R package qdist: A functional programming approach to truncated probability ditributions

Introduction

Truncated probability distribution are widely used in many applied areas of statistics. Loss severity models in the finance or insurance industries are good cases in point. Package `stats` provides a wide set of functions for distribution models. Other packages, such as package `evd`, provide extra distributional models. Truncated distribution models can be found in package `truncdist`(Novomestky and Nadarajah, 2012; Nadarajah and Kotz, 2006). Package `truncdist` provides an efficient method for computing from truncated distribution but requires a change in our programming style when compared with the approach we use for non-truncated models. This article aims to provide an alternative method for computing from truncated probability distribution. This method is made of two steps:

- define the required functions suitable for computing from specific truncated probability models by using the tools available within this package;
- use the newly created functions for computing from truncated probability models.

Truncated probability function in R

R provides probability, density, quantile and random number generator functions using a stable naming convention both for the names of the functions and in the first argument of these functions:

- probability distribution functions `p<dist>(q)` with `q` vector of quantiles;
- density functions `d<dist>(x)` with `x` vector of quantiles;
- quantile functions `q<dist>(p)` with `p` vector of probabilities;
- random number generation `r<dist>(n)` with `n` number of observations.

where `<dist>` indicates the distribution family.

As a result we have: `pnorm(q)`, `dnorm(x)`, `qnorm(p)` and `rnorm(n)` for normal distribution and similarly for all other distributions.

Therefore, we use to write:

```r
pnorm(q = 8:12, mean = 10, sd = 1)
```

```
## [1] 0.02275013 0.15865525 0.50000000 0.84134475 0.97724987
```

to get probability values at `8:12` from a normal distribution with parameters `mean=10` and `sd=1`.

In case we need values from a truncated distribution, as far as we know, we need to load an extra package such as `truncdist`.

```r
require("truncdist")
```

The package itself works perfectly. In fact, assuming that function `pnorm()` exists, we can get probability values from a normal distribution left truncated at `a` and right truncated at `b`, with parameters `mean = 10` and `sd = 1` by simply writing:

```
ptrunc(q = 8:12, spec = "norm", mean = 10, sd = 1, a = 9, b = 11)
```

```
## [1] 0.0 0.0 0.5 1.0 1.0
```

where `a = 9` and `b = 11` respectively represent the left and the right thresholds for truncation.

Nevertheless, the above command requires a change in our programming style. We are used to have a single R function for each distribution: `pnorm()`, `pweibull()` etc. . .

We could easily write function `tpnorm()`, a truncated version for `pnorm()`, as:

```
tpnorm <- function (q, mean = 0, sd = 1, L = -Inf, U = Inf, ...){
  q <- pmax(pmin(q,U),L)
  pq <- stats::pnorm(q = q, mean = mean, sd = sd, ...)
  pL <- stats::pnorm(q = L, mean = mean, sd = sd, ...)
  pU <- stats::pnorm(q = U, mean = mean, sd = sd, ...)
  p <- (pq-pL)/(pU-pL)
  p
}
```

This function clearly works:

```
tpnorm(q = 8:12, mean = 10, sd = 1, L = 9, U = 11)
```

```
## [1] 0.0 0.0 0.5 1.0 1.0
```

but it is limited to the normal distribution. Following this approach, we would need to write a different function for each probability distribution and, we'll have to admit that all of this could become quite time consuming and not really efficient.

As an alternative approach, we decided to develop package `qdist` available at https://github.com/andreaspano/qdist .

This package is made of four functions:

- `ptruncate()`: truncated probability distributions;
- `dtruncate()`: truncated density functions;
- `qtruncate()`: truncated quantile functions;
- `rtruncate()`: truncated random numbers generator functions.

These functions share the same logic: they take as input a probability distribution as a character string and return the equivalent truncated distribution as a function object so that we can write:

```
devtools::install_github("andreaspano/qdist", build=FALSE)
require(qdist)
tpnorm <- ptruncate("norm")
tdnorm <- dtruncate("norm")
tqnorm <- qtruncate("norm")
trnorm <- rtruncate("norm")
```

As a first detailed example, let's consider `ptruncate()` in more details applied to the normal case.

