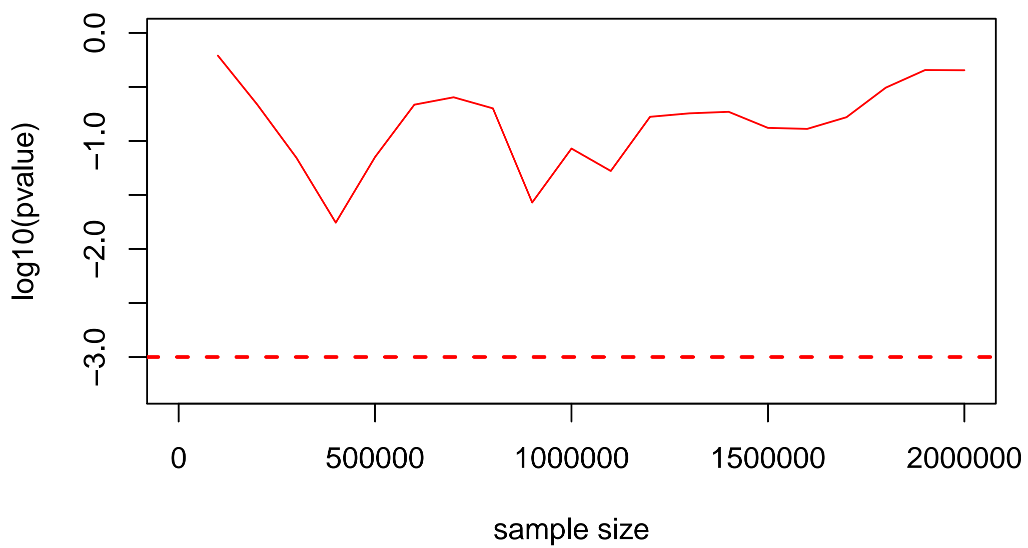


Figure 3: p-values of a chisquare goodness-of-fit test. p-value is plotted against sample size.

2.2 Creating the Frequency Table

Computing the bin counts of a histogram usually is straightforward. For our purpose the following should be kept in mind:

- We have to create huge samples. Thus execution time is important. So we should avoid as many computations as possible besides running the random variate generator.
- Random variate generators transform uniform random numbers into non-inform ones. Thus break points should be given in the uniform scale. In particular, the bins should have equal probabilities except for the case where we want to test against a particular alternative.

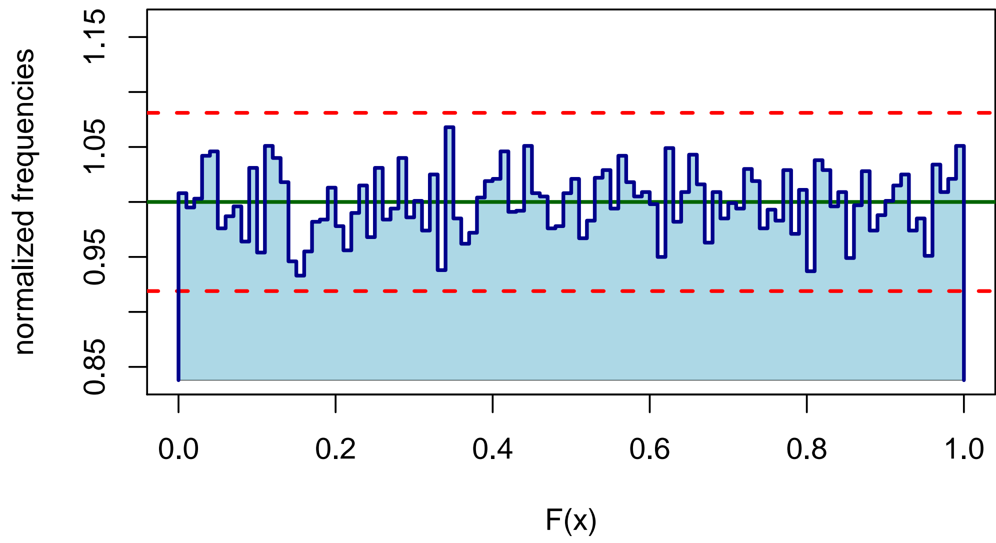
Thus we have implemented the following variants:

1. If the quantile function is given, we transform the break points of in u-scale into break points into x-scale. Thus we only have to draw a random sample. This is the fasted method to create the table. Moreover, the theoretical distribution of the frequencies is as requested by the user (e.g., equidistributed). The drawback of this method is, that it requires the quantile function and that it is more sensitive against round-off than the other variants.
2. If the distribution function is given, we can transform the non-uniform random variates into uniform ones. Obviously this is the most expensive method as the evaluation of distribution functions is usually (much) more expensive than drawing a random number.
3. A hybrid of the these two approaches works as follows: Draw one small random sample (i.e., the first of the requested repetitions). Use the requested break points in u-scale and compute the corresponding empirical quantiles. This we get break points in x-scale as in Variant 1. Then compute the expected probabilities for each bin by means of the distribution function. This variant is as fast as Variant 1 but is less sensitive against round-off errors. On the other hand, the bin frequencies are slightly changed compared to the requested ones.

