# S4 Classes for Distributions—a manual for packages "distr", "distrEx", "distrEllipse", "distrMod", "distrSim", "distrTEst", "distrTeach", version 2.9

Peter Ruckdeschel[*]
Matthias Kohl[†]
Thomas Stabla[‡]
Florian Camphausen[§]

Institute for Mathematics
School of Mathematics and Science
Oldenburg University
PO box 25 03
26111 Oldenburg (Oldb.)
Germany
e-Mail: peter.ruckdeschel@uni-oldenburg.de

January 12, 2025

## Abstract

"distr" is a package for R from version 1.8.1 onwards that is distributed under LGPL-3. Its own current version is 2.9. The aim of this package is to provide a conceptual treatment of random variables (r.v.'s) by means of S4–classes. A mother

---

[*]Carl von Ossietzky Universität Oldenburg
[†]Hochschule Furtwangen
[‡]Graf-Münster-Gymnasium, Bayreuth
[§]European Central Bank, Frankfurt

1

class Distribution is introduced with slots for a parameter and for functions r, d, p, and q for simulation, respectively for evaluation of density / c.d.f. and quantile function of the corresponding distribution. All distributions of the "stats" package are implemented as subclasses of either AbscontDistribution or DiscreteDistribution, which themselves are again subclasses of UnivariateDistribution. By means of these classes, we may automatically generate new objects of these classes for the laws of r.v.'s under standard mathematical univariate transformations and under standard bivariate arithmetical operations acting on independent r.v.'s. Package "distr" in this setting works as basic package for further extensions. These start with package "distrEx", covering statistical functionals like expectation, variance and the median evaluated at distributions, as well as distances between distributions and basic support for multivariate and conditional distributions. Next, from version 2.0 on, comes package "distrMod" which uses these concepts to provide an object orientated competitor to fitdistr from package "MASS" in covering estimation in statistical models. Further on there are packages "distrSim" for the standardized treatment of simulations, also under contaminations and package "distrTEst" with classes and methods for evaluations of statistical procedures on such simulations. Finally, from version 2.0 on, there is package "distrTeach" to embody illustrations for basic stats courses using our distribution classes.

From version 2.4 on, we have moved support for extreme value distributions, as well as for certain scale-shape distributions to the new package "RobExtremes". This concerns the Gumbel, Weibull, Pareto distributions.

# Contents

Parts of this document appeared in an earlier and much shorter form in *R-News*, **6**(2) as "S4 Classes for Distributions", c.f. [8], which in its published form refers to package versions 1.6, resp. 0.4-2. This present document takes into account the subsequent revisions and versions.

# 0   Motivation

R up to now contains powerful techniques for virtually any useful distribution using the
suggestive naming convention `[prefix]<name>` as functions where `[prefix]` stands for `r`,
`d`, `p`, or `q` and `<name>` is the name of the distribution.

There are limitations of this concept, however: You can only use distributions which are
implemented in some library already or for which you yourself have provided an implemen-
tation. In many natural settings you want to formulate algorithms once for all distributions,
so you should be able to treat the actual distribution `<name>` as sort of a variable.

You may of course paste together prefix and the value of `<name>` as a string and then use
**eval**(**parse**(....)). This is neither very elegant nor flexible, however.

Instead, we would rather like to implement the algorithm by passing an object of some
distribution class as argument to the function. Even better though, we would use a generic
function and let the `S4`-dispatching mechanism decide what to do at run-time. In particular,
we would like to automatically generate the corresponding functions `r`, `d`, `p`, and `q` for the
law of expressions like $X+3Y$ for objects $X$ and $Y$ of class Distribution, or, more general, of a
transformation of $X, Y$ under a function $f: \mathbb{R}^2 \to \mathbb{R}$ which is already realized as a function
in R.

This is possible with package `"distr"`. As an example, try

```
## Loading required package:  distrTEst
## Loading required package:  setRNG
## Loading required package:  distrSim
## Loading required package:  distr
## Loading required package:  startupmsg
## *** Utilities for Start-Up Messages ***
##
## Version information in start-up messages is currently suppressed.  To see
such information on startup as in versions of this pkg prior to this versionr,
set option "StartupBanner" to a value different to {"off", "no-version", "no
- version"}, e.g., by options("StartupBanner" = "complete") or by options("StartupBanner"
= NULL) or by options("StartupBanner" = "something else").
## Detailed information about which packages are currently loaded or attached
at which version (regardless of whether these have start-up messages managed
by this package) can be obtained by "sessionInfo()".
## Loading required package:  sfsmisc
## *** Object Oriented Implementation of Distributions ***
##
## Attaching package:  'distr'
## The following objects are masked from 'package:stats':
##
```

```
##      df, qqplot, sd
## *** Simulation Classes Based on Package 'distr' ***
##
## Attaching package:  'distrSim'
## The following object is masked from 'package:stats':
##
##      simulate
## The following object is masked from 'package:base':
##
##      rbind
## *** Estimation and Testing Classes Based on Package 'distr' ***
## Loading required package:  distrEx
## *** Extensions of Package 'distr' ***
##
## Attaching package:  'distrEx'
## The following objects are masked from 'package:stats':
##
##      IQR, mad, median, var
## Loading required package:  distrTeach
## *** Extensions of Package 'distr' for Teaching
## Stochastics/Statistics in Secondary School ***
## Loading required package:  distrMod
## Loading required package:  RandVar
## *** Implementation of Random Variables ***
## Loading required package:  MASS
## Loading required package:  stats4
## *** Object Oriented Implementation of Probability Models ***
##
## Attaching package:  'distrMod'
## The following object is masked from 'package:stats4':
##
##      confint
## The following object is masked from 'package:stats':
##
##      confint
## The following object is masked from 'package:base':
##
##      norm
```

```
require(distr)
N <- Norm(mean = 2, sd = 1.3)
P <- Pois(lambda = 1.2)
Z <- 2*N + 3 + P
Z

## Distribution Object of Class: AbscontDistribution

## Warning in methods::show(x):  arithmetics on distributions are understood
as operations on r.v.'s
## see 'distrARITH()'; for switching off this warning see '?distroptions'

plot(Z)
```
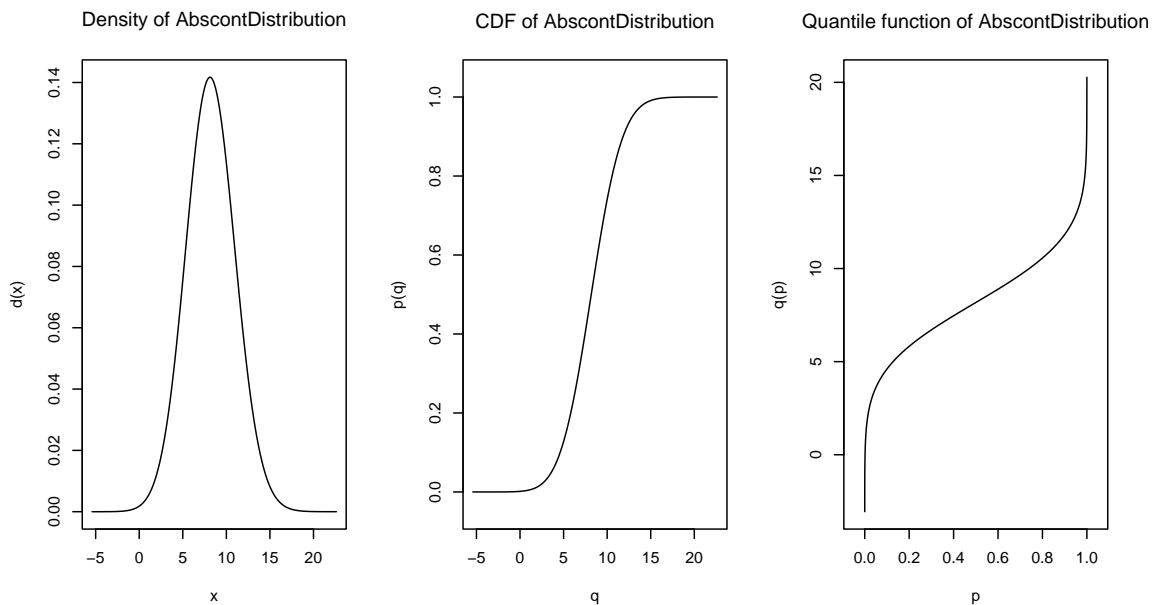


p(Z)(0.4)