The newly generated function `tpnorm()` has the same formals as the original `pnorm()`, plus two extra parameters: `L` and `U`, respectively for lower and upper truncation thresholds set by default to `-Inf` and `+Inf`:

```
args(pnorm)
```

```
## function (q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
## NULL
```

```
args(tpnorm)
```

```
## function (q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE,
##     L = -Inf, U = Inf)
## NULL
```

Once we have defined `tpnorm()` we can use it for generating probability values from a non-truncated normal distribution by leaving parameters `L` and `U` set to their defaults:

```
q <- seq(6, 14, len = 100)
p_L_U <- tpnorm(q, mean = 10, sd = 1)
```

or from any truncated normal distribution by setting values for `L` and `U`:

```
p_L9_U11 <- tpnorm(q, mean = 10, sd = 1, L = 9, U = 11)
```

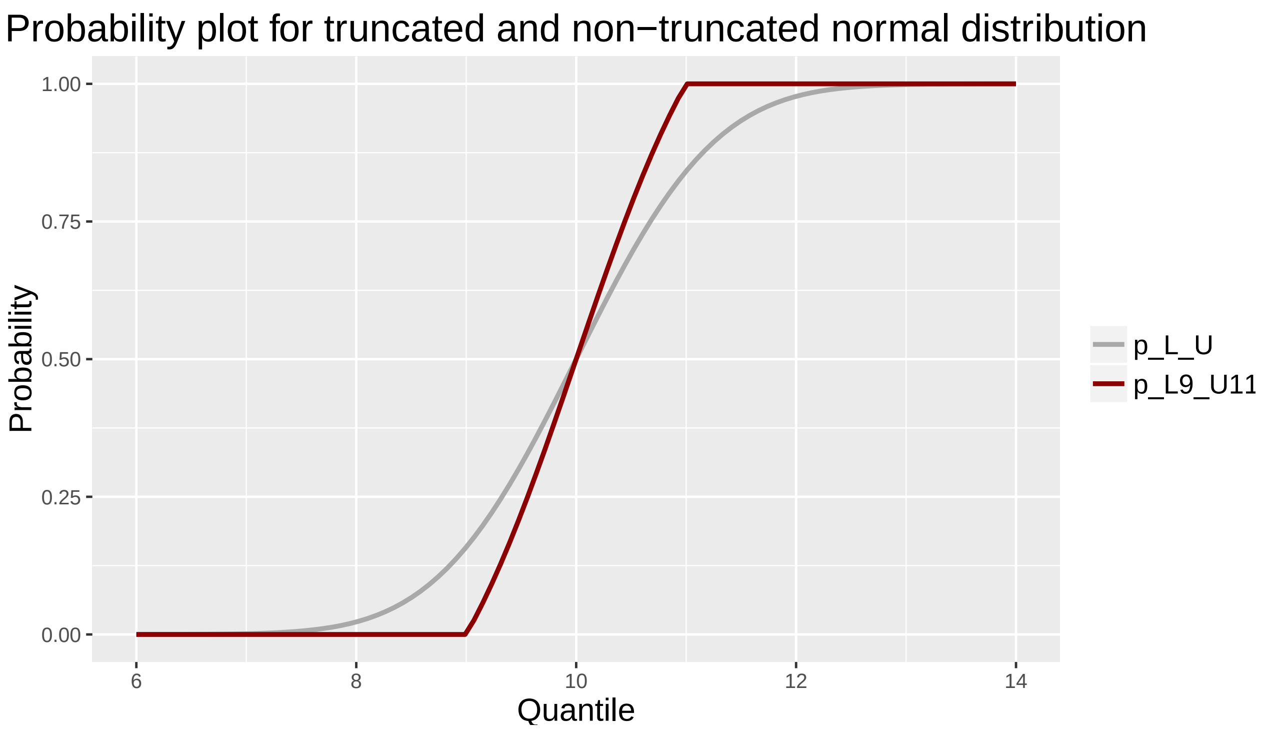
We can visualize these results by plotting them:

```
require(ggplot2)

df <- data.frame(
  q=q,
  tpnorm_type = factor(c(rep("p_L_U", times=length(q)),
    rep("p_L9_U11", times=length(q))), levels = c("p_L_U", "p_L9_U11")),
  value = c(p_L_U, p_L9_U11)
)

pl <- ggplot(data = df, mapping = aes(x=q, y=value, col=tpnorm_type)) +
  geom_line(size=1) +
  scale_color_manual(values = c("p_L_U" = "darkgray", "p_L9_U11" ="darkred")) +
  xlab("Quantile") + ylab("Probability") +
  ggtitle("Probability plot for truncated and non-truncated normal distribution") +
  theme(legend.title=element_blank(), legend.text= element_text(size=12),
    plot.title = element_text(size=17), axis.title=element_text(size=14))

print(pl)
```