The next example shows the difference between Variant 1 and 3 (see Fig. 4):

Example

```
> ##   Create and plot frequency table.
> ## ....................................................................
> #
> ## Variant 1: use qnorm
> ft1 <- rvgt.ftable(n=1e4, rep=10, rdist=rnorm, qdist=qnorm, plot=TRUE)
> ## Variant 3: use pnorm
> ft2 <- rvgt.ftable(n=1e4, rep=10, rdist=rnorm, pdist=pnorm, plot=TRUE)
```



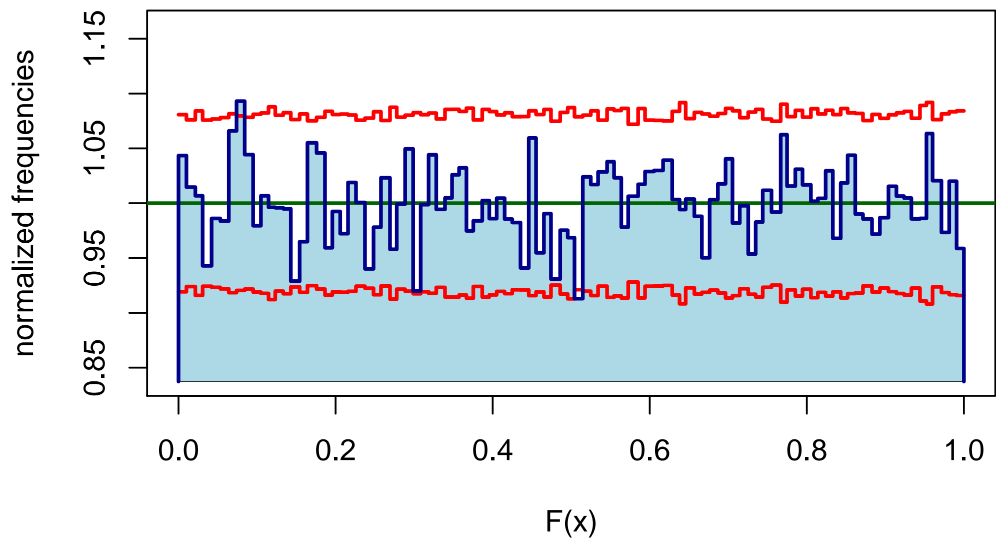


Figure 4: Histogram of a sample of normal distributed random variates. first line: Variant 1. second line: Variant 3.

2.3 Goodness-of-Fit Tests

Once the frequency table is available one can run one or more goodness-of-fit tests on it. Currently the following tests are implemented:

- Chisquare goodness-of-fit test.
- M-Test (adjusted residual test) by Fuchs and Kenett [4].

The plotting routine allows to compare the result of several tests (which need not be for the same random number generator). Here is an example that performs the above tests and plots the results (see Fig. 5):

Example

```
> ##   Perform goodness-of-fit tests
> ## ......................................................................
> #
> ## Create a frequency table.
> ft <- rvgt.ftable(n=1e5, rep=20, rdist=rnorm, qdist=qnorm)
> #
> ## Perform goodness-of-fit tests
> res.chisq <- rvgt.chisq(ft)
> res.mtest <- rvgt.Mtest(ft)
> #
> ## Show results in one plot
> plot.rvgt.htest(list(res.chisq, res.mtest))
```

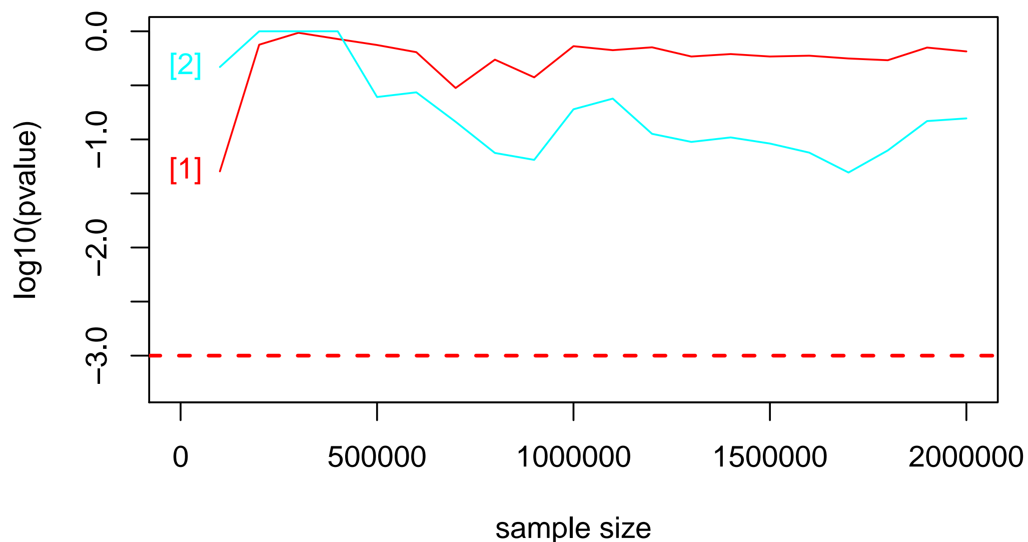


Figure 5: p-values for goodness-of-fit tests: chisquare test [1] and M-test [2].