## [1] 0.002415402

q(Z)(0.3)

## [1] 6.705068

## in RStudio or Jupyter IRKernel, use q.l(.)(.) instead of q(.)(.)

7

```
Zs <- r(Z)(50)
Zs

##  [1]  8.4303103 15.4731004  4.5682032 11.1601514  5.5728035  9.8973480
##  [7]  4.5698884  9.4473478  9.4832590  5.8930930  5.2690375  9.9522166
## [13]  9.1883037  8.6875770 10.4881532  6.8476457 12.5965248  9.0848308
## [19]  7.3835435 10.0116158 11.4744459 13.3854843 10.4183203  7.5898847
## [25]  8.7720097  8.8094360  5.1426333 12.7033233  4.6559936  6.4193488
## [31] 12.7836665 13.2456751  9.7278791  6.4084225  7.8170125  9.4071120
## [37]  7.8607373  5.1310282  3.4230735  8.8638229 11.0838559 10.4881485
## [43]  8.5232069  5.5479968 11.7465655  0.7520342  3.0165698  5.1382840
## [49]  8.9760909  2.7652583
```

In the environments of RStudio, see https://posit.co and Jupyter IRKernel, see https://github.com/IRkernel/IRkernel, calls to q are caught away from standard R evaluation and are treated in a non-standard way. This non-standard evaluation in particular throws errors at calls to our accessor methods q to slot q of the respective distribution object. For RStudio, this holds for calls from the console (i.e., in the .GlobalEnv environment) only, while calls to our accessor q in different environment are not affected. This is not true in IRKernel, where all calls to our accessor q are affected. To amend this, from version 2.6 on, we provide function q.l (for left-continuous quantile function) as alias to our accessors q, so that all our package functionality also becomes available in RStudio and IRKernel. The same holds, albeit with less consequences in RStudio and IRKernel for accessor p: Here we have an alias p.r (for the right-continuous cdf) for it.

**Comment:**
Let N an object of class "Norm" with parameters **mean**=2, **sd**=1.3 and let P an object of class "Pois" with parameter lambda=1.2. Assigning to Z the expression 2*N+3+P, a new distribution object is generated —of class "AbscontDistribution" in our case— so that identifying N, P, Z with random variables distributed according to N, P, Z, $\mathcal{L}(Z) = \mathcal{L}(2 * N + 3 + P)$, and writing p(Z)(0.4) we get $P(Z \leq 0.4)$, q(Z)(0.3) the 30%-quantile of Z, and with r(Z)(50) we generate 50 pseudo random numbers distributed according to Z, while the **plot** command generates the above figure.

# 1    Concept

In developing our packages, we had the following principles in mind: We wanted to be open in our design so that our classes could easily be extended by any volunteer in the R community to provide more complex classes of distributions as multivariate distributions, times series distributions, conditional distributions. As an exercise, the reader is encouraged to

implement extrem value distributions from the package `"evd"`[1]. The largest effort will in fact be the documentation. . .

We also wanted to preserve naming and notation from `R`-`"stats"` as far as possible so that any programmer used to `S` could quickly use our package. Even more so, as the distributions already implemented to `R` are all well tested and programmed with skills we lack, we use the existing `r`, `d`, `p`, and `q`-functions wherever possible, only wrapping them by small code sniplets to our class hierarchy.

Third we wanted to use a suggestive notation for our automatically generated methods `r`, `d`, `p`, and **q**, which we think is now largely achieved. All this should make intensive use of object orientation in order to be able to use inheritance and method overloading. Let us briefly explain why we decided to realize `r`, `d`, `p`, and **q** as part of our class definitions: Doing so, we place ourselves somewhere between pure object orientation where methods would be *slots* —in the language of the `S4`-concept, confer [2]— and the `S4` paradigm where methods "live their own life" apart from the classes, or, to **q**, which should be regarded use [1]'s terminology, we use COOP[2]-style for `r`, `d`, `p`, and **q** methods, and FOOP[3] -style for `"normal"` methods.

The `S4`-paradigm with methods which are not attached to an object but rather behave differently according to the classes of their arguments is fine if there are particular user-written methods for only some few general distribution classes like AbscontDistribution, as in the case for **plot** or `"+"` (c.f. [5], Section 2.2). During a typical `R` session with `"distr"`, however, there will be a lot of, mostly automatically generated objects of our distribution classes, each with its own `r`, `d`, `p`, and **q**; this even applies to intermediate expressions like 2∗N, 2∗N+3 to eventually produce Z in the example in the motivation. Treating `r`, `d`, `p`, and **q** as generic functions, we would need to generate new classes for each expression 2∗N, 2∗N+3, Z and, correspondingly, particular `S4`-methods for `r`, `d`, `p`, and **q** for each of these new classes; apparently, this would produce overly many classes for an effective inheritance structure.

In providing arithmetics for distributions, we have to deviate a little from the paradigm of `S` as a functional language: For operators like "+", additional parameters controlling the precision of the results cannot be handily passed as arguments. For this purpose we provide global options which may be inspected and modified by distroptions, getdistrOption[4] in complete analogy to **options**, **getOption**. Finally our concept as to parameters: Contrary to the standard `R`-functions like **rnorm** we only permit length 1 for parameters like **mean**, because we see the objects as implementations of univariate random variables, for which

---

[1]a solution to this "homework" may be found in the sources to `"distrEx"`, resp. from version 2.4 on, in the sources to `"RobExtremes"`

[2]class-object-orientated programming, as e.g. in `C++`

[3]function-object-orientated programming, as in the `S4`-concept

[4]Upto version 0.4-4, we used a different mechanism to inspect/modify global options of `"distrEx"` (see section 5.2); corresponding functions distrExoptions, getdistrExOption for package `"distrEx"` are available from version 1.9 on.

vector-valued parameters make no sense; rather one could gather several objects with possibly different parameters to a vector/list of distributions. Of course, the original functions **rnorm** etc. remain unchanged and still allow for vector-valued parameters. Kouros Owzar in an off-list mail raised the point, that in case of multiple parameters as in case of the normal or the Γ-distribution, it might be useful to be able to pass these multiple parameters in vectorized form to the generating function. We, too, think that this is a good idea, but have shifted this question to the new extension package **"distrMod"** which covers more general treatment of statistical models, see section 4.

## 2   Organization in classes

Loosely speaking we have three large groups of classes: distribution classes (in **"distr"**), simulation classes (in **"distrSim"**) and an evaluation class (in **"distrTEst"**), where the latter two are to be considered only as tools which allow a unified treatment of simulations and evaluation of statistical estimation (perhaps also tests and predictions later) under varying simulation situations. Additionally, package **"distrEx"** provides classes for discrete multivariate distributions and for factorized, conditional distributions, as well as a bundle of functionals and distances (see below).

### 2.1   Distribution classes

The purpose of the classes derived from the class Distribution is to implement the concept of a r.v./distribution as such in R.

All classes derived from Distribution have a slot param for a parameter, a slot img for the range and the constitutive slots r, d, p, and q.

From version 1.9 on, up to arguments referring to a parameter of the distribution (like **mean** for the normal distribution), these function slots have the same arguments as those of package **"stats"**, i.e.; for a distribution object X we may call these functions as

- r(X)(n)        —except for objects of class Hyper, where there is a slot n already, so here the argument name to r is nn.

- d(X)(x, **log** = FALSE)

- p(X)(**q**, **lower**.tail = TRUE, **log**.p = FALSE)

- **q**(X)(p, **lower**.tail = TRUE, **log**.p = FALSE)

For the arguments of these function slots see e.g. **rnorm** from package **"stats"**. Note that, as usual, slots d, p, and **q** are vectorized in their first argument, but are not on the subsequent ones. The idea is to gain higher precision for the upper tails or when multiplying probabilities.