As a second example, we consider a density function for a left truncated Weibull distribution. We first generate function `tdweibull()` by using `dtruncate()`:

```
tdweibull <- dtruncate("weibull")
```

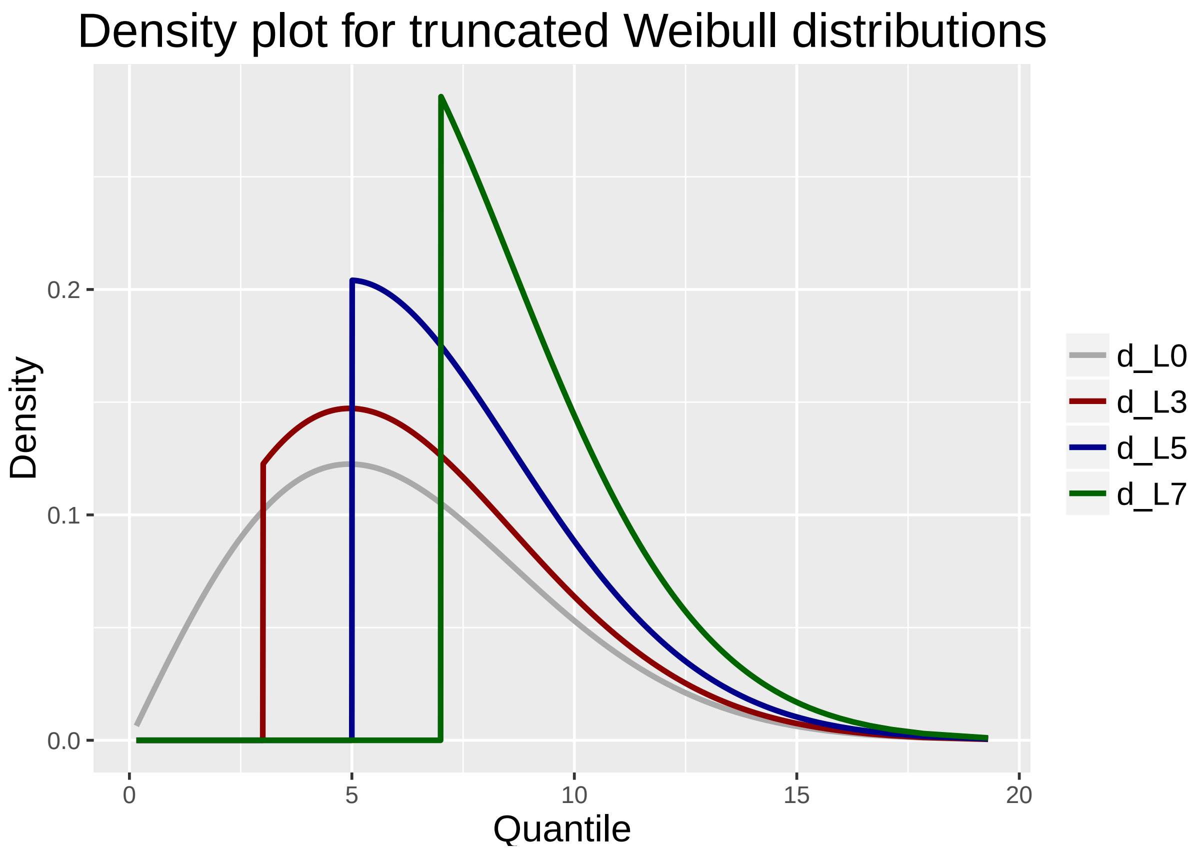
and afterward, we can use it as:

```
x <- qweibull(ppoints(1000), shape = 2, scale = 7)
d_L0 <- tdweibull(x, shape = 2, scale = 7)
d_L3 <- tdweibull(x, shape = 2, scale = 7, L= 3)
d_L5 <- tdweibull(x, shape = 2, scale = 7, L= 5)
d_L7 <- tdweibull(x, shape = 2, scale = 7, L= 7)
```