2.4 A Defective Random Variate Generator

R provides a defective normal random variate generator called `"Buggy Kinderman-Ramage"`[3]. So let us run our routines on this generator (see Figs. 6 and 7):

Example

```
> ##   A buggy Gaussian random variate generator
> ## ......................................................................
> #
> ## Use a buggy Gaussian random variate generator
> RNGkind(normal.kind="Buggy Kinderman-Ramage")
> #
> ## Create a frequency table.
> ft <- rvgt.ftable(n=1e5, rep=50, rdist=rnorm, qdist=qnorm)
> #
> ## Plot histogram (see Fig. 6)
> plot(ft)
> #
> ## Perform goodness-of-fit tests
> res.chisq <- rvgt.chisq(ft)
> res.mtest <- rvgt.Mtest(ft)
> #
> ## Show results of tests (see Fig. 7)
> plot.rvgt.htest(list(res.chisq, res.mtest))
```

The effect of increasing sample can be visualized by the following piece of code (see Fig. 8):

Example

```
> ##   Effect of increasing sample sizes
> ## ......................................................................
> #
> ## Create frequency table for buggy generator.
> RNGkind(normal.kind="Buggy Kinderman-Ramage")
> ft <- rvgt.ftable(n=1e5, rep=50, rdist=rnorm, qdist=qnorm)
> #
> ## Now plot histograms for the respective sample sizes
> ## 1e5, 10*1e5, and 50*1e5.
> plot(ft,rows=1)
> plot(ft,rows=1:10)
> plot(ft,rows=1:50)
```

2.5 Perturb a Random Variate Generator

The power is an important feature of a test. We need to know which deviation from the theoretical distribution can be detected and at which sample size will the defect become significant. Thus we have added functions `pertadd()` and `pertsub()` to perturb existing random variate generators. Thus one can investigate the influence known deviations.

[3] See [8] for some background.

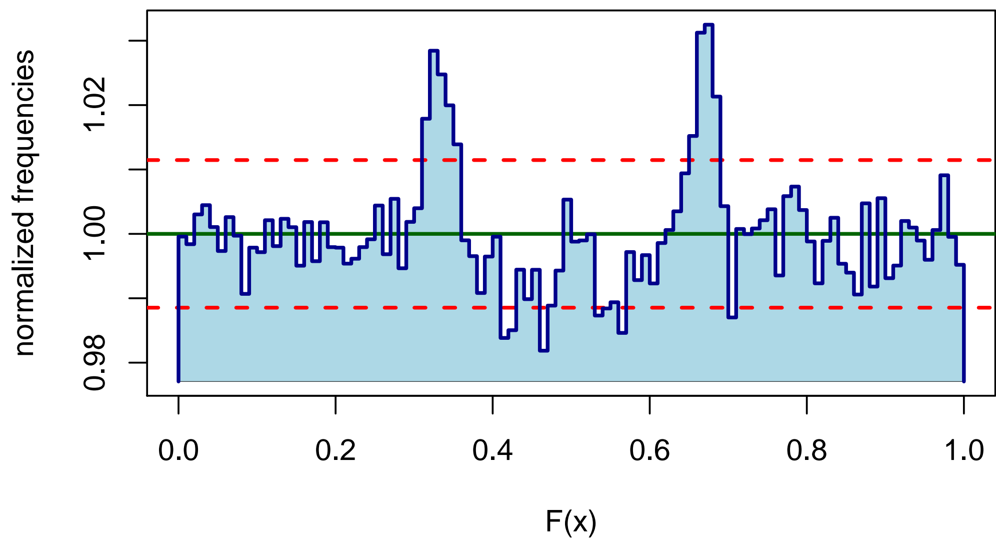


Figure 6: Histogram of a defective sample of normal distributed random variates (`"Buggy Kinderman-Ramage"`).