### 2.1.1 Subclasses

To begin with, we consider univariate distributions giving `S4`-class UnivariateDistribution, and as typical subclasses, we introduce classes for absolutely continuous and discrete distributions —AbscontDistribution and DiscreteDistribution.

The former, from version 1.9 on, has a slot gaps of class OptionalMatrix, i.e.; an object which may either be NULL or a **matrix**. This slot, if non-NULL, contains left and right endpoints of intervals where the density of the object is 0. This slot may be inspected by the accessor gaps() and modified by a corresponding replacement method. It may also be filled automatically by setgaps(object, exactq = 6, ngrid = 50000), where upon evaluation of the d-slot on a grid of length ngrid, all regions in the range[5] of the distribution where the density is smaller than $10^{-\text{exactq}}$ are set to gaps. Internally, we have helper functions .consolidategaps to merge adjacent intervals and mergegaps to merge slots of different objects.
For saved objects from earlier versions, we provide the functions isOldVersion and conv2NewVersion to check whether the object was generated by an older version of this package and to convert such an object to the new format, respectively.

Class DiscreteDistribution has a slot support, a vector containing the support of the distribution, which is truncated to the lower/upper TruncQuantile in case of an infinite support. TruncQuantile is a global option of `"distr"` described in section 5. For version 2.8.0 on, it has an additional internal slot .finSupport which is a logical of length 2. The first entry says if the left endpoint of the distribution is finite, the second if the right endpoint is finite.
Also from version 1.9 on, class DiscreteDistribution has a subclass LatticeDistribution for supports consisting of[6] an affine linear lattice of form $p + iw$ for $p \in \mathbb{R}$, $w \in \mathbb{R}$, $w \neq 0$ and $i = 0, 1, \ldots, L$, $L \in \mathbb{N} \cup \infty$. This class gains a slot lattice of class Lattice (see below). The purpose of this class is mainly its use in DFT/FFT methods for convolution. Slot lattice may be inspected by the usual accessor function lattice(). As by inheritance, all subclasses of LatticeDistribution which prior to version 1.9 were direct subclasses of DiscreteDistribution gain a slot lattice, too, we provide again isOldVersion and conv2NewVersion methods to check whether the object was generated by an older version of this package and to convert such an object to the new format, respectively. Also note that internally, we suppress lattice points from the support where the probability is 0.

Objects of classes LatticeDistribution resp. DiscreteDistribution, and from version 2.0 on, also AbscontDistribution, may be generated using the generating functions LatticeDistribution() resp. DiscreteDistribution() resp. AbscontDistribution(); see also the corresponding help. E.g., to produce a discrete distribution with support $(1, 5, 7, 21)$ with corresponding probabilities

---

[5]more precisely: between lower and upper TruncQuantile; TruncQuantile is a global option of `"distr"` described in section 5

[6]or at least if filled with points carrying no mass have a representation as an affine linear lattice

$(0.1, 0.1, 0.6, 0.2)$ we may write

```
D <- DiscreteDistribution(supp = c(1,5,7,21), prob = c(0.1,0.1,0.6,0.2))
D

## Distribution Object of Class: DiscreteDistribution

plot(D)
```
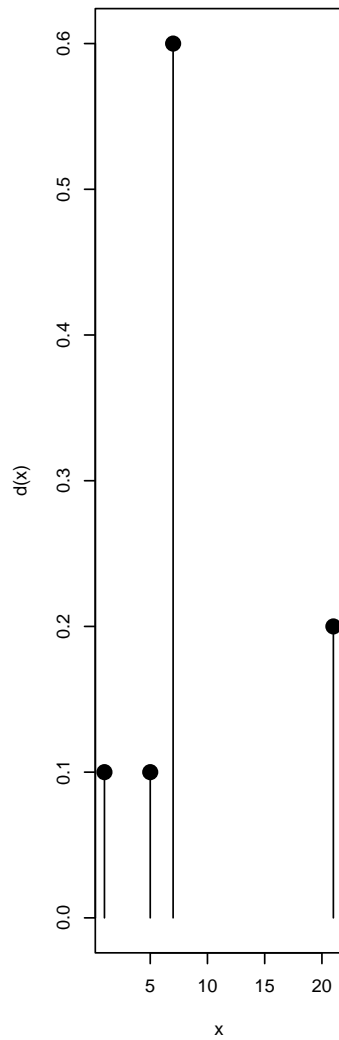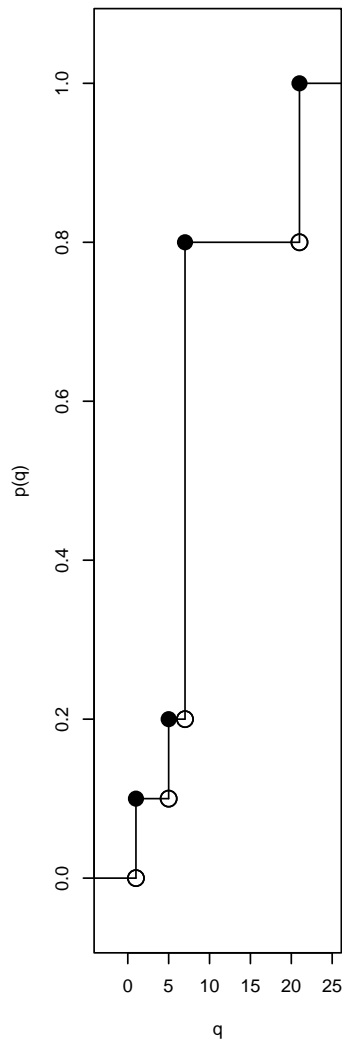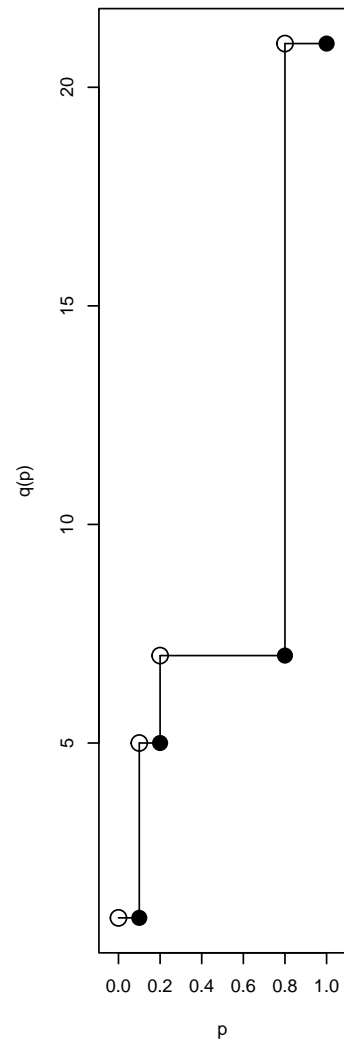


and to generate an absolutely continuos distribution with density proportional to $e^{-|x|^3}$,

