Contents

rbeta(a,b) beta(a,b) random variates, where a and b are the beta distribution shape parameters
rbinomial(n,p) binomial(n,p) random variates, where n is the number of trials and p is the success probability
rcauchy(a,b) Cauchy(a,b) random variates, where a is the location parameter and b is the scale parameter
rchisq(df) chi-squared, with df degrees of freedom, random variates
rcauchy(a,b) exponential random variates with scale b
rgamma(a,b) gamma(a,b) random variates, where a is the gamma shape parameter and b is the scale parameter
rhypergeometric(N,K,n) hypergeometric random variates
rigaussian(m,a) inverse Gaussian random variates with mean m and shape parameter a
rlaplace(m,b) Laplace(m,b) random variates with mean m and scale parameter b
rlogistic() logistic variates with mean 0 and standard deviation \( \pi/\sqrt{3} \)
rlogistic(s) logistic variates with mean 0, scale s, and standard deviation \( s\pi/\sqrt{3} \)
rlogistic(m,s) logistic variates with mean m, scale s, and standard deviation \( s\pi/\sqrt{3} \)nbinomial(n,p) negative binomial (Gaussian) random variates, that is, variates from a normal distribution with a mean of 0 and a standard deviation of 1
rnormal() standard normal (Gaussian) random variates, where \( m \) is the mean and the standard deviation is 1
rnormal(m) normal(m,1) (Gaussian) random variates, where \( m \) is the mean and the standard deviation is 1
rnormal(m,s) normal(m,s) (Gaussian) random variates, where \( m \) is the mean and \( s \) is the standard deviation
rpoisson(m) Poisson(m) random variates, where \( m \) is the distribution mean
rt(df) Student’s t random variates, where df is the degrees of freedom
runiform() uniformly distributed random variates over the interval (0, 1)
runiform(a,b) uniformly distributed random variates over the interval \( (a, b) \)
runiformint(a,b) uniformly distributed random integer variates on the interval \( [a, b] \)
**Functions**

The term “pseudorandom number” is used to emphasize that the numbers are generated by formulas and are thus not truly random. From now on, we will drop the “pseudo” and just say random numbers.

For information on setting the random-number seed, see [R] set seed.

---

**runiform()**

Description: uniformly distributed random variates over the interval \((0, 1)\)

- runiform() can be seeded with the set seed command; see [R] set seed.
- Range: \(c(\text{epsdouble})\) to \(1 - c(\text{epsdouble})\)

**runiform(a, b)**

Description: uniformly distributed random variates over the interval \((a, b)\)

- Domain \(a\): \(c(\text{mindouble})\) to \(c(\text{maxdouble})\)
- Domain \(b\): \(c(\text{mindouble})\) to \(c(\text{maxdouble})\)
- Range: \(a + c(\text{epsdouble})\) to \(b - c(\text{epsdouble})\)

**runiformint(a, b)**

Description: uniformly distributed random integer variates on the interval \([a, b]\)

- If \(a\) or \(b\) is nonintegral, runiformint(a,b) returns runiformint(floor(a), floor(b)).
- Domain \(a\): \(-2^{53}\) to \(2^{53}\) (may be nonintegral)
- Domain \(b\): \(-2^{53}\) to \(2^{53}\) (may be nonintegral)
- Range: \(-2^{53}\) to \(2^{53}\)

**rbeta(a, b)**

Description: beta(a,b) random variates, where \(a\) and \(b\) are the beta distribution shape parameters

Besides using the standard methodology for generating random variates from a given distribution, rbeta() uses the specialized algorithms of Johnk (Gentle 2003), Atkinson and Whittaker (1970, 1976), Devroye (1986), and Schmeiser and Babu (1980).

- Domain \(a\): \(0.05\) to \(1e+5\)
- Domain \(b\): \(0.15\) to \(1e+5\)
- Range: \(0\) to \(1\) (exclusive)
**rbinomial(n,p)**

Description: binomial(n,p) random variates, where n is the number of trials and p is the success probability

Besides using the standard methodology for generating random variates from a given distribution, rbinomial() uses the specialized algorithms of Kachitvichyanukul (1982), Kachitvichyanukul and Schmeiser (1988), and Kemp (1986).