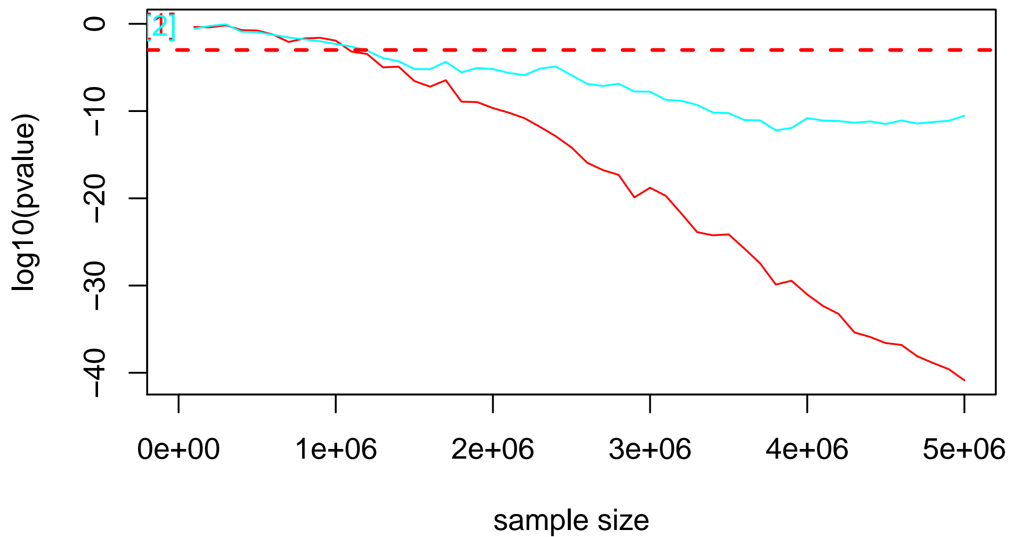


Figure 7: p-values for goodness-of-fit tests for the `"Buggy Kinderman-Ramage"` generator: chisquare test [1,red] and M-test [2,cyan].

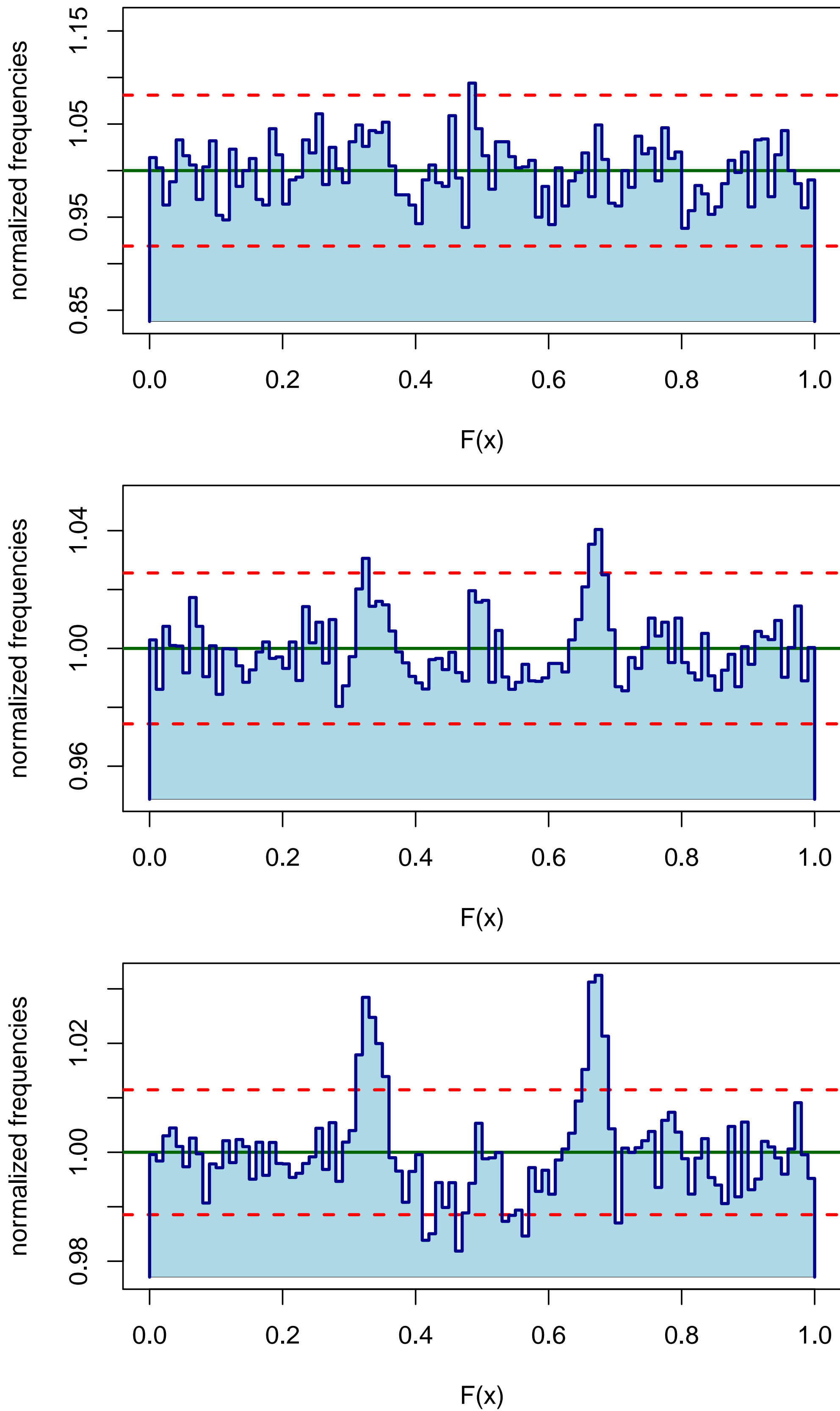


Figure 8: Histogram of a defect-iv sample of normal distributed random variates (`"Buggy Kinderman-Ramage"`) for increasing sample sizes.

