Benchmarking `rtn()`'s Performance

Jonathan Olmsted
`jpolmstedgmail.com`

March 28, 2014

This brief document shows some performance benchmarks of **RcppTN**'s `rtn()` compared to other truncated Normal distribution RNG's in R. The other functions considered come from the R packages **truncnorm** and **msm**.

1 Three different RNG's

Broadly speaking, calls to the three different R functions are similar. In the simplest case (no truncation), they even identical return values.

```
library(RcppTN)
library(truncnorm)
library(msm)
library(microbenchmark)

set.seed(1)
rtn()  # RcppTN

## [1] -0.6265

set.seed(1)
rtnorm(n = 1)  # msm

## [1] -0.6265

set.seed(1)
rtruncnorm(n = 1)  # truncnorm

## [1] -0.6265
```

But, this is not true in general. Differences in return values result from the use of different algorithms and different implementations of the same algorithm. First, consider a standard Normal distribution truncated below 4 and above 4.1. Here, the output from the **RcppTN** package and the **truncnorm** package agree.

```
set.seed(1)
rtn(.mean = 0, .sd = 1, .low = 4, .high = 4.1)
```

```
## [1] 4.027

set.seed(1)
rtnorm(n = 1, mean = 0, sd = 1, lower = 4, upper = 4.1)

## [1] 4.034

set.seed(1)
rtruncnorm(n = 1, mean = 0, sd = 1, a = 4, b = 4.1)

## [1] 4.027
```

Yet, in the case of truncation below 5 without any truncation from above, the output from the **RcppTN** and the **msm** package agree. Again, this is just a result of how each sampler is implemented using R's base RNG functionality. None of these return values is incorrect, per se, but it is worth noting that the functions do not produce identical output, even if they are all valid RNG's for the same distribution.

```
set.seed(1)
rtn(.mean = 0, .sd = 1, .low = 5, .high = Inf)

## [1] 5.145

set.seed(1)
rtnorm(n = 1, mean = 0, sd = 1, lower = 5, upper = Inf)

## [1] 5.145

set.seed(1)
rtruncnorm(n = 1, mean = 0, sd = 1, a = 5, b = Inf)

## [1] 5.151
```

2 Compiled Code is faster than Interpreted Code

In setting up a series of conditions under which to compare performance, we will consider drawing samples of size 10, 1,000, and 100,000. And will consider standard Normal distributions restricted to the intervals $[-1, 1]$, $[5, \infty]$, $[-\infty, 10]$, $[4, 7]$, $[4, 4.1]$, $[-\infty, \infty]$, and $[50, 100]$.

```
sizes <- c(1e1, 1e3, 1e5)
lows <- c(-1, 5, -Inf, 4, 4, -Inf, 50)
highs <- c(1, Inf, 10, 7, 4.1, Inf, 100)
```

Both **RcppTN** and **truncnorm** use compiled code for their RNG. However, the RNG in **msm** is written in R. As a result, the performance cost that one would expect manifests in even the a simple case where the standard Normal distribution is truncated below at -1 and above at 1. This case is sufficiently easy that even a naive Accept-Reject sampler works perfectly fine. Yet, the C(++)-based implementations are over 15 times faster in drawing samples of size 1,000

```
s <- sizes[2]

microbenchmark(
    "rtn" = rtn(.mean = rep(0, s),
    .low = rep(lows[1], s),
    .high = rep(highs[1], s),
    .checks = FALSE
    ),
    "rtruncnorm" = rtruncnorm(n = s,
    a = rep(lows[1], s),
    b = rep(highs[1], s)
    ),
    "rtnorm" = rtnorm(n = s,
    lower = rep(lows[1], s),
    upper = rep(highs[1], s)
    ),
    times = 100
    )

## Unit: microseconds
##        expr    min     lq median     uq    max neval
##         rtn  130.1  143.1  147.8  161.0  227.3   100
##  rtruncnorm  172.7  183.2  193.5  203.2  293.4   100
##      rtnorm 2324.0 2438.3 2550.2 3130.8 5861.0   100
```

A similarly large performance cost due to writing the RNG in R is seen in a harder case: a standard Normal distribution truncated below at 4 and above at 4.1. The sample size is still 1,000.