with the following results:

```
df <- data.frame(
  x=x,
  tdweibull_type = factor(c(rep("d_L0", times=length(x)), rep("d_L3", times=length(x)),
    rep("d_L5", times=length(x)), rep("d_L7", times=length(x))),
    levels = c("d_L0", "d_L3", "d_L5", "d_L7")),
  value = c(d_L0, d_L3, d_L5, d_L7)
)

pl <- ggplot(data = df, mapping = aes(x=x, y=value, col=tdweibull_type)) +
  geom_line(size=1) +
  scale_color_manual(values = c("d_L0" = "darkgray", "d_L3"="darkred",
    "d_L5" = "darkblue", "d_L7"="darkgreen")) +
  xlab("Quantile") + ylab("Density") +
  ggtitle("Density plot for truncated Weibull distributions") +
  theme(legend.title=element_blank(), legend.text= element_text(size=12),
    plot.title = element_text(size=18), axis.title=element_text(size=14))
print(pl)
```



As an example of random number generator function we consider the Gumbel distribution from package `evd`:

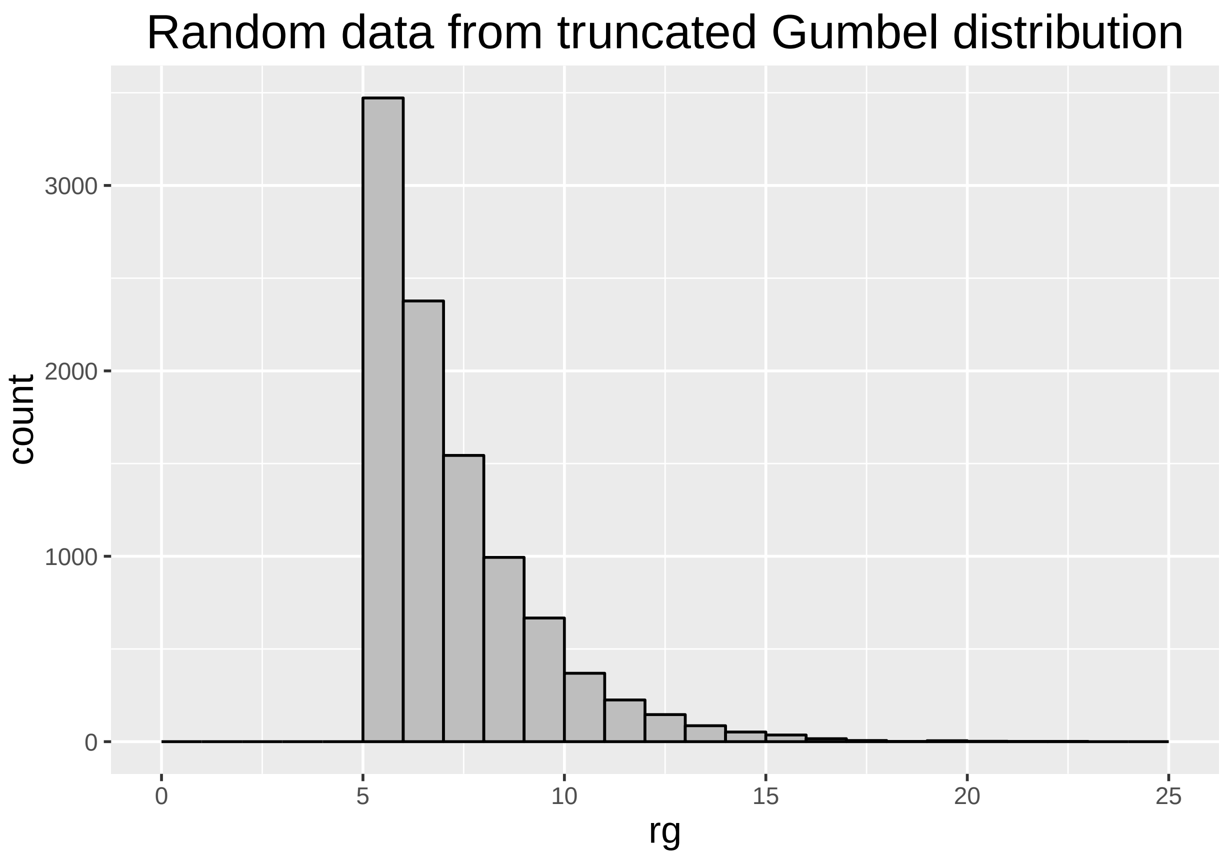
```
require(evd)
trgumbel <- rtruncate("gumbel")
args(trgumbel)
```

```
## function (n, loc = 0, scale = 1, L = -Inf, U = Inf)
## NULL
```

```
rg <- trgumbel(10^4, loc = 3, scale = 2, L = 5)
df <- data.frame(rg=rg)

pl <- ggplot(data = df, mapping = aes(x=rg)) +
  geom_histogram(fill="grey", color="black", breaks=0:25) + xlim(c(0,25)) +
  ggtitle("Random data from truncated Gumbel distribution") +
  theme(plot.title = element_text(size=18), axis.title=element_text(size=14))

print(pl)
```