3 Approximation Errors in Numeric Inversion

Random variate generators that are based on inverting the distribution function preserve the structure of the underlying uniform random number generator. Given the fact that state-of-the-art uniform random number generators are well tested, it is sufficient to estimate (maximal) approximation errors.

As for the histogram based methods computing, plotting and analyzing of such errors can be quite time consuming. So the numeric error of an approximate inverse distribution function $G^{-1}(u)$ is often a very volatile function of u and requires the computation at a lot of points. For plotting we can condense the information by partitioning $(0,1)$ into intervals of equal length. In each of these the error is computed at equidistributed points and minimum, maximum, median and quartiles are estimated and stored. Thus we save memory and it is much faster to plot and compare errors for different methods or distributions.

We have implemented a two step procedure:

1. Create a table of quantiles of the error in each of the intervals.

2. Visualize errors.

Thus we again can compare error of different approximation methods.

3.1 Measures for Approximation Errors

Let F and F^{-1} denote cumulative distribution function (CDF) and inverse distribution function (quantile function). Let G^{-1} be an approximation for F^{-1}.

Then the following measures may be used for describing approximation errors.

u-error: [**recommended**]

$$\epsilon_u(u) = |u - F(G^{-1}(u))|$$

absolute x-error:

$$\epsilon_x(u) = |F^{-1}(u) - G^{-1}(u)|$$

relative x-error:

$$\tilde{\epsilon}_x(u) = \frac{|F^{-1}(u) - G^{-1}(u)|}{|F^{-1}(u)|}$$

We have implemented also three errors but we convinced that it is the u-error that is best suited in the framework of Monte Carlo simulation (MC) and quasi-Monte Carlo methods (QMC).

It has some properties that make it a convenient and practical relevant measure of error for numerical inversion:

- Uniform pseudo-random number generators work with integer arithmetic and return points on a grid. Thus these pseudo-random points have a limited resolution, typically $2^{-32} \approx 2.3 \times 10^{-10}$ or (less frequently) machine precision $2^{-52} \approx 2.2 \times 10^{-16}$. Consequently, the positions of pseudo-random numbers U are not random at all at a scale that is not much larger than their resolution. u-errors can be seen as minor deviations of the underlying uniform pseudo-random points U_i from their "correct" positions. We consider this deviation as negligible if it is (much) smaller than the resolution of the pseudo-random variate generator.

- The same holds for QMC experiments where the F-discrepancy [3, 9] of a point set $\{X_i\}$ is computed as discrepancy of the set $\{F(X_i)\}$. If the X_i are generated by exact inversion

their F-discrepancy coincides with the discrepancy of the underlying low-discrepancy set. Thus $\varepsilon_u(u)$ can be used to estimate the maximal change of the F-discrepancy compared to the "exact" points.

- Consider a sequence of approximations F_n^{-1} to the inverse CDF F^{-1} such that $\varepsilon_{u,n}(u) < \frac{1}{n}$ and let F_n be the corresponding CDF. Then $|F(x) - F_n(x)| = |F(F_n^{-1}(u)) - F_n(F_n^{-1}(u))| = |F(F_n^{-1}(u)) - u| = \varepsilon_{u,n}(u) \to 0$ for $n \to \infty$. That is, the CDFs F_n converge weakly to the CDF F of the target distribution and the corresponding random variates $X_n = F_n^{-1}(U)$ converge in distribution to the target distribution [1].

- $\varepsilon_u(u)$ can easily be computed provided that we can compute F sufficiently accurately.

We are therefore convinced that the u-error is a natural concept for the approximation error of numerical inversion.

3.2 u-Error

Here is a simple example. It shows median, interquartile range and range (minimum and maximum) of the u-error in 100 equidistributed intervals in u-scale, see Fig. 9. Of course it its possible to increase the number of these intervals by setting argument `res` to, e.g., `1000`, see Fig. 10.

Example

```
> ##   u-error
> ## ......................................................................
> #
> ## Approximate inverse CDF of normal distribution using splines.
> aqn <- splinefun(x=pnorm((-100:100)*0.05), y=(-100:100)*0.05,
+                  method="monoH.FC")
> #
> ## Create a table of u-errors for the approximation errors.
> ## Use a sample of size of 10^5 random variates and
> ## a resolution of only 100 points.
> ue <- uerror(n=1e5, res=100, aqdist=aqn, pdist=pnorm)
> #
> ## Plot u-errors.
> plot(ue)
```