we write

```
AC <- AbscontDistribution(d = function(x) exp(-abs(x)^3), withStand = TRUE)
AC

## Distribution Object of Class: AbscontDistribution

plot(AC)
```
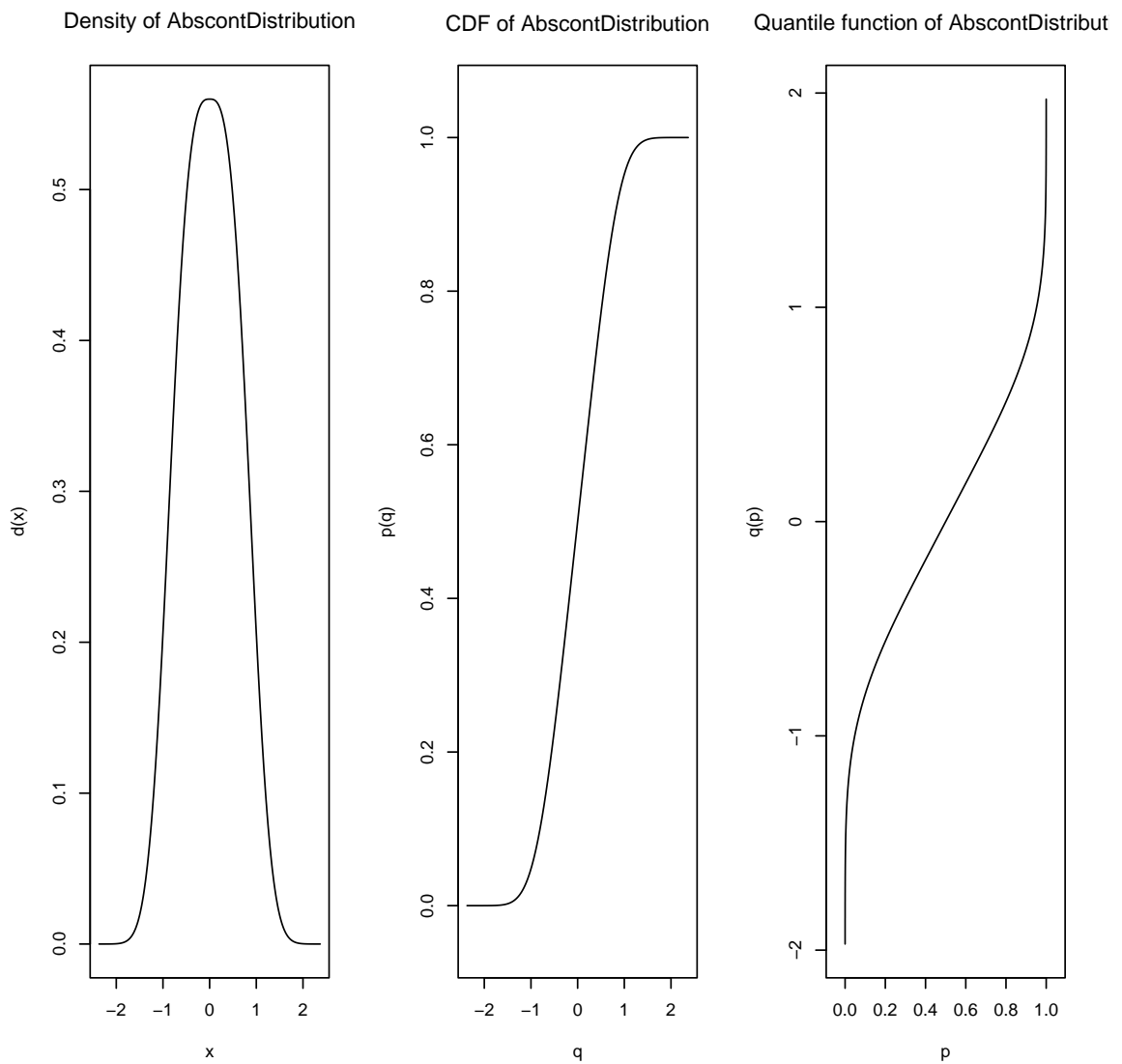


As subclasses of these absolutely continuous and discrete classes, we have implemented all

parametric families which already exist in the `"stats"` package of R in form of `[prefix]<name>` functions —by just providing wrappers to the original R-functions.

Schematically, the inheritance relations as well as the slots of the corresponding classes may be read off from Figure 1. Class `LatticeDistribution` and slot `gaps`, as well as additional classes `AffLinAbscontDistribution`, `AffLinDiscreteDistribution`, `AffLinLatticeDistribution` (c.f. section 3.2) are still lacking in this graphic so far, however, as well as the classes introduced in version 2.0.

The most powerful use of our package probably consists in operations to automatically generate new slots `r`, `d`, `p`, and `q` —induced by mathematical transformations. This is discussed in some detail in subsection 3.

### 2.1.2 Classes for Mixture Distributions

**Lists of distributions** As a first step, we allow distributions to be gathered in lists, giving classes `DistrList` and `UnivarDistrList`, where in case of the latter, all elements must be univariate distributions. For these, the usual indexing operations with `[[.]]` are available. As we will use these lists to construct more general mixture distributions in some subsequent versions, we have moved these routines to package `"distr"` from version 1.9 on.

**Mixing distributions** To be able to work with distributions which are neither purely absolutely continuous nor purely discrete, like e.g. the distribution of $\min(X, 1)$ for $X \sim \mathcal{N}(0,1)$, from package version 2.0 on, we support mixtures of distributions. These are realized as subclasses of class `UnivariateDistribution`. To begin with, we introduce a class `UnivarMixingDistribution` as subclass of class `UnivariateDistribution` which additionally has two slots `MixCoeff` and `MixDistr`. While the former is a numeric vector taking up the mixture coefficients of the distribution, the latter is an object of class `UnivarDistrList` as described below, taking up the distributions of the mixture components; as usual, these slots have their respective accessor and replacement functions. Usually, this mixing distribution will neither have a Lebesgue density nor be purely discrete, having a counting density. So slot `d` as a rule will be empty. Objects of this class may be generated by the generating function `UnivarMixingDistribution()`, see also the corresponding help. In addition there is the function `flat.mix` to simplify such an object converting it to an object of class `UnivarLebDecDistribution`; confer subsection 3.3. Note that these mixing distributions may be recursive, i.e. components of slot `MixDistr` may again be of class `UnivarMixingDistribution`.

```
library(distr)
M1 <- UnivarMixingDistribution(Norm(), Pois(lambda=1), Norm(),
      withSimplify = FALSE)
M2 <- UnivarMixingDistribution(M1, Norm(), M1, Norm(), withSimplify = FALSE)
M2
```

14
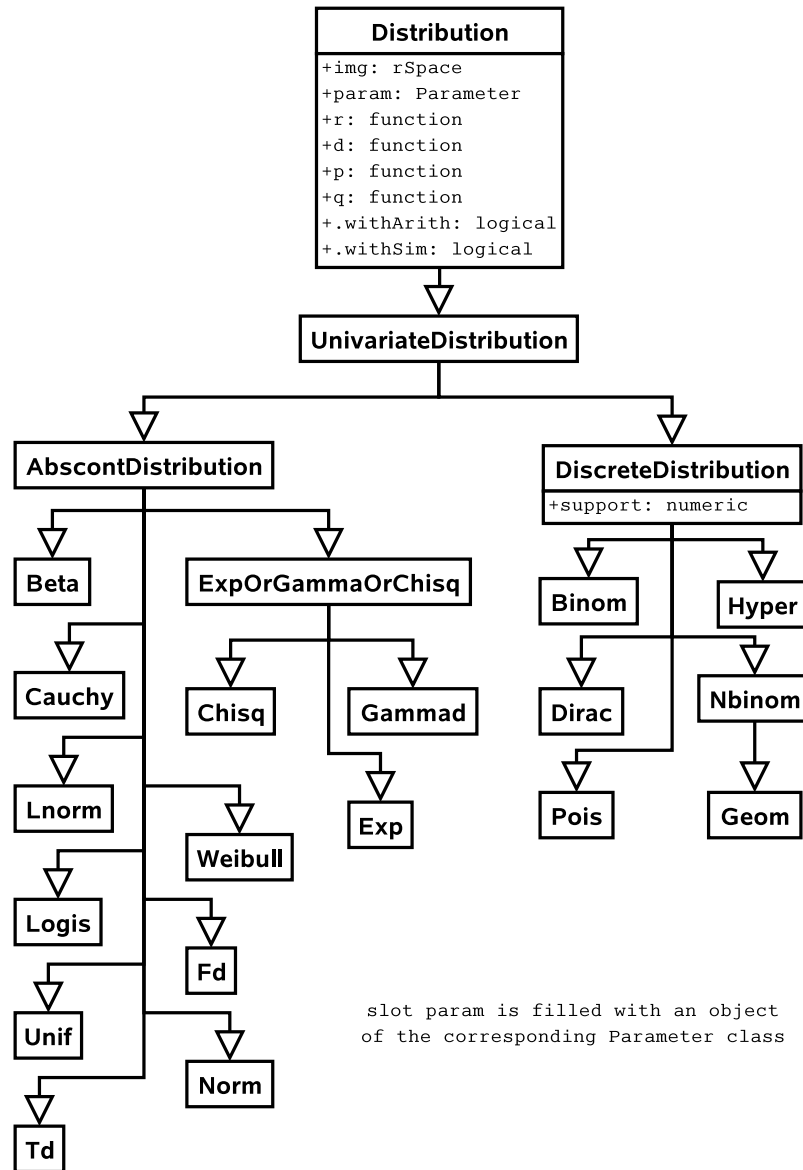
Figure 1: Inheritance relations and slots of the corresponding (sub-)classes for Distribution where we do not repeat inherited slots

```
## An object of class "UnivarMixingDistribution"
##   ---------------------------------------------
##   It consists of  4 components
##   Components:
##   [[1]]An object of class "UnivarMixingDistribution"
```

```
##          :---------------------------------------------
##          :It consists of   3 components
##          :Components:
##          :[[1]]Distribution Object of Class: Norm
##          :        :mean: 0
##          :        :sd: 1
##          :[[2]]Distribution Object of Class: Pois
##          :        :lambda: 1
##          :[[3]]Distribution Object of Class: Norm
##          :        :mean: 0
##          :        :sd: 1
##          :---------------------------------------------
##          :Weights:
##          :0.333000        :0.333000        :0.333000        :
##   ---------------------------------------------
##   [[2]]Distribution Object of Class: Norm
##          :mean: 0
##          :sd: 1
##   [[3]]An object of class "UnivarMixingDistribution"
##          :---------------------------------------------
##          :It consists of   3 components
##          :Components:
##          :[[1]]Distribution Object of Class: Norm
##          :        :mean: 0
##          :        :sd: 1
##          :[[2]]Distribution Object of Class: Pois
##          :        :lambda: 1
##          :[[3]]Distribution Object of Class: Norm
##          :        :mean: 0
##          :        :sd: 1
##          :---------------------------------------------
##          :Weights:
##          :0.333000        :0.333000        :0.333000        :
##   ---------------------------------------------
##   [[4]]Distribution Object of Class: Norm
##          :mean: 0
##          :sd: 1
##   ---------------------------------------------
##   Weights:
##   0.250000 0.250000 0.250000 0.250000
```