Domain n: 1 to 1e+11
Domain p: 1e–8 to 1–1e–8
Range: 0 to n

**rcauchy(a,b)**

Description: Cauchy(a,b) random variates, where a is the location parameter and b is the scale parameter

Domain a: $-1e+300$ to $1e+300$
Domain b: $1e–100$ to $1e+300$
Range: $c(\text{mindouble})$ to $c(\text{maxdouble})$

**rchisq(df)**

Description: chi-squared, with df degrees of freedom, random variates

Domain df: $2e–4$ to $2e+8$
Range: 0 to $c(\text{maxdouble})$

**rexponential(b)**

Description: exponential random variates with scale b

Domain b: $1e–323$ to $8e+307$
Range: $1e–323$ to $8e+307$

**rgamma(a,b)**

Description: gamma(a,b) random variates, where a is the gamma shape parameter and b is the scale parameter

Methods for generating gamma variates are taken from Ahrens and Dieter (1974), Best (1983), and Schmeiser and Lal (1980).

Domain a: $1e–4$ to $1e+8$
Domain b: $c(\text{smallestdouble})$ to $c(\text{maxdouble})$
Range: 0 to $c(\text{maxdouble})$

**rhypergeometric(N,K,n)**

Description: hypergeometric random variates

The distribution parameters are integer valued, where N is the population size, K is the number of elements in the population that have the attribute of interest, and n is the sample size.

Besides using the standard methodology for generating random variates from a given distribution, rhypergeometric() uses the specialized algorithms of Kachitvichyanukul (1982) and Kachitvichyanukul and Schmeiser (1985).

Domain N: 2 to $1e+6$
Domain K: 1 to $N–1$
Domain n: 1 to $N–1$
Range: $\max(0,n – N + K)$ to $\min(K,n)$
4 Random-number functions

rigaussian\((m, a)\)
Description: inverse Gaussian random variates with mean \(m\) and shape parameter \(a\)

\[ \text{rigaussian()} \text{ is based on a method proposed by Michael, Schucany, and Haas (1976).} \]
Domain \(m\): 1e–10 to 1000
Domain \(a\): 0.001 to 1e+10
Range: 0 to \(c\)(maxdouble)

rlaplace\((m, b)\)
Description: Laplace\((m, b)\) random variates with mean \(m\) and scale parameter \(b\)

Domain \(m\): \(-1e+300\) to \(1e+300\)
Domain \(b\): \(1e–300\) to \(1e+300\)
Range: \(c\)(mindouble) to \(c\)(maxdouble)

rlogistic()
Description: logistic variates with mean 0 and standard deviation \(\pi/\sqrt{3}\)

The variates \(x\) are generated by \(x = \text{invlogistic}(0, 1, u)\), where \(u\) is a random uniform(0,1) variate.
Range: \(c\)(mindouble) to \(c\)(maxdouble)

rlogistic\((s)\)
Description: logistic variates with mean 0, scale \(s\), and standard deviation \(s\pi/\sqrt{3}\)

The variates \(x\) are generated by \(x = \text{invlogistic}(0, s, u)\), where \(u\) is a random uniform(0,1) variate.
Domain \(s\): 0 to \(c\)(maxdouble)
Range: \(c\)(mindouble) to \(c\)(maxdouble)

rlogistic\((m, s)\)
Description: logistic variates with mean \(m\), scale \(s\), and standard deviation \(s\pi/\sqrt{3}\)

The variates \(x\) are generated by \(x = \text{invlogistic}(m, s, u)\), where \(u\) is a random uniform(0,1) variate.
Domain \(m\): \(c\)(mindouble) to \(c\)(maxdouble)
Domain \(s\): 0 to \(c\)(maxdouble)
Range: \(c\)(mindouble) to \(c\)(maxdouble)

rnbinomial\((n, p)\)
Description: negative binomial random variates

If \(n\) is integer valued, \(\text{rnbinomial()}\) returns the number of failures before the \(n\)th success, where the probability of success on a single trial is \(p\). \(n\) can also be nonintegral.
Domain \(n\): 1e–4 to 1e+5
Domain \(p\): 1e–4 to 1–1e–4
Range: 0 to \(2^{53} - 1\)
rnormal()
Description: standard normal (Gaussian) random variates, that is, variates from a normal distribution with a mean of 0 and a standard deviation of 1
Range: c(mindouble) to c(maxdouble)