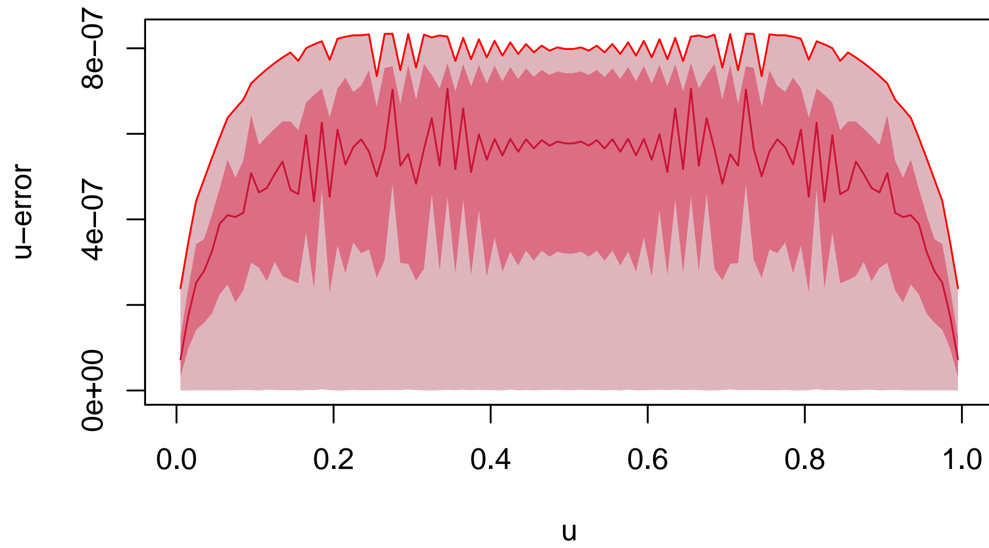


Figure 9: u-error of an approximate inverse CDF (using splines) for the normal distribution; resolution = 100.

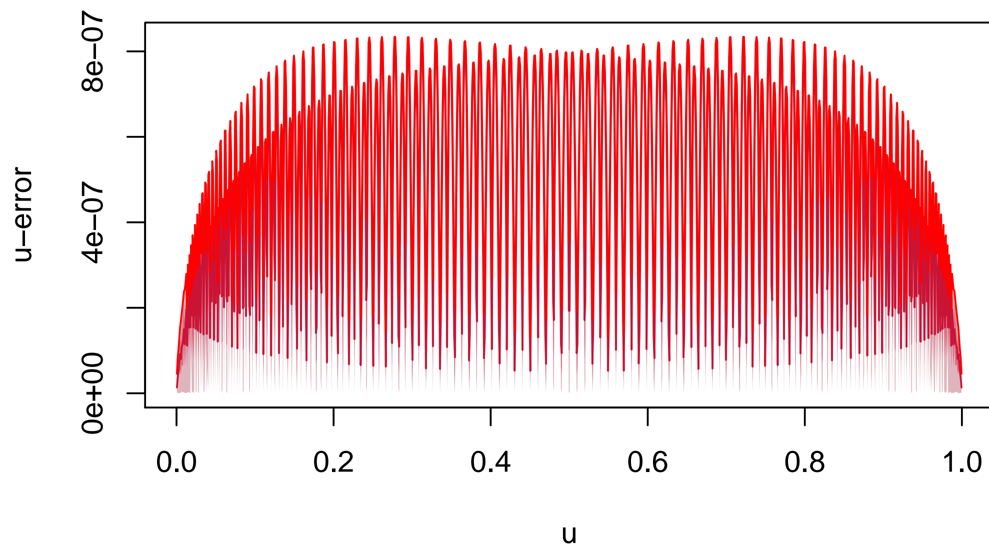


Figure 10: u-error of an approximate inverse CDF (using splines) for the normal distribution; resolution = 1000.

On the other hand it is also possible to zoom into a particular subdomain of the inverse CDF using argument `udomain` (see Fig. 11):

Example

```
> ##   u-error
> ## ......................................................................
> #
> ## Approximate inverse CDF of normal distribution using splines.
> aqn <- splinefun(x=pnorm((-100:100)*0.05), y=(-100:100)*0.05,
+                  method="monoH.FC")
> #
> ## Create a table of u-errors for the approximation errors
> ## for the subdomain (0.6, 0.65).
> ## Use a sample of size of 10^5 random variates.
> ue <- uerror(n=1e5, aqdist=aqn, pdist=pnorm, udomain=c(0.6,0.65))
> plot(ue)
```