**Lebesgue Decomposed distributions**  As seen in the above example of $\min(X, 1)$, classes DiscreteDistribution and Abscontdistribution are not closed under arithmetic operations. To have such a closure, from version 2.0 on, we introduce class UnivarLebDecDistribution, which realizes a Lebesgue decomposition of a univariate distribution into a discrete and an absolutely continuous distribution. Of course, we still cannot cover distributions having a non-trivial continuous but not absolutely continuous part like the Cantor distribution, but class UnivarLebDecDistribution provides a sufficiently general compromise. Class UnivarLebDecDistribution is a subclass of class UnivarMixingDistribution, where in addition we assume that both slots MixCoeff and MixDistr are of length 2, and that the first component of slot MixDistr is of class AbscontDistribution while the second is of class DiscreteDistribution. For this class there are particular accessors acWeight, discreteWeight for the respective weights and acPart, discretePart for the respective distributions. Again there is a generating function UnivarMixingDistribution(). In addition there is the function flat .LCD to simplify such an object converting it to an object of class UnivarLebDecDistribution; confer subsection 3.3.Classes AbscontDistribution, DiscreteDistribution and UnivarLebDecDistribution are grouped to a virtual class (more specifically a class union) AcDcLcDistribution.

**Compound distributions**  From version 2.1 on, we also support compound distributions, i.e. the distributions $D$ of form $D = \mathcal{L}(\sum_{i=1}^{N} X_i)$, $X_i \overset{\text{i.i.d.}}{\sim} F$, a distribution on $\mathbb{R}$, and, independent of the $X_i$, $N \sim Q$ a distribution on $\mathbb{N}_0$. These distributions are implemented as class CompoundDistribution which is a subclass of class UnivarMixingDistribution; in addition this class has two slots, NumbOfSummandsDistr, the distribution $Q$ of the number of summands (or frequency distribution) and SummandsDistr the distribution $F$ of the summands. Correspondingly, in package **"distrEx"** there are specialized expectation and variance methods.

### 2.1.3   Classes for multivariate distributions and for conditional distributions

In **"distrEx"**, we provide the following classes for handling multivariate distributions:

**Multivariate distribution classes**  Multivariate distributions are much more complicated than univariate ones, which is why but a few exceptional ones have already been implemented to R in packages like **"multnorm"**. In particular it is not so clear what a slot q should mean and, in higher dimensions slot p, and possibly also slot d may become awkward. So, for multivariate distributions, realized as class MultivariateDistribution, we only insist on slot r, while the other functional slots may be left void.

The easiest case is the case of a discrete multivariate distribution with finite support which is implemented as class DiscreteMVDistribution.

17

**Conditional distribution classes** Also arising in multivariate settings only are conditional distributions. In our approach, we realize factorized, conditional distributions where the (factorized) condition is in fact treated as an additional parameter to the distribution. The condition is realized as an object of class Condition, which is a slot of corresponding classes UnivariateCondDistribution. This latter is the mother class to classes AbscontCondDistribution and DiscreteCondDistribution. The most important application of these classes so far is regression, where the distribution of the observation given the covariates is just realized as a UnivariateCondDistribution.

### 2.1.4 Parameter classes

As most distributions come with a parameter which often is of own interest, we endow the corresponding slots of a distribution class with an own parameter class, which allows for some checking like "Is the parameter lambda of an exponential distribution non-negative?", "Is the parameter size of a binomial a positive integer?"
Consequently, we have a method liesIn that may answer such questions by a TRUE/FALSE statement. Schematically, the inheritance relations of class Parameter as well as the slots of the corresponding (sub-)classes may be read off in Figure 2 where we do not repeat inherited slots. The most important set to be used as parameter domain/sample space (rSpace) will be an Euclidean space. So rSpace and EuclideanSpace are also implemented as classes, the structure of which may be read off in Figure 3.

From version 1.9 on, we also have a subclass Lattice, which is still lacking in the preceding figure. It has slots pivot (of class "numeric"), width (of class "numeric" but tested against "==0") and Length (of class "numeric" but tested to be an integer ">0" or Inf). All slots may be inspected/modified by the usual accessor/replacement functions.

## 2.2 Simulation classes

From version 1.6 on, the classes and methods of this subsection are available in package "distrSim".

The aim of simulation classes is to gather all relevant information about a simulation in a correspondingly designed class. To this end we introduce the class Dataclass that serves as a common mother class for both "real" and simulated data. As derived classes we then have a simulation class where we also gather all information needed to reconstruct any particular simulation.
From version 1.8 of this package on, we have changed the format how data / simulations are stored: In order to be able to cope with multivariate, regression and (later) time series distributions, we have switched to the common array format samplesize x obsDim x runs where obsDim is the dimension of the observations. For saved objects from earlier versions, we provide the functions isOldVersion and conv2NewVersion to check whether the object was generated by an older version of this package and to convert such an object to the new
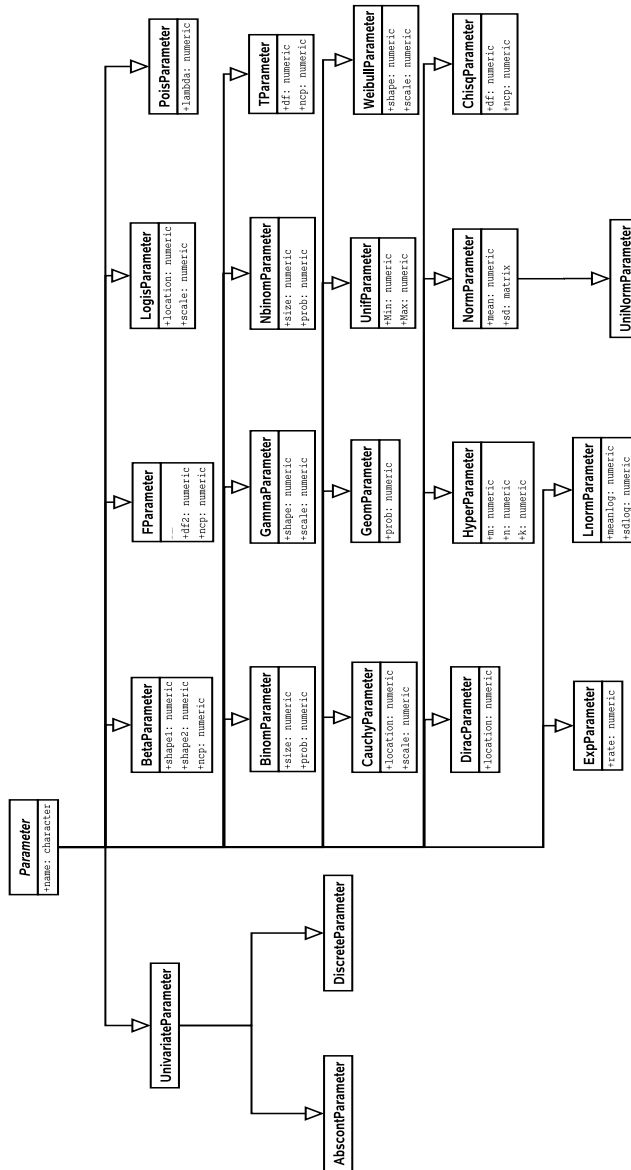
Figure 2: Inheritance relations and slots of the corresponding (sub-)classes for Parameter