rnormal(m)
Description: normal(m,1) (Gaussian) random variates, where \(m\) is the mean and the standard deviation is 1
Domain \(m\): c(mindouble) to c(maxdouble)
Range: c(mindouble) to c(maxdouble)

rnormal(m,s)
Description: normal(m,s) (Gaussian) random variates, where \(m\) is the mean and \(s\) is the standard deviation.
The methods for generating normal (Gaussian) random variates are taken from Knuth (1998, 122–128); Marsaglia, MacLaren, and Bray (1964); and Walker (1977).
Domain \(m\): c(mindouble) to c(maxdouble)
Domain \(s\): 0 to c(maxdouble)
Range: c(mindouble) to c(maxdouble)

rpoisson(m)
Description: Poisson(m) random variates, where \(m\) is the distribution mean
Poisson variates are generated using the probability integral transform methods of Kemp and Kemp (1990, 1991) and the method of Kachitvichyanukul (1982).
Domain \(m\): 1e–6 to 1e+11
Range: 0 to \(2^{53} – 1\)

rt(df)
Description: Student’s \(t\) random variates, where \(df\) is the degrees of freedom
Student’s \(t\) variates are generated using the method of Kinderman and Monahan (1977, 1980).
Domain \(df\): 1 to \(2^{53} – 1\)
Range: c(mindouble) to c(maxdouble)

rweibull(a,b)
Description: Weibull variates with shape \(a\) and scale \(b\)
The variates \(x\) are generated by \(x = \text{invweibulltail}(a, b, 0, u)\), where \(u\) is a random uniform(0,1) variate.
Domain \(a\): 0.01 to 1e+6
Domain \(b\): 1e–323 to 8e+307
Range: 1e–323 to 8e+307
rweibull($a, b, g$)
Description: Weibull variates with shape $a$, scale $b$, and location $g$
The variates $x$ are generated by $x = \text{invweibulltail}(a, b, g, u)$, where $u$ is a random uniform(0,1) variate.
Domain $a$: 0.01 to 1e+6
Domain $b$: 1e–323 to 8e+307
Domain $g$: $-8e+307$ to $8e+307$
Range: $g + c(\text{epsdouble})$ to 8e+307

rweibullph($a, b$)
Description: Weibull (proportional hazards) variates with shape $a$ and scale $b$
The variates $x$ are generated by $x = \text{invweibullphtail}(a, b, 0, u)$, where $u$ is a random uniform(0,1) variate.
Domain $a$: 0.01 to 1e+6
Domain $b$: 1e–323 to 8e+307
Range: 1e–323 to 8e+307

rweibullph($a, b, g$)
Description: Weibull (proportional hazards) variates with shape $a$, scale $b$, and location $g$
The variates $x$ are generated by $x = \text{invweibullphtail}(a, b, g, u)$, where $u$ is a random uniform(0,1) variate.
Domain $a$: 0.01 to 1e+6
Domain $b$: 1e–323 to 8e+307
Domain $g$: $-8e+307$ to $8e+307$
Range: $g + c(\text{epsdouble})$ to 8e+307

Remarks and examples

It is ironic that the first thing to note about random numbers is how to make them reproducible. Before using a random-number function, type

```
set seed #
```
where # is any integer between 0 and $2^{31} - 1$, inclusive, to draw the same sequence of random numbers. It does not matter which integer you choose as your seed; they are all equally good. See [R] set seed.

`runiform()` is the basis for all the other random-number functions because all the other random-number functions transform uniform (0,1) random numbers to the specified distribution.

`runiform()` implements the 64-bit Mersenne Twister (mt64), the stream 64-bit Mersenne Twister (mt64s), and the 32-bit “keep it simple stupid” (kiss32) random-number generators (RNGs) for generating uniform (0,1) random numbers. `runiform()` uses the mt64 RNG by default.