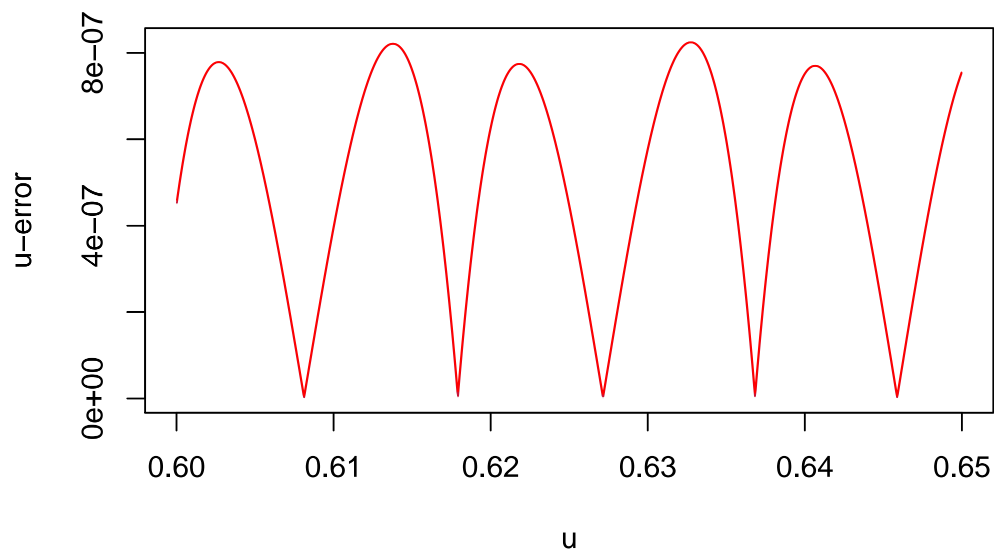


Figure 11: u-error of an approximate inverse CDF (using splines) for the normal distribution on interval $(0.6, 0.65)$; resolution = 1000.

It is also possible to show the u-errors of more than one approximation in one plot (see Fig. 12). The argument `tol` inserts the dashed line in the plot and can be used to indicate the maximal tolerated error.

Example

```
> ##   u-error
> ## ..............................................................
> #
> ## Approximate inverse CDF of normal and gamma distribution using splines.
> aqn <- splinefun(x=pnorm((-100:100)*0.05), y=(-100:100)*0.05,
+                  method="monoH.FC")
> aqg <- splinefun(x=pgamma((0:500)*0.1,shape=5),
+                  y=(0:500)*0.1, method="monoH.FC")
> #
> ## Compute u-errors for these approximations
> uen <- uerror(n=1e5, aqdist=aqn, pdist=pnorm)
> ueg <- uerror(n=1e5, aqdist=aqg, pdist=pgamma, shape=5)
> #
> ## Plot u-errors
> plot.rvgt.ierror(list(uen,ueg), tol=1.e-6)
```

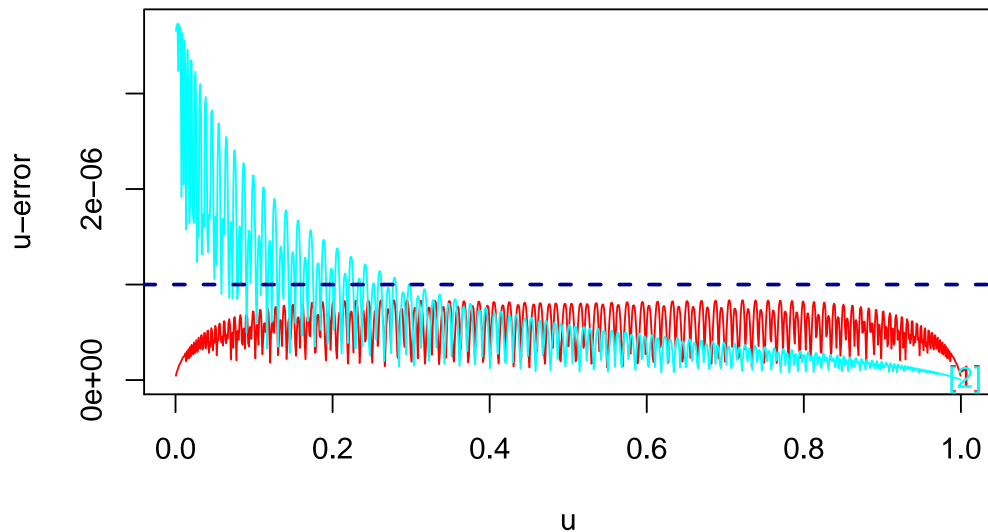


Figure 12: u-error of an approximate inverse CDF (using splines) for the normal [1,red] and gamma distribution [2,cyan]; resolution = 1000.

3.3 x-Error

We are convinced that x-errors (absolute and relative) are not the measure of choice for approximation errors of for numerical inverse in the framework of MC and QMC methods. Nevertheless, there are two examples how these errors can be computed.

Example

```
> ##   Absolute x-error
> ## ..............................................................
> #
> ## Approximate inverse CDF of normal distribution using splines.
> aqn <- splinefun(x=pnorm((-100:100)*0.05), y=(-100:100)*0.05,
+                  method="monoH.FC")
> #
> ## Create a table of absolute x-errors.
> xea <- xerror(n=1e5, aqdist=aqn, qdist=qnorm, kind="abs")
> #
> ## Plot x-errors
> plot(xea)
```