format, respectively. For objects generated from version 1.8 on, you get the package version of package **"distrSim"**, under which they have been generated by a call to getVersion(). Finally, coming from robust statistics we also consider situations where the majority of the data stems from an ideal situation/distribution whereas a minority comes from a contaminating source. To be able to identify ideal and contaminating observations, we also store
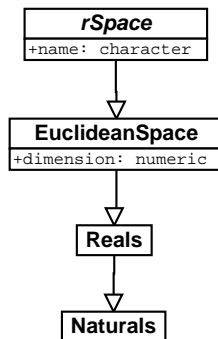
Figure 3: Inheritance relations and slots of the corresponding (sub-)classes for rSpace

this information in an indicator variable.

As the actual values of the simulations only play a secondary role, and as the number of simulated variables can become very large, but still easily reproducible, it is not worth storing all simulated observations but rather only the information needed to reproduce the simulation. This can be done by savedata.

Schematically, the inheritance relations of class Dataclass as well as the slots of the corresponding (sub-)classes may be read off in Figure 4 where we do not repeat inherited slots. Also, analogously to package "distr", global options for the output by methods plot and
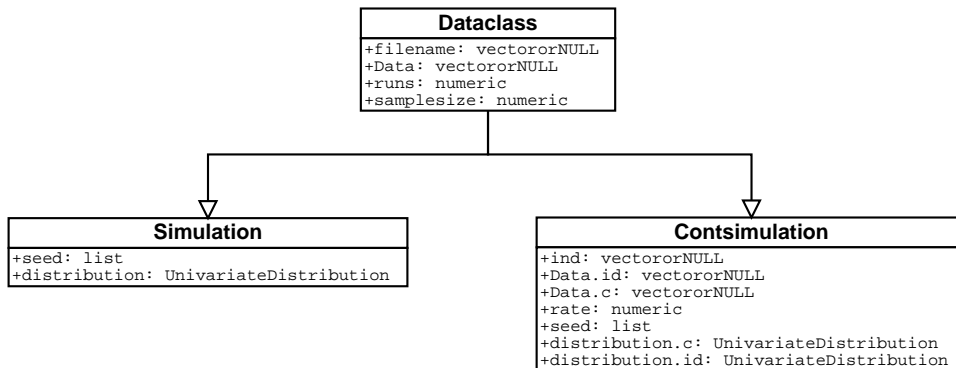


Figure 4: Inheritance relations and slots of the corresponding (sub-)classes for Dataclass

summary are controlled by distrSimoptions() and getdistrSimoptions()

## 2.3 Evaluation class

From version 1.6 on, the class and methods of this subsection are available in package
"distrTEst".

When investigating properties of a new procedure (e.g. an estimator) by means of simulations, one typically evaluates this procedure on a large set of simulation runs and gets a result for each run. These results are typically not available within seconds, so that it is worth storing them. To organize all relevant information about these results, we introduce a class Evaluation the slots of which is filled by method evaluate —see subsection 3.14. Schematically, the slots of this class may be read off in Figure 5. A corresponding savedata

| **Evaluation** |
| --- |
| +name: character |
| +filename: character |
| +call.ev: call |
| +result: vectororNULL |
| +estimator: OptionalFunction |

Figure 5: Slots of class Evaluation

method saves the object of class Evaluation in two files in the R-working directory: one using the filename <filename> also stores the results; the other one, designed to be "human readable", comes as a comment file with filename <filename>.**comment** only stores the remaining information. The filename can be specified in the optional argument fileN to savedata; by default it is concatenated from the filename slot of the Dataclass object and <estimatorname>, which you may either pass as argument estimatorName or by default is taken as the R-name of the corresponding R-function specified in slot estimator.

From version 1.8 on, slot result in class Evaluation is of class DataframeorNULL, i.e.; may be either a data frame or NULL, and slot **call**.ev in class Evaluation is of class "CallorNULL", i.e.; may be either a call or NULL. Also, we want to gather Evaluation objects in a particular data structure EvaluationList (see below), so we have to be able to check whether all data sets in the gathered objects coincide. For this purpose, from this version on, class Evaluation has an additional slot Data of class Dataclass. In order not to burden the objects of this class too heavily with uninformative simulated data, in case of a slot Data of one of the simulation-type subclasses of Dataclass, this Data itself has an empty Data-slot.

## 2.4 EvaluationList class

The class and methods of this subsection are available in package `"distrTEst"`.
In order to compare different procedures / estimators for the same problem, it is natural to gather several Evaluation objects with results of the same range (e.g. a parameter space) generated on the same data, i.e.; on the same Dataclass object. To this end, from version 1.8 on, we have introduced class EvaluationList. Schematically, the slots of this class may be read off in Figure 6. The common Data slot of the Evaluation objects in an EvaluationList

| **EvaluationList** |
|---|
| +name: character |
| +Elist: list |
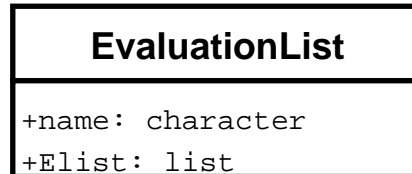
Figure 6: Slots of class EvaluationList

object may be accessed by the accessor method Data.

# 3 Methods

## 3.1 Arithmetics

We have made available quite general arithmetical operations to our distribution objects, generating new image distributions automatically. In this context some comments are due as to the interpretation of corresponding arithmetic expressions of distribution objects:

CAVEAT: **These arithmetics operate on the corresponding r.v.'s and not on the distributions.**

(For the latter, they only would make sense in restricted cases like convex combinations).

Martin Mächler pointed out that this might be confusing. So, this warning is also issued on attaching package `"distr"`, and, by default, again whenever a Distribution object, produced by such arithmetics is shown or printed; this also applies to the last line in

```
A1 <- Norm(); A2 <- Unif()
A1 + A2

## Distribution Object of Class: AbscontDistribution
```

```
## Warning in methods::show(x):  arithmetics on distributions are understood
as operations on r.v.'s
## see 'distrARITH()'; for switching off this warning see '?distroptions'
```

```
Warning message:
arithmetics on distributions are understood as operations on r.v.'s
see 'distrARITH()'; for switching off this warning see '?distroptions' in:
print(object)
```

This behaviour will soon be annoying so you may switch it off setting the global option
WarningArith to FALSE (see section 5).