`runiform()` uses the kiss32 RNG only when the user version is less than 14 or when the RNG has been set to kiss32; see [P] version for details about setting the user version. We recommend that you do not change the default RNG, but see [R] set rng for details.
Technical note

Although we recommend that you use `runiform()`, we made generator-specific versions of `runiform()` available for advanced users who want to hardcode their generator choice. The function `runiform_mt64()` always uses the mt64 RNG to generate uniform (0, 1) random numbers, the function `runiform_mt64s()` always uses the mt64s RNG to generate uniform (0, 1) random numbers, the function `runiform_kiss32()` always uses the kiss32 RNG to generate uniform (0, 1) random numbers. In fact, generator-specific versions are available for all the implemented distributions. For example, `rnormal_mt64()`, `rnormal_mt64s`, and `rnormal_kiss32()` use transforms of mt64, mt64s, and kiss32 uniform variates, respectively, to generate standard normal variates.

Technical note

Both the mt64 and the kiss32 RNGs produce uniform variates that pass many tests for randomness. Many researchers prefer the mt64 to the kiss32 RNG because the mt64 generator has a longer period and a finer resolution and requires a higher dimension before patterns appear; see Matsumoto and Nishimura (1998).

The mt64 RNG has a period of $2^{19937} - 1$ and a resolution of $2^{-53}$; see Matsumoto and Nishimura (1998). Each stream of the mt64s RNG contains $2^{128}$ random numbers, and mt64s has a resolution of $2^{-53}$; see Haramoto et al. (2008). The kiss32 RNG has a period of about $2^{126}$ and a resolution of $2^{-32}$; see Methods and formulas below.

Technical note

This technical note explains how to restart a RNG from its current spot.

The current spot in the sequence of a RNG is part of the state of a RNG. If you tell me the state of a RNG, I know where it is in its sequence, and I can compute the next random number. The state of a RNG is a complicated object that requires more space than the integers used to seed a generator. For instance, an mt64 state is a 5011-digit, base-16 number preceded by three letters.

If you want to restart a RNG from where it left off, you should store the current state in a macro and then set the state of the RNG when you want to restart it. For example, suppose we set a seed and draw some random numbers.

```
. set obs 3
        number of observations (N) was 0, now 3
. set seed 12345
. generate x = runiform()
. list x
    +--------+
    | x      |
    |--------|
    | 1.00    |
    | 2.00    |
    | 3.00    |
    +--------+
```

For instance, an mt64 state is a 5011-digit, base-16 number preceded by three letters.
We store the state of the RNG so that we can pick up right here in the sequence.

```
. local rngstate "c(rngstate)"
```

We draw some more random numbers.

```
. replace x = runiform()
(3 real changes made)
. list x
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>.5597356</td>
</tr>
<tr>
<td>2.</td>
<td>.5744513</td>
</tr>
<tr>
<td>3.</td>
<td>.2076905</td>
</tr>
</tbody>
</table>

Now, we set the state of the RNG to where it was and draw those same random numbers again.

```
. set rngstate 'rngstate'
. replace x = runiform()
(0 real changes made)
. list x
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>.5597356</td>
</tr>
<tr>
<td>2.</td>
<td>.5744513</td>
</tr>
<tr>
<td>3.</td>
<td>.2076905</td>
</tr>
</tbody>
</table>

## Methods and formulas

All the nonuniform generators are based on the uniform mt64, mt64s, and kiss32 RNGs.

The mt64 RNG is well documented in Matsumoto and Nishimura (1998) and on their website [http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html](http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html). The mt64 RNG implements the 64-bit version discussed at [http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt64.html](http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt64.html). The mt64s RNG is based on a method proposed by Haramoto et al. (2008). The default seed of all three RNGs is 123456789.