This approach works for both continuous and discrete probability functions. Let's consider the case of main quantiles generated from a Poisson distribution. We first generate the truncated quantile function:

```
tqpois <- qtruncate("pois")
```

We use the newly created function for computing percentiles from four Poisson distributions truncated at different tresholds:

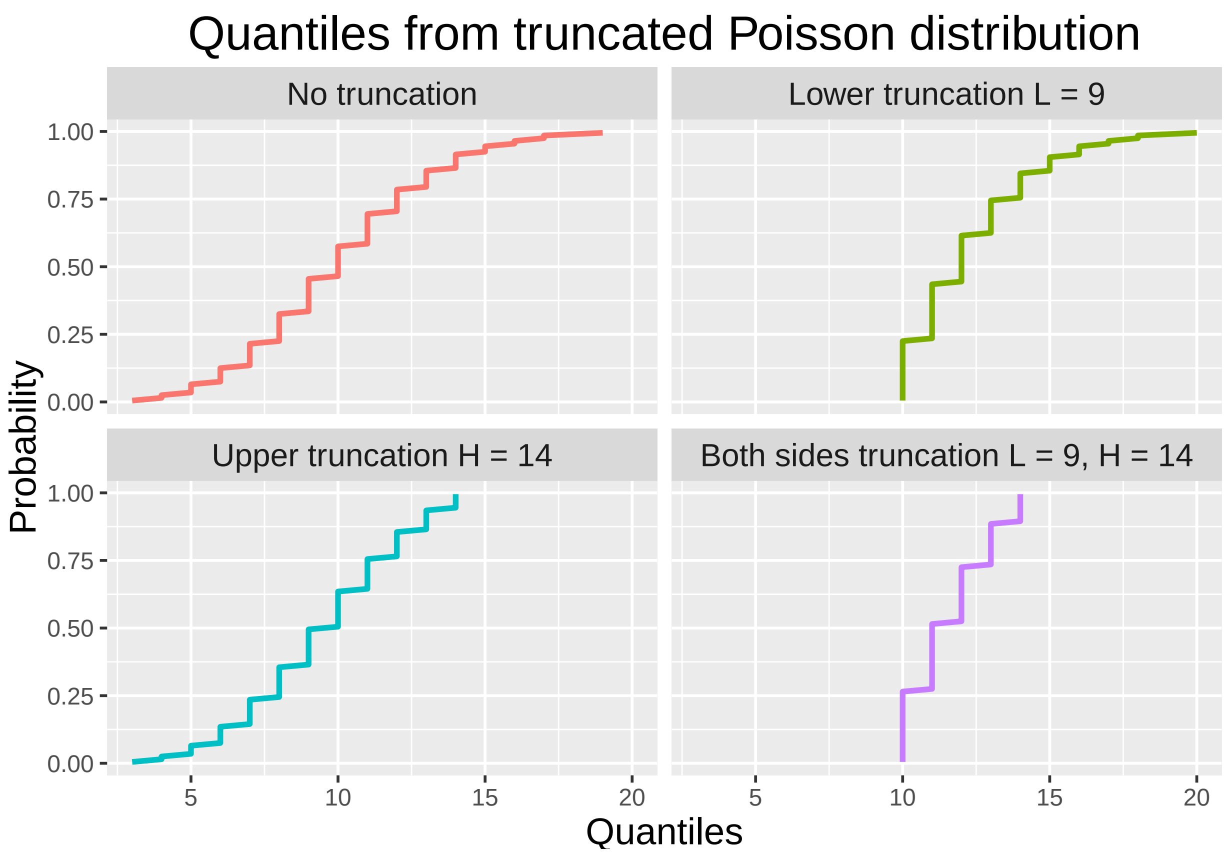
```
p <- ppoints(100)
#No truncation
q <- tqpois(p, lambda = 10)
#Lower truncation L = 9
q_L9 <- tqpois(p, lambda = 10, L = 9)
#Upper truncation H = 14
q_U14 <- tqpois(p, lambda = 10, U = 14)
#Both sides truncation L = 9, H = 14
q_L9_U14 <- tqpois(p, lambda = 10, L = 9, U = 14)
```

Finally, we plot the computed percentiles:

```
df <- data.frame(p=p,
  tqpois_type=factor(c(rep("No truncation", times=length(p)),
    rep("Lower truncation L = 9", times=length(p)),
    rep("Upper truncation H = 14", times=length(p)),
    rep("Both sides truncation L = 9, H = 14", times=length(p))),
    levels= c("No truncation", "Lower truncation L = 9",
      "Upper truncation H = 14", "Both sides truncation L = 9, H = 14")),
    value=c(q, q_L9, q_U14, q_L9_U14))

pl <- ggplot(data = df, mapping = aes(x=value, y=p, col=tqpois_type)) +
  geom_line(size=1, show.legend = FALSE) +
  ylab("Probability") + xlab("Quantiles") +
  ggtitle("Quantiles from truncated Poisson distribution") +