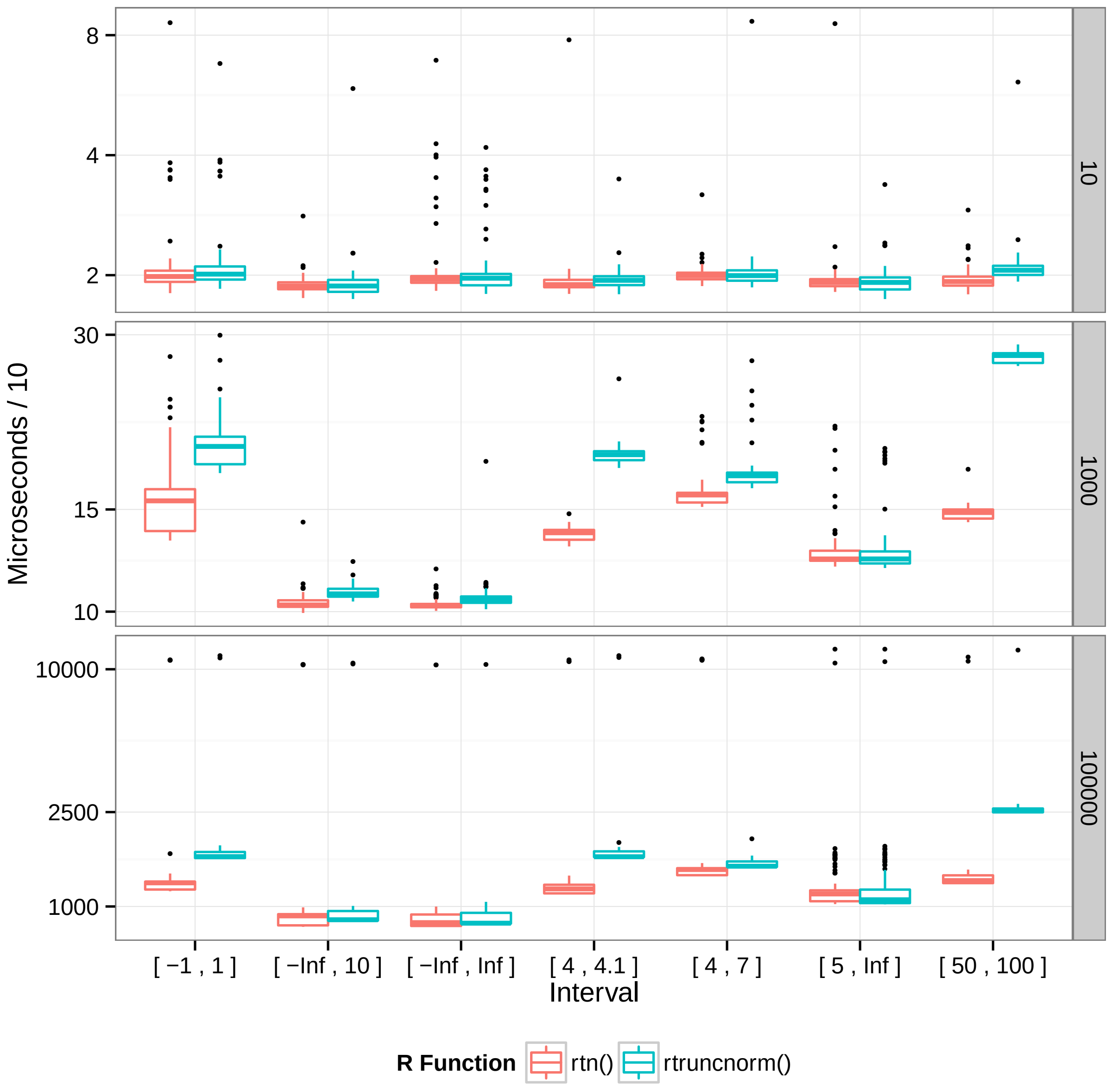
```
microbenchmark(
    "rtn" = rtn(.mean = rep(0, s),
    .low = rep(lows[5], s),
    .high = rep(highs[5], s),
    .checks = FALSE
    ),
    "rtruncnorm" = rtruncnorm(n = s,
    a = rep(lows[5], s),
    b = rep(highs[5], s)
    ),
    "rtnorm" = rtnorm(n = s,
    lower = rep(lows[5], s),
    upper = rep(highs[5], s)
    ),
    times = 100
    )

## Unit: microseconds
##        expr    min     lq median     uq    max neval
##         rtn  130.7  137.4  140.3  147.5 1476.7   100
##  rtruncnorm  173.7  183.0  186.2  191.0  255.9   100
##      rtnorm 2608.1 2828.6 2947.2 3529.4 4497.9   100
```

For this reason, the `rtnorm()` function from the **msm** package is excluded from subsequent analysis. Not only is it assumed that it will be the slowest for the different sample sizes and truncation bounds considered, but it will just take too long to build the vignette if it is included.

3 RcppTN vs. truncnorm

The RNG's in **RcppTN** and **truncnorm** are written in **Rcpp**-based C++ and C, respectively. However, they implement different mathematical algorithms. The former uses Robert (1995) and the latter uses Geweke (1991). To compare the R-level performance of the two, the full set of conditions described above is considered.



*Results of benchmarking comparing RNG performance for Truncated Normal distributions from **RcppTN** and **truncnorm**.*

The motivation for **RcppTN** isn't speed, but the **Rcpp**-based implementation performs quite-well. For larger sample sizes (e.g., $\geq 10^6$), `rtn()` does not necessarily keep its efficiency edge.