Benchmarking `rtn()`'s Performance

Jonathan Olmsted
`jpolmstedgmail.com`

September 3, 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] -0.6264538

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

## [1] -0.6264538

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

## [1] -0.6264538
```

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] 4.026551

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

## [1] 4.034397

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

## [1] 4.026551
```

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] 5.145435

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

## [1] 5.145435

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

## [1] 5.151036
```

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      mean   median        uq      max neval
##         rtn 139.844 147.0310  171.9281 150.9480  164.3575 1411.007   100
##  rtruncnorm 155.970 158.8685  188.3300 161.7520  166.9810 2300.541   100
##      rtnorm 691.414 722.6430 1002.7170 737.6945 1204.0055 3201.541   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.

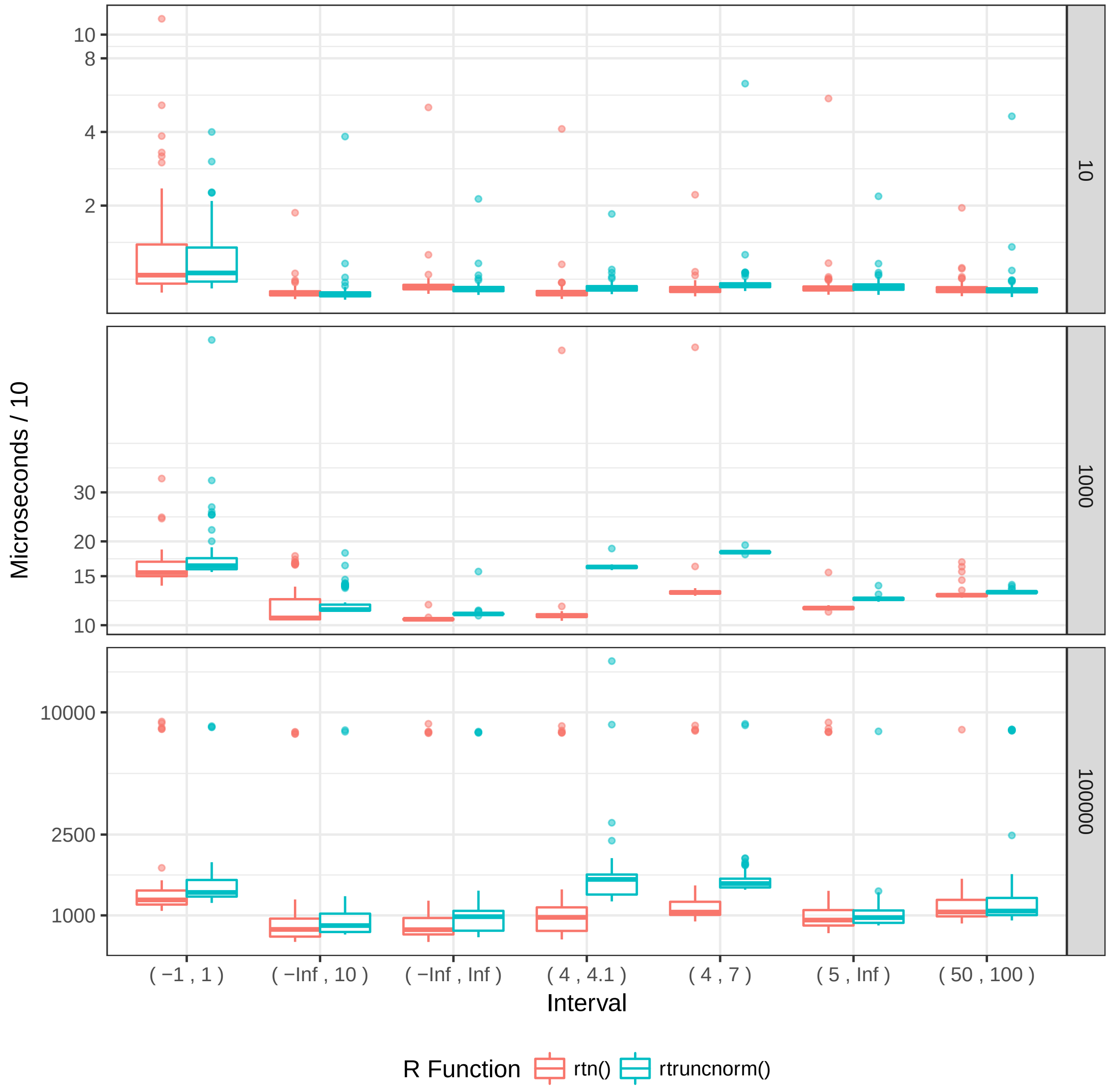
```
microbenchmark(
    "rtn" = rtn(.mean = rep(0, s),
    .low = rep(lows[5], s),
    .high = rep(highs[5], s),
    .checks = FALSE
    ),
    "rtruncnorm" = rtruncnorm(n = s,
    a = rep(lows[5], s),
    b = rep(highs[5], s)
    ),
    "rtnorm" = rtnorm(n = s,
    lower = rep(lows[5], s),
    upper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

## Unit: microseconds
##        expr      min       lq      mean   median       uq      max neval
##         rtn  105.234  112.128  122.4467  117.996  129.235  191.536   100
##  rtruncnorm  160.533  167.328  179.1963  174.119  181.001  278.646   100
##      rtnorm 1122.528 1213.619 1630.0178 1300.100 1729.848 5186.062   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.