  facet_wrap(~tqpois_type, nrow = 2) +
  theme(strip.text = element_text(size=12),
    plot.title = element_text(size=18), axis.title=element_text(size=14))

print(pl)
```



The gamma case

The gamma case may result a bit faulty because of the check on input parameters implemented within the body of the gamma set of functions. As a consequence, suppose we define a truncated quantile the gamma distribution as:

```
tqgamma <- qtruncate("gamma")
```

When we use `tqgamma()` with parameter rate it works but may return a set of `warnings`:

```r
tqgamma(.25, shape = 1, rate = .3)
```

```
## Warning in pdist(q = Inf, scale = 1/rate, lower.tail = TRUE, log.p =
## FALSE, : specify 'rate' or 'scale' but not both

## Warning in pdist(q = -Inf, scale = 1/rate, lower.tail = TRUE, log.p =
## FALSE, : specify 'rate' or 'scale' but not both

## Warning in pdist(q = -Inf, scale = 1/rate, lower.tail = TRUE, log.p =
## FALSE, : specify 'rate' or 'scale' but not both

## Warning in pdist(q = Inf, scale = 1/rate, lower.tail = TRUE, log.p =
## FALSE, : specify 'rate' or 'scale' but not both

## Warning in pdist(q = -Inf, scale = 1/rate, lower.tail = TRUE, log.p =
## FALSE, : specify 'rate' or 'scale' but not both

## Warning in qdist(p = 0.25, shape = 1, rate = 0.3, scale = 1/rate,
## lower.tail = TRUE, : specify 'rate' or 'scale' but not both

## [1] 0.9589402
```