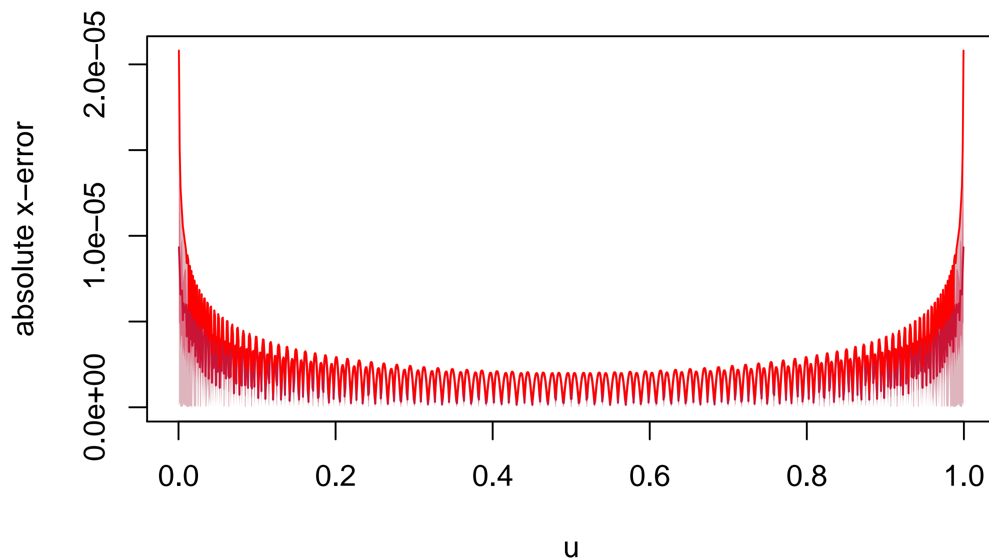


Figure 13: Absolute x-error of an approximate inverse CDF (using splines) for the normal distribution; resolution = 1000.

Example

```
> ##   Relative x-error
> ## .....................................................................
> #
> ## Approximate inverse CDF of normal distribution using splines.
> aqn <- splinefun(x=pnorm((-100:100)*0.05), y=(-100:100)*0.05,
+                  method="monoH.FC")
> #
> ## Create a table of relative x-errors.
> xer <- xerror(n=1e5, aqdist=aqn, qdist=qnorm, kind="rel")
> #
> ## Plot x-errors
> plot(xer)
```

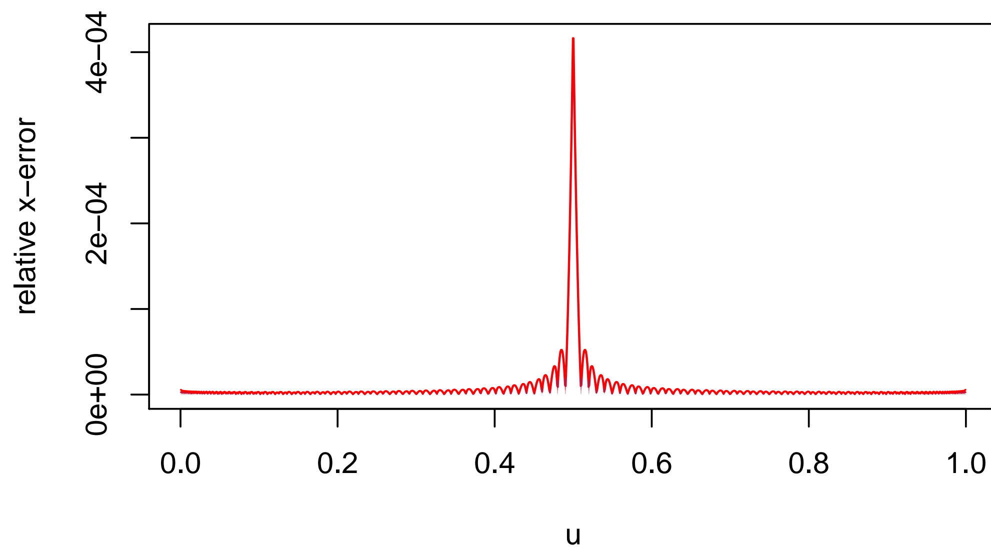


Figure 14: Relative x-error of an approximate inverse CDF (using splines) for the normal distribution; resolution = 1000.

References

[1] P. Billingsley. *Probability and Measure.* Wiley & Sons, New York, 1986.

[2] L. Devroye. *Non-Uniform Random Variate Generation.* Springer-Verlag, New-York, 1986.

[3] K.-T. Fang and Y. Wang. *Number-theoretic Methods in Statistics*, volume 51 of *Monographs on Statistics and Applied Probability.* Chapman and Hall, London, 1994.

[4] C. Fuchs and R. Kenett. A test for detecting outlying cells in the multinomial distribution and two-way contingency tables. *J. Am. Stat. Assoc.*, 75:395–398, 1980.

[5] W. Hörmann, J. Leydold, and G. Derflinger. *Automatic Nonuniform Random Variate Generation.* Springer-Verlag, Berlin Heidelberg, 2004.

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

[7] M. L. Overton. *Numerical Computing with IEEE Floating Point Arithmetic.* SIAM, Philadelphia, 2001.

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

[9] B. Tuffin. *Simulation accélérée par les méthodes de Monte Carlo et quasi-Monte Carlo: théorie et applications.* PhD thesis, Université de Rennes 1, 1997.