

Using **RcppTN** in R and C++

Jonathan Olmsted
jpolmsted@gmail.com

This version updated: November 21, 2017

This brief document shows simple usage of the function `rtn()` provided by the R package **RcppTN** for drawing from an arbitrary sequence of truncated Normal distributions. Much of the value added by the **RcppTN** package comes from providing a C++-level API to call in development of other **Rcpp**-based C++ codes. Use of this API is also demonstrated. While no other R packages currently provide this functionality in an API, some existing implementations for drawing from a truncated Normal distribution at the R-level include **truncnorm** and **msm**.

1 Installation

Currently, there is no CRAN version of the package, so the simplest installation mechanism is using the `install_github()` function from the `devtools` package.

```
library("devtools")
install_github(repo = "RcppTN",
               username = "olmjo",
               subdir = "pkg",
               ref = "development"
)
```

2 R-level Usage

2.1 RNG

Usage of the `rtn()` function in R is straightforward (albeit not feature-rich). Without any options, we get a single draw from the standard Normal distribution. And, this draw respects R's RNG state so the stream of output is reproducible.

```
library("RcppTN")
set.seed(1)
rtn()

## [1] -0.6264538

set.seed(1)
rtn()

## [1] -0.6264538
```

Under this implementation of the Robert (1995) algorithm, a request for a single draw from a Standard Normal distribution truncated from $-\infty$ to ∞ — the default behavior of the function when called without any arguments — results in the same return value as a single draw from a Standard Normal distribution using `rnorm()`. This is just a by-product of the implementation and holds no practical significance.¹

```
set.seed(1)
rtn()

## [1] -0.6264538

set.seed(1)
rtn(.mean = 0, .sd = 1, .low = -Inf, .high = Inf)

## [1] -0.6264538

set.seed(1)
rtn()

## [1] -0.6264538

set.seed(1)
rnorm(1)

## [1] -0.6264538
```

Of course, `rtn()`'s behavior given RNG seeds is exactly as you would expect for any other generator in **R**.

```
set.seed(11)
rtn()

## [1] -0.5910311

rtn()

## [1] 0.02659437

set.seed(1)
rtn()

## [1] -0.6264538

rtn()

## [1] 0.1836433

set.seed(11)
rtn()
```

¹See the R package documentation for the citation to the algorithm.

```
## [1] -0.5910311

rtn()

## [1] 0.02659437
```

In practice, this **R**-level function will likely be used in one of two ways:

1. drawing many values from the same truncated Normal distribution
2. drawing many values from different truncated Normal distributions

For the `rtn()` function, these two uses look very similar. The function accepts a `.mean` argument, an `.sd` argument, a `.low` argument, and a `.high` argument. Each should be a vector of length K corresponding to the K distributions of interest. The function does not handle value recycling for the user, so the construction of these vectors must be done *before or during* the call of the `rtn()` function. Incorrectly sized inputs result in an error.

```
## Not Run -- will cause error
rtn(.mean = c(0, 1), .sd = 1)
```

Importantly, this function returns an **NA** value for draws corresponding to invalid input parameters along with a warning. **NA**-inducing input parameters don't interfere with other valid parameters and a vector of the requested length is returned.

For example,

```
rtn(0, -1, 0, 1)

## Warning in checkOutputs(out):  NAs returned.  Check for invalid parameters.
## [1] NA

rtn(0, 1, 0, -1)

## Warning in checkOutputs(out):  NAs returned.  Check for invalid parameters.
## [1] NA

rtn(c(0,0), c(1,1), c(0,0), c(-Inf,Inf))

## Warning in checkOutputs(out):  NAs returned.  Check for invalid parameters.
## [1]      NA 1.178489
```

To suppress input and output checks, use the following:

```
## Not Run -- no warning given
rtn(0, -1, 0, 1, .checks = FALSE)
```

However, this is not recommended unless inputs are being checked before use. Skipping checks in `rtn()` provides a slight performance advantage, but most applications will benefit more from safer code.