when we use the same function with parameter `scale`, it returns an error:

```r
tqgamma(ppoints(10), shape = 1, scale = 3)
```

```
## Error in pdist(q = Inf, rate = 1, lower.tail = TRUE, log.p = FALSE, shape = 1, : specify 'rate' or '
```

Note that, if we simply redefine `qgamma()` and `pgamma()` as:

```r
qgamma <- function (p, shape, rate = 1, lower.tail = TRUE, log.p = FALSE) {
  scale <- 1/rate
  .Call(stats:::C_qgamma, p, shape, scale, lower.tail, log.p)
}

pgamma <- function (q, shape, rate = 1, lower.tail = TRUE, log.p = FALSE) {
  scale <- 1/rate
  .Call(stats:::C_pgamma, q, shape, scale, lower.tail, log.p)
}
```

now everything should work fine:

```r
tqgamma <- qtruncate("gamma")
tqgamma(p = .25, shape = 1, rate = 3)
```

```
## [1] 0.09589402
```

Extending the computation

Once a truncated distribution is defined, we can use it as a building block for further implementations. Suppose we want to define a function for maximum likelihood estimate for the truncated normal distribution, we can achieve this by first defining the truncated density function for the normal distribution:

```
tdnorm <- dtruncate("norm")
```

and, subsequently, by using `tdnorm()` within an estimator function:

```
ltnorm <- function(x, L = -Inf, U = Inf) {
  theta <- c(mean(x), sd(x))
  ll <- function(theta, x, L = -Inf, U = Inf) {
    mean <- theta[1]
    sd <- theta[2]
    ld <- tdnorm(x = x, mean = mean, sd = sd, L = L , U = U, log=TRUE)
    -sum(ld)
  }
  optim(par = theta, fn = ll, x = x, L = L, U = U)[["par"]]
}
```

As a result:

```
trnorm <- rtruncate("norm")
x <- trnorm(n = 1000, mean = 5, sd = 2, L = 3, U = 6)
ltnorm( x = x, L = 3, U = 6)
```

```
## [1] 4.5584566 0.7435244
```

Computational details

Package `qdist` can be thought as a function factory (Wickham) for truncated distribution functions.

Function `ptruncate()`, for example, takes as input the distribution name: say *norm* and proceeds as follows:

- gets the corresponding function `pnorm()`;
- uses `pnorm()` to create a function, say `probability()` that computes truncated probability for normal distributions;
- modify the formals of `probability()` so that it has the same formals as `pnorm()` plus `L` and `U` corresponding to the lower and upper threshold for truncation;
- returns `probability()`.

Package `qdist` is based on two key concepts being part of the functional nature of the R programming language:

- the environment of a function can be used as a placeholder for other objects;
- function formals can be manipulated.

Environment of a function

The *R Language Definition* manual defines environments as: *"consisting of two things. A frame, consisting of a set of symbol-value pairs, and an enclosure, a pointer to an enclosing environment"*

This idea of a pointer to an enclosing environment is at the core of the R mechanism when looking for objects:

"When R looks up the value for a symbol the frame is examined and if a matching symbol is found its value will be returned. If not, the enclosing environment is then accessed and the process repeated".

Environments in R play a crucial role as they just work in the background of any R functionality. Any R session has an associated environment as returned by:

```
environment()
```

```
## <environment: R_GlobalEnv>
```

and, very important for our purposes, any function has an associated environment as stated by the *R Language Definition* manual: *"functions have three basic components: a formal argument list, a body and an environment".*

Specifically, the *environment of a function* is the environment that was active at the time that the function was created. Generally, for user defined function, the *Global environment*:

```
f <- function() 0
environment(f)
```

```
## <environment: R_GlobalEnv>
```

or, when a function is defined within a package, the environment associated to that package:

```
environment(mean)
```

```
## <environment: namespace:base>
```

Along with the environment where the function was created, functions interact with several other environments. The *evaluation environment* of a function is one of them.