### kiss32 generator

The kiss32 uniform RNG implemented in `runiform()` is based on George Marsaglia’s (G. Marsaglia, 1994, pers. comm.) 32-bit pseudorandom-integer generator kiss32. The integer kiss32 RNG is composed of two 32-bit pseudorandom-integer generators and two 16-bit integer generators (combined to make one 32-bit integer generator). The four generators are defined by the recursions

\[
\begin{align*}
x_n &= 69069x_{n-1} + 1234567 \mod 2^{32} \\
y_n &= y_{n-1}(I + L^{13})(I + R^{17})(I + L^{5}) \\
z_n &= 65184(z_{n-1} \mod 2^{16}) + \text{int}(z_{n-1}/2^{16}) \\
w_n &= 63663(w_{n-1} \mod 2^{16}) + \text{int}(w_{n-1}/2^{16})
\end{align*}
\]
In (2), the 32-bit word $y_n$ is viewed as a $1 \times 32$ binary vector; $L$ is the $32 \times 32$ matrix that produces a left shift of one ($L$ has 1s on the first left subdiagonal, 0s elsewhere); and $R$ is $L$ transpose, affecting a right shift by one. In (3) and (4), $\text{int}(x)$ is the integer part of $x$.

The integer kiss32 RNG produces the 32-bit random integer

$$R_n = x_n + y_n + z_n + 2^{16} w_n \mod 2^{32}$$

The kiss32 uniform RNG implemented in runiform() takes the output from the integer kiss32 RNG and divides it by $2^{32}$ to produce a real number on the interval $(0, 1)$. (Zeros are discarded, and the first nonzero result is returned.)

The recursion (5)–(8) have, respectively, the periods

$$2^{32}$$

$$2^{32} - 1$$

$$(65184 \cdot 2^{16} - 2)/2 \approx 2^{31}$$

$$(63663 \cdot 2^{16} - 2)/2 \approx 2^{31}$$

Thus the overall period for the integer kiss32 RNG is

$$2^{32} \cdot (2^{32} - 1) \cdot (65184 \cdot 2^{15} - 1) \cdot (63663 \cdot 2^{15} - 1) \approx 2^{126}$$

When Stata first comes up, it initializes the four recursions in kiss32 by using the seeds

$$x_0 = 123456789$$

$$y_0 = 521288629$$

$$z_0 = 362436069$$

$$w_0 = 2262615$$

Successive calls to the kiss32 uniform RNG implemented in runiform() then produce the sequence

$$R_1^{\prime} \cdot 2^{32} \cdot R_2^{\prime} \cdot 2^{32} \cdot R_3^{\prime} \cdot 2^{32} \cdot \cdots$$

Hence, the kiss32 uniform RNG implemented in runiform() gives the same sequence of random numbers in every Stata session (measured from the start of the session) unless you reinitialize the seed. The full seed is the set of four numbers $(x, y, z, w)$, but you can reinitialize the seed by simply issuing the command

$$\text{set seed } #$$

where # is any integer between 0 and $2^{31} - 1$, inclusive. When this command is issued, the initial value $x_0$ is set equal to #, and the other three recursions are restarted at the seeds $y_0$, $z_0$, and $w_0$ given above. The first 100 random numbers are discarded, and successive calls to the kiss32 uniform RNG implemented in runiform() give the sequence

$$R_{101}^{\prime} \cdot 2^{32} \cdot R_{102}^{\prime} \cdot 2^{32} \cdot R_{103}^{\prime} \cdot 2^{32} \cdot \cdots$$
However, if the command

```
.set seed 123456789
```

is given, the first 100 random numbers are not discarded, and you get the same sequence of random numbers that the kiss32 RNG produces when Stata restarts; also see [R] set seed.

## Acknowledgments

We thank the late George Marsaglia, formerly of Florida State University, for providing his kiss32 RNG.

We thank John R. Gleason (retired) of Syracuse University for directing our attention to Wichura (1988) for calculating the cumulative normal density accurately, for sharing his experiences about techniques with us, and for providing C code to make the calculations.

We thank Makoto Matsumoto and Takuji Nishimura for deriving the Mersenne Twister and distributing their code for their generator so that it could be rapidly and effectively tested.

## References


Also see

[FN] Functions by category

[D] egen — Extensions to generate

[D] generate — Create or change contents of variable

[R] set rng — Set which random-number generator (RNG) to use

[R] set rngstream — Specify the stream for the stream random-number generator

[R] set seed — Specify random-number seed and state

[M-5] runiform() — Uniform and nonuniform pseudorandom variates

[U] 13.3 Functions