Function distrARITH() displays the following comment

```
#######################################################################
#  On arithmetics operating on distributions in package "distr"
#######################################################################


                    Attention:

Special caution is due in the followin issues

%---------------------------------------------------------------------
% Interpretation of arithmetics
%---------------------------------------------------------------------

Arithmetics on distribution objects are understood as operations on
corresponding random variables (r.v.'s) and _not_ on distribution
functions or densities;
e.g.

    sin( Norm() + 3 * Norm() ) + 2

returns a distribution object representing the distribution of the r.v.

    sin(X+3*Y)+2

where X and Y are r.v.'s i.i.d. N(0,1).


%---------------------------------------------------------------------
% Adjusting accuracy
%---------------------------------------------------------------------

Binary operators like "+", "-" would loose their elegant calling
 e1 + e2 if they had to be called with an extra argument controlling
their accuracy. Therefore, this accuracy is controlled by global options.
These options are inspected and set by distroptions(), getdistrOption(),
```

```
see ?distroptions

%---------------------------------------------------------------------
% Multiple instances in expressions and independence
%---------------------------------------------------------------------

Special attention has to be paid to arithmetic expressions of
distributions involving multiple instances of the same symbol:

/-> All arising instances of distribution objects in arithmetic
    expressions are assumed stochastically independent. <-/

As a consequence, whenever in an expression, the same symbol for
an object occurs more than once, every instance means a new
independent distribution.

So for a distribution object X, the expressions X+X and 2*X are _not_
equivalent.

The first means the convolution of distribution X with distribution
X, i.e. the distribution of the r.v. X1 + X2, where X1 and X2 are
identically distributed according to X.
In contrast to this, the second expression means the distribution of
the r.v. 2 X1 = X1 + X1, where again X1 is distributed according to X.

Hence always use 2*X, when you want to realize the second case.

Similar caution is due for X^2 and X*X and so on.

%---------------------------------------------------------------------
% Simulation based results varying from call to call
%---------------------------------------------------------------------

At several instances (in particular for non-monotone functions from group
 Math like sin(), cos()) new distributions are generated by means of
RtoDPQ, RtoDPQ.d, RtoDPQ.LC. In these functions, slots d, p, q are
filled by simulating a large number of random variables, hence they are
stochastic estimates.

So don't be surprised if they will change from call to call.
```

## 3.2   Affine linear transformations

We have overloaded the operators "+", "−", "∗", "/" such that affine linear transformations which involve only single univariate r.v.'s are available; i.e. is expressions like Y=(3∗X+5)/4 are permitted for an object X of class AbscontDistribution or DiscreteDistribution (or some subclass), giving again an object Y of class AbscontDistribution or DiscreteDistribution (in general). Here the corresponding transformations of the d, p, and

q-functions are done analytically.

From version 1.9 on, we use subclasses AffLinAbscontDistribution, AffLinDiscreteDistribution, AffLinLatticeDistribution as classes of the return values to enhance accuracy of functinals like E, var, etc. in package "distrEx". These classes in addition to their counterparts without prefix "AffLin" have slots a, b, and X0, to capture the fact that an object of this class is distributed as a * X0 + b. Also, we introduce a class union AffLinDistribution of classes AffLinAbscontDistribution and AffLinDiscreteDistribution. Consequently, the result Y of Y <−a1 *X + b1 for an object X of (a subclass of) class AffLinDiscreteDistribution (if a !=0) is of the same class as X but with slots Y@a = a1 *X@a, Y@b = b1 + a1 *X@b, Y@X0 = X@X0. In version 2.0, the same principle has been applied to introduce class AffLinUnivarLebDecDistribution. All AffLin-xxx distribution classes are grouped to a virtual class (more specifically a class union) AffLinDistribution.

## 3.3 Decompositions, Flattening and Other Simplifications

**Decompositions:** One of the issues when programming the distribution of the multiplication of independent random variables is that we have to treat positive and negative part (and, if nontrivial, point mass to 0) separately. To this end, from version 2.0 on, there are methods decomposePM to decompose a discrete, an absolutely continuous or a Lebesgue decomposed distribution into its respective parts.

```
decomposePM(Norm())

## $neg
## $neg$D
## Distribution Object of Class: AbscontDistribution
##
## $neg$w
## [1] 0.5
##
##
## $pos
## $pos$D
## Distribution Object of Class: AbscontDistribution
##
## $pos$w
## [1] 0.5

    decomposePM(Binom(2,0.3)-Binom(5,.4))

## $neg
## $neg$D
```

```
## Distribution Object of Class: DiscreteDistribution
##
## $neg$w
## [1] 0.758944
##
##
## $`0`
## $`0`$D
## Distribution Object of Class: Dirac
##  location: 0
##
## $`0`$w
## [1] 0.1780704
##
##
## $pos
## $pos$D
## Distribution Object of Class: DiscreteDistribution
##
## $pos$w
## [1] 0.0629856
```

```
    decomposePM(UnivarLebDecDistribution(Norm(),Binom(2,0.3)-Binom(5,.4),
                acWeight = 0.3))
```

```
## $pos
## $pos$D
## An object of class "UnivarLebDecDistribution"
##  --- a Lebesgue decomposed distribution:
##
##     Its discrete part (with weight 0.227000) is a
##  Distribution Object of Class: DiscreteDistribution
##  This part is accessible with 'discretePart(<obj>)'.
##
##     Its absolutely continuous part (with weight 0.773000) is a
##  Distribution Object of Class: AbscontDistribution
##  This part is accessible with 'acPart(<obj>)'.
##
## $pos$w
## discreteWeight
##      0.1940899
```

```
##
##
## $neg
## $neg$D
## An object of class "UnivarLebDecDistribution"
##  --- a Lebesgue decomposed distribution:
##
##     Its discrete part (with weight 0.780000) is a
##  Distribution Object of Class: DiscreteDistribution
##  This part is accessible with 'discretePart(<obj>)'.
##
##     Its absolutely continuous part (with weight 0.220000) is a
##  Distribution Object of Class: AbscontDistribution
##  This part is accessible with 'acPart(<obj>)'.
##
## $neg$w
## discreteWeight
##      0.6812608
##
##
## $`0`
## $`0`$D
## Distribution Object of Class: Dirac
##  location: 0
##
## $`0`$w
## discreteWeight
##      0.1246493
```

**Simplification by flattening:**    On the other hand, concatenating mathematical operations would easily yield quite complicated structures. A first thing to do is to look whether some components carry mass (approximately) 0. simplifyD uses this to cancel out such components, and if possible return simpler types; see also the help to this function.

Also, sometimes one would like to let collapse a whole list of distributions (as in the MixDistr of a UnivarMixingDistribution object) into a simpler UnivarLebDecDistribution-class form. This is what is done in the the functions flat.mix and flat.LCD.

```
D1 <- Norm()
D2 <- Pois(1)
D3 <- Binom(1,.4)
```

```
D4 <- UnivarMixingDistribution(D1,D2,D3, mixCoeff = c(0.4,0.5,0.1),
      withSimplify = FALSE)
D <- UnivarMixingDistribution(D1,D4,D1,D2, mixCoeff = c(0.4,0.3,0.1,0.2),
      withSimplify = FALSE)
D

## An object of class "UnivarMixingDistribution"
##   --------------------------------------------
##   It consists of   4 components
##   Components:
##   [[1]]Distribution Object of Class: Norm
##         :mean: 0
##         :sd: 1
##   [[2]]An object of class "UnivarMixingDistribution"
##         :--------------------------------------------
##         :It consists of   3 components
##         :Components:
##         :[[1]]Distribution Object of Class: Norm
##         :        :mean: 0
##         :        :sd: 1
##         :[[2]]Distribution Object of Class: Pois
##         :        :lambda: 1
##         :[[3]]Distribution Object of Class: Binom
##         :        :size: 1
##         :        :prob: 0.4
##         :--------------------------------------------
##         :Weights:
##         :0.400000        :0.500000        :0.100000        :
##   --------------------------------------------
##   [[3]]Distribution Object of Class: Norm
##         :mean: 0
##         :sd: 1
##   [[4]]Distribution Object of Class: Pois
##         :lambda: 1
##   --------------------------------------------
##   Weights:
##   0.400000 0.300000 0.100000 0.200000
##   --------------------------------------------

D0<-flat.mix(D)
D0
```

```
## An object of class "UnivarLebDecDistribution"
##  --- a Lebesgue decomposed distribution:
##
##     Its discrete part (with weight 0.380000) is a
##  Distribution Object of Class: DiscreteDistribution
##  This part is accessible with 'discretePart(<obj>)'.
##
##     Its absolutely continuous part (with weight 0.620000) is a
##  Distribution Object of Class: AbscontDistribution
##  This part is accessible with 'acPart(<obj>)'.

## Warning in methods::show(x):  arithmetics on distributions are understood
as operations on r.v.'s
## see 'distrARITH()'; for switching off this warning see '?distroptions'
```

Many arithmetic operations described in the subsequent sections do this simplification on their return value, according to the global option SimplifyD.

**Simplification by collapsing:** Dealing with discrete distributions, several arithmetical/mathematical operations tend to create new quite unequally spaced supports (mostly by joining supports of operands), often accumulating support points somewhere. In order to overcome this, from version 2.1 on, following a proposal by Jacob van Etten, whenever support points get closer to each other than prescribed in global option DistrResolution (see also section 5), and if in addition global option DistrCollapse is TRUE, we collapse these support point, using the median of the collapsed points as new support point, to which we attribute the cumulated probability mass. If in addition global option DistrCollapse.Unique.Warn is TRUE we issue a warning on collapsing occasions.

## 3.4 The group math of unary mathematical operations

Also the group math of unary mathematical operations is available for distribution classes; so expressions like $\mathbf{exp}(\mathbf{sin}(3*X+5)/4)$ are permitted. The corresponding r method consists in simply performing the transformation to the simulated values of X. The corresponding (default-) d, p and q-functions are obtained by simulation, using the technique described in the following subsection.

By means of **substitute**, the bodies of the r, d, p, q-slots of distributions show the parameter values with which they were generated; in particular, convolutions and applications of the group math may be traced in the r-slot of a distribution object, compare
r($\mathbf{sin}$(Norm()) + $\mathbf{cos}$(Unif() $*3 + 2$)).

Initially, it might be irritating that the same "arithmetic" expression evaluated twice in a row gives two different results, compare

```
  A1 <- Norm(); A2 <- Unif()
  d(sin(A1 + A2))(0.1)

## [1] 0.3781077

  d(sin(A1 + A2))(0.1)

## [1] 0.3781077

  sin(A1 + A2)

## Distribution Object of Class: AbscontDistribution

## Warning in methods::show(x):  arithmetics on distributions are understood
as operations on r.v.'s
## see 'distrARITH()'; for switching off this warning see '?distroptions'
## Warning in methods::show(x):  slots d,p,q have been filled using simulations;
for switching off this warning see '?distroptions'
```

This is due to the fact, that all slots are filled starting from simulations. To explain this, a warning is issued by default, whenever a Distribution object, filled by such simulations is shown or printed; this also applies to the last line in the preceding code sniplet. This behaviour may again be switched off by setting the global option WarningSim to FALSE (see section 5).

As they are frequently needed, from version 1.9 on, math operations **abs**(), **exp**(), and —if an R-version ≥ 2.6.0 is used— also **log**() are implemented in an analytically exact form, i.e.; with exact expressions for slots d, p, and q.

## 3.5  Construction of d, p, and q from r

In order to facilitate automatic generation of new distributions, in particular those arising as image distributions under transformations of correspondingly distributed random variables, we provide ad hoc methods that should be overloaded by more exact ones wherever possible. As, at least in principle each of these slots is sufficient for the reconstruction of the other ones, we follow the following strategy:

| d | p | q | r | reconstruction |
|---|---|---|---|---|
| + | + | + | + | no reconstruction necessary |
| + | + | + | − | r as `q(X)(runif(n))` |
| + | + | − | + | q by numerical inversion from p |
| + | + | − | − | q again from p and r again from slot q |
| + | − | + | + | p by numerical integration from d |
| + | − | + | − | p from d, and r from q |
| + | − | − | + | p from d, and q from p |
| + | − | − | − | p from d, q from p and r from q |
| − | + | + | + | d by numerical differentiation (with `D1ss` from package `"sfsmisc"` from p |
| − | + | + | − | d from p, r from q |
| − | + | − | + | d, q from p |
| − | + | − | − | d, q from p, r from q |
| − | − | + | + | p by numerical inversion from q, d from p |
| − | − | + | − | p, r from q, d from p |
| − | − | − | + | use `RtoDPQ` |
| − | − | − | − | not allowed |

More specifically, by means of the function RtoDPQ we first generate $10^{\mathtt{RtoDPQ.e}}$ random numbers where RtoDPQ.e is a global option of this package and is discussed in section 5. A density estimator is evaluated along this sample, the distribution function is estimated by the empirical c.d.f. and, finally, the quantile function is produced by numerical inversion. Of course the result is rather crude as it relies on the law of large numbers only, but this way all transformations within the group math become available. If the input of the transformation is of class UnivarLebDecDistribution, RtoDPQ is replaced by RtoDPQ.LC. In this case, replicated values are taken as belonging to the discrete part, for which the distribution is generated according to the corresponding frequencies with the generating function DiscreteDistribution(). With the remaining, non replicated values, the absolutely continuous part is reconstructed just as with RtoDPQ.

Where laws under transformations can easily be computed exactly —as for affine linear transformations— we replace this procedure by the exactly transformed d, p, q-methods.

## 3.6 Convolution

A convolution method for two independent r.v.'s is implemented by means of explicit calculations for discrete summands, and by means of DFT/FFT[7] if one of the summands is absolutely continuous or (from version 1.9 on:) both are lattice distributed with a common lattice as support. This method automatically generates the law of the sum of two independent variables/distributions $X$ and $Y$ of any univariate distributions —or in S4-jargon: the addition operator "+" is overloaded for two objects of class UnivariateDistribution and corresponding subclasses.

## 3.7 Further Binary Operators

Having implemented a class for Lebesgue decomposed distributions, we have been able to realize further binary operators, in particular we have exact analytical constructions for multiplication, division, exponentiation:

```
A1 <- Norm(); A2 <- Unif()
A1A2 <- A1*A2
plot(A1A2)
```

---

[7]Details to be found in [5]
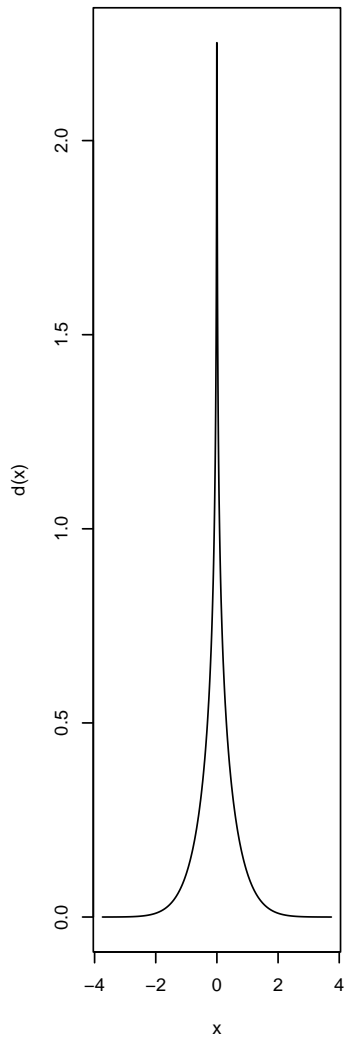
Density of AbscontDistribution     CDF of AbscontDistribution     Quantile function of AbscontDistribut

```
A12 <- 1/(A2 + .3)
plot(A12)
```

Density of AbscontDistribution    CDF of AbscontDistribution    Quantile function of AbscontDistribut

```
B <- Binom(5,.2)+1
A1B <- A1^B
plot(A1B, xlim=c(-3,3))
```

Density of AbscontDistribution        CDF of AbscontDistribution        Quantile function of AbscontDistribut

```
plot(1.2^A1)
```

Density of AbscontDistribution     CDF of AbscontDistribution     Quantile function of AbscontDistribut