The *evaluation environment* of a function is created any time a function is called and is used to host the computation of the function.

The evaluation environment is destroyed when the function exits.

As any environment, the evaluation environment of a function has a parent: the environment of the function.

When we define a function, the function itself knows about its environment and, as a consequence, the function has access to all symbols belonging to that environment.

As an example we may consider a function defined in a dedicated environment along with some other objects defined in the same environment.

```
env <- new.env()
with(env,{y <- 99; g <- function(x){x+y}})
with(env, g(1))
```

```
## [1] 100
```

As we can see, clearly `g()` knows that `x=1` as it was passed to the function as an argument but, `g()` also remembers that `y=99` as y belongs the the environment `env`: the environment of `g()`.

Playing with the environment of a function and the execution environment of a function, we can create a function `g()` that returns a function `f()`:

```
g <- function() {
  f <- function() 0
  f
}
```

create `f()` as a result of a call to `g()`

```
f <- g()
```

and normally run `f()` as:

```
f()
```

```
## [1] 0
```

in this case, the *evaluation environment* of `g()` corresponds to the *environment* of `f()` as `f()` is created within the *evaluation environment* of `g()`.

As a result, when `g()` exits, its *evaluation environment* is not destroyed as it became the *environment* of `f()`.

Following this line, we can define `g(y)` so that the *evaluation environment* of `g(y)` is used to pass any argument `y` to `f(x)`:

```
g <- function(y) {
  f <- function(x) {x+y}
}
```

We can now define and run `f(x)` as:

```
f <- g(1)
f(x = 2)
```

```
## [1] 3
```

This mechanism allows us to define a *functions factory*: a function `g(y)` that, by varying the values assigned to `y`, allow many `fi(x)` to be defined with very little effort:

```
f1 <- g(1)
f2 <- g(2)
f3 <- g(3)
```

and use them straighfortly:

```
f1(x = 100)
```

```
## [1] 101
```

```
f2(x = 100)
```

```
## [1] 102
```

```
f3(x = 100)
```

```
## [1] 103
```

Formals argument list

The argument list of a function, as stated in *R Language Definition* manual is: *a comma-separated list of arguments. An argument can be a symbol, or a symbol = default construct.*

Function `formals()` returns the formal arguments of a function as an object of class `pairlist`.

```
formals_sd <- formals(sd)
formals_sd
```

```
## $x
##
##
## $na.rm
## [1] FALSE
```

```
class(formals_sd)
```

```
## [1] "pairlist"
```

As a replacement method exists for function formals :

```
exists("formals<-")
```

```
## [1] TRUE
```

formals of a function can manipulated by using function `alist()`: a `list()` type function that handles unevaluated arguments

```
f <- function(x, y=0) x+y
f(1)
```

```
## [1] 1
```

```
formals(f) <- alist(x=, y=1)
f(1)
```

```
## [1] 2
```

As an example of practical use of `formals()` we may decide to re-define function `mean()` that defaults `na.rm` to `TRUE` by simply:

```r
formals(mean.default)[["na.rm"]] <- TRUE
mean(c(1,2,NA))
```

```
## [1] 1.5
```

Bibliography

Nadarajah S. and Kotz S. . R programs for truncated distributions. *Journal of Statistical Software, Code Snippets*, 16(2):1–8, 8 2006. ISSN 1548-7660. URL http://www.jstatsoft.org/v16/c02. [p1]

Novomestky F. and Nadarajah S. . *truncdist: Truncated Random Variables*, 2012. URL http://CRAN.R-project.org/package=truncdist . R package version 1.0-1. [p1]

Wickham H. . Advanced r. http://adv-r.had.co.nz/. Accessed: 2014-07-31. [p7]