2.1.1 Multiple Draws from a Single Distribution

Multiple draws from the same distribution may be requested with a function call like the following:

```
set.seed(1)
output <- rtn(.mean = rep(0, 1000),
             .sd = rep(1, 1000),
             .low = rep(1, 1000),
             .high = rep(2, 1000)
            )
length(output)

## [1] 1000

mean(output)

## [1] 1.388858
```

Here, we are generating 1,000 draws, with each draw, x , coming from $N(0, 1)$ truncated below at 1 and above at 2. The population mean of this distribution is

$$E[x] = \mu + \frac{\phi(\frac{a-\mu}{\sigma}) - \phi(\frac{b-\mu}{\sigma})}{\Phi(\frac{b-\mu}{\sigma}) - \Phi(\frac{a-\mu}{\sigma})} \cdot \sigma,$$

where $\mu = 0$, $\sigma = 1$, ϕ denotes the pdf of the standard Normal distribution, Φ denotes the standard cdf of the standard Normal distribution, and a and b are the lower and upper bounds of truncation, respectively. So, for the above parameter values we have

$$\begin{aligned} E[x] &= \mu + \frac{\phi(\frac{a-\mu}{\sigma}) - \phi(\frac{b-\mu}{\sigma})}{\Phi(\frac{b-\mu}{\sigma}) - \Phi(\frac{a-\mu}{\sigma})} \cdot \sigma \\ &= 0 + \frac{.242 - .054}{.977 - .841} \cdot 1 \\ &\approx 1.383 \end{aligned}$$

Our sample mean for the 1,000 draws (1.389) is close to the population mean (1.383). To get a better sense of how dispersed the sampling distribution for the mean of a sample of 1,000 draws from this distribution is, we can simulate it.

```
bigoutput <- rep(NA, 1000)
for (i in 1:length(bigoutput)) {
  bigoutput[i] <- mean(rtn(.mean = rep(0, 1000),
                             .sd = rep(1, 1000),
                             .low = rep(1, 1000),
                             .high = rep(2, 1000)
                            )
                     )
}
summary(bigoutput)
```

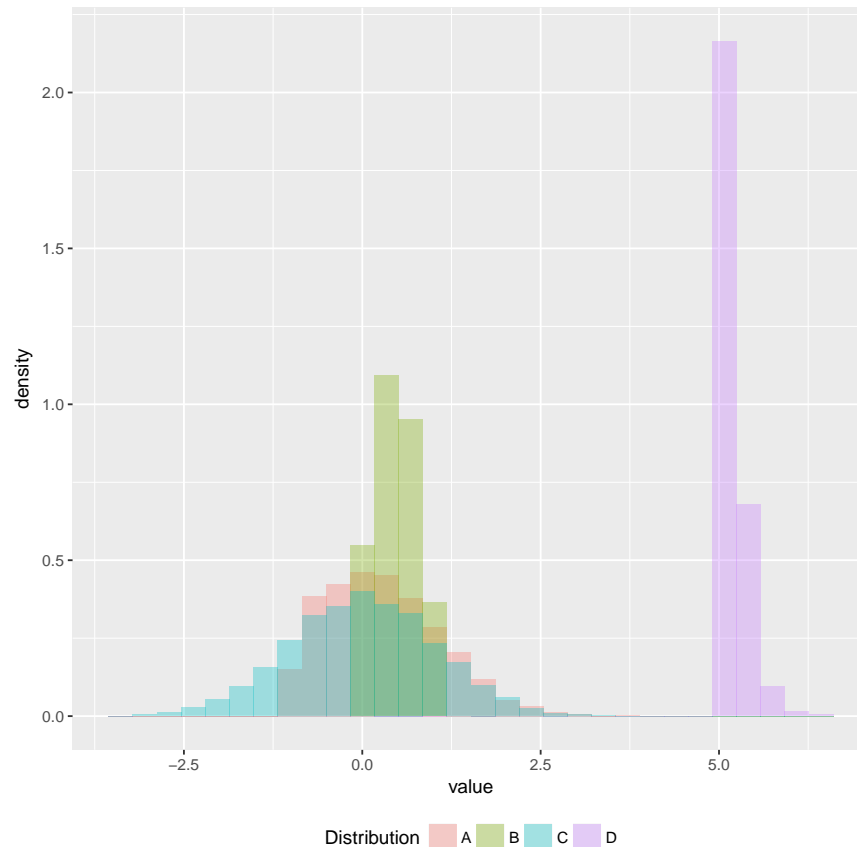
##	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
##	1.357	1.377	1.383	1.383	1.389	1.409

Looking at the summary of the sample means, we see that the sampling distribution of sample means is centered directly on the population mean.

As shown above, valid input for `rtn()` includes `-Inf` and `Inf`. Below are histograms for four different truncated Normal distributions. The `rtn()` function works perfectly well in simulating draws from regions that have a low (read nearly 0) density in a non-truncated Normal distribution. Distribution “D” is an example of this.

```
outputA <- rtn(.mean = rep(0, 5000),
               .sd = rep(1, 5000),
               .low = rep(-1, 5000),
               .high = rep(Inf, 5000)
             )
outputB <- rtn(.mean = rep(0, 5000),
               .sd = rep(1, 5000),
               .low = rep(0, 5000),
               .high = rep(1, 5000)
             )
outputC <- rtn(.mean = rep(0, 5000),
               .sd = rep(1, 5000),
               .low = rep(-Inf, 5000),
               .high = rep(Inf, 5000)
             )
outputD <- rtn(.mean = rep(0, 5000),
               .sd = rep(1, 5000),
               .low = rep(5, 5000),
               .high = rep(Inf, 5000)
             )

dfOutput <- rbind(data.frame(value = outputA, dist = "A"),
                  data.frame(value = outputB, dist = "B"),
                  data.frame(value = outputC, dist = "C"),
                  data.frame(value = outputD, dist = "D")
                )
```



2.1.2 Multiple Draws from Different Distributions

Taking multiple draws from different distributions proceeds in a similar way, though the construction of the arguments passed to `rtn()` changes a bit. If we were interested in characterizing a distribution of draws from a truncated Normal distribution where one (or more) of the parameters was, itself, stochastic, `rtn()` could easily be put to use. Here, the vector of lower bounds and upper bounds are each the result of an `rtn()` function call (notice that $a < b$ by construction).

Then, we can sample 1,000 draws from this truncated Normal distribution of interest where the mean and standard deviation are fixed, but the bounds of truncation, themselves, are taken from a distribution (in this case, a truncated Normal distribution).

```

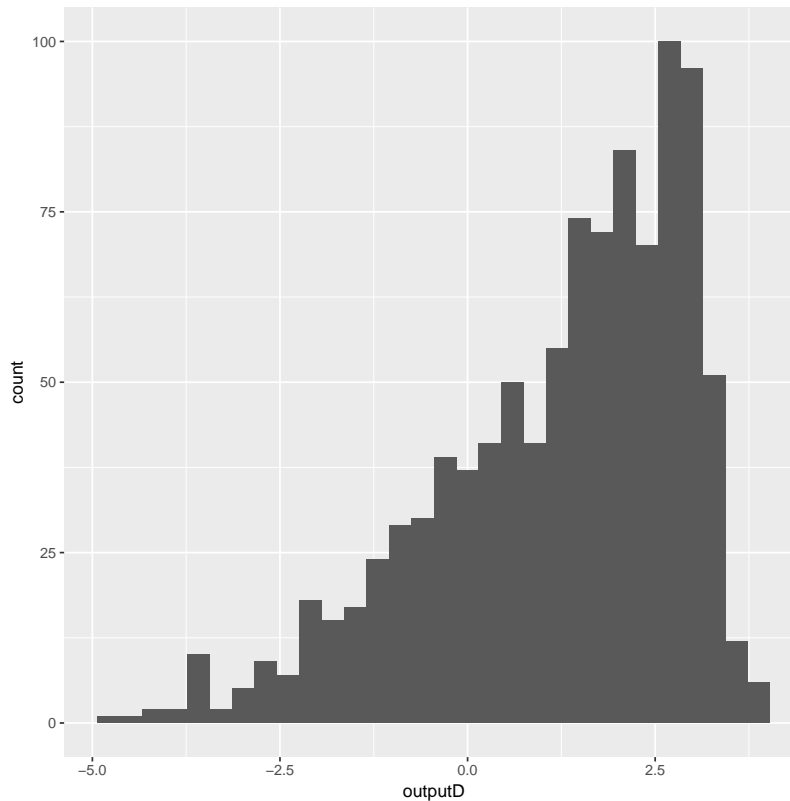
lows <- rtn(rep(0, 1000),
            rep(3, 1000),
            rep(-10, 1000),
            rep(3, 1000)
            )
highs <- rtn(rep(0, 1000),
            rep(3, 1000),
            rep(3, 1000),
            rep(4, 1000)
            )
all(lows < highs)

```

```
## [1] TRUE

outputD <- rtn(.mean = rep(0, 1000),
               .sd = rep(3, 1000),
               .low = lows,
               .high = highs
             )

ggplot() +
  geom_histogram(aes(x = outputD))
```



This sampling distribution is non-standard and the easiest way to characterize it would be through a simulation like the above.

2.2 Other Functions

In addition to random number generation, functions are provided for calculating other quantities of interest.

To calculate the expectation of a given truncated Normal distribution, use `etn()`:

```
etn(.mean = 0,
    .sd = 1,
    .low = 0,
    .high = 10
  )
```

```
## [1] 0.7978846

etn(0, 1, 3.5, 3.7)

## [1] 3.588118
```

The variance can be found in a similar way using `vtn()`:

```
vtn(.mean = 0,
    .sd = 1,
    .low = 0,
    .high = 10
)

## [1] 0.3633802

vtn(0, 1, 3.5, 3.7)

## [1] 0.003244555
```

The density at a specific value for a given Truncated normal distribution is found with `dttn()`:

```
dttn(.x = 4,
     .mean = 0,
     .sd = 1,
     .low = 0,
     .high = 10
)

## [1] 0.0002676605

dttn(3.6, 0, 1, 3.5, 3.7)

## [1] 4.901908
```

Finally, the entropy of a given truncation Normal distribution is found with `enttn()`:

```
enttn(.mean = rep(0, 2),
      .sd = c(.01, 100),
      .low = rep(-1, 2),
      .high = rep(1, 2)
)

## [1] -3.1862317 0.6931472
```

3 C++-level Usage

This section documents how to use the C++-level functionality in subsequent C++ development. Specifically, using the **RcppTN** C++ API via `sourceCpp()` and an **Rcpp**-based R package are shown. Presently, the following functions are exposed at the C++ level.

rtn1

```
double rtn1(double mean, double sd, double low, double high) ;
```

etn1

```
double etn1(double mean, double sd, double low, double high) ;
```

vtn1

```
double vtn1(double mean, double sd, double low, double high) ;
```

dtn1

```
double dtn1(double x, double mean, double sd, double low, double high) ;
```

enttn1

```
double enttn1(double mean, double sd, double low, double high) ;
```

Caveats. The R-level function ultimately calls these C++-level functions. So, all of the features of the R-level function apply here (e.g., respecting R' RNG state). However, as is true in **Rcpp**, this is left to the user to enforce. No checking or error handling is provided with these functions. These functions live in the **RcppTN** namespace.

3.1 Examples

Via `sourceCpp()`. In non-package R code, use is very straightforward due to the mechanisms provided by **Rcpp**. Include the appropriate header file as you would for **Rcpp**. In addition, use the depends pseudo-attribute with “// [[**Rcpp**::depends(**RcppTN**)]]” to ensure that linker finds the symbols. From there, use is as you would expect.

```
library("Rcpp")
sourceCpp(code = "
#include <Rcpp.h>

#include <RcppTN.h>
// [[Rcpp::depends(RcppTN)]]

using namespace Rcpp ;

// [[Rcpp::export]]
List rcpp_hello_world() {
  double a = RcppTN::rtn1(0.0, 1.0, 3.5, 3.7) ;
  double b = RcppTN::etn1(0.0, 1.0, 3.5, 3.7) ;
  double c = RcppTN::vtn1(0.0, 1.0, 3.5, 3.7) ;
  double d = RcppTN::dtn1(3.6, 0.0, 1.0, 3.5, 3.7) ;
  double e = RcppTN::enttn1(0.0, 1.0, 3.5, 3.7) ;
```

```

    NumericVector y = NumericVector::create(a, b, c, d, e) ;
    List z = List::create(y) ;
    return(z) ;
}
"
    )

rcpp_hello_world()

```

Via an Rcpp-based package. In R, use `Rcpp.package.skeleton()` from **Rcpp** to create an empty, but functional, R package.

```

library("Rcpp")
Rcpp.package.skeleton(path=~ /Desktop")

```

Navigate inside the newly created `anRpackage` directory and edit the `DESCRIPTION` file. Add `RcppTN` to the `Depends:` and `LinkingTo:` lines of the file as in

```

Depends: RcppTN
LinkingTo: Rcpp, RcppTN

```

Now, edit the C++ function `rcpp_hello_world()` in `anRpackage/src/rcpp_hello_world.cpp` to read

```

#include <Rcpp.h>
#include <RcppTN.h>

using namespace Rcpp;

// [[Rcpp::export]]
List rcpp_hello_world() {
    double a = RcppTN::rtn1(0.0, 1.0, 3.5, 3.7) ;
    double b = RcppTN::etn1(0.0, 1.0, 3.5, 3.7) ;
    double c = RcppTN::vtn1(0.0, 1.0, 3.5, 3.7) ;
    double d = RcppTN::dtn1(3.6, 0.0, 1.0, 3.5, 3.7) ;
    double e = RcppTN::enttn1(0.0, 1.0, 3.5, 3.7) ;
    NumericVector y = NumericVector::create(a, b, c, d, e) ;
    List z = List::create( y ) ;
    return(z) ;
}

```

To see the effect of this, install the “`anRpackage`” package and load it in R. From there, make subsequent calls to the `rcpp_hello_world()` function. With a similar approach, the `rtn1()` function can be called in a more useful way within other C++-level codes without the need for re-coding the wheel. The only difference between this approach and the `sourceCpp()` approach is that the `depends` pseudo-attribute is no longer needed and is replaced by the modification to the `LinkingTo:` field of the `DESCRIPTION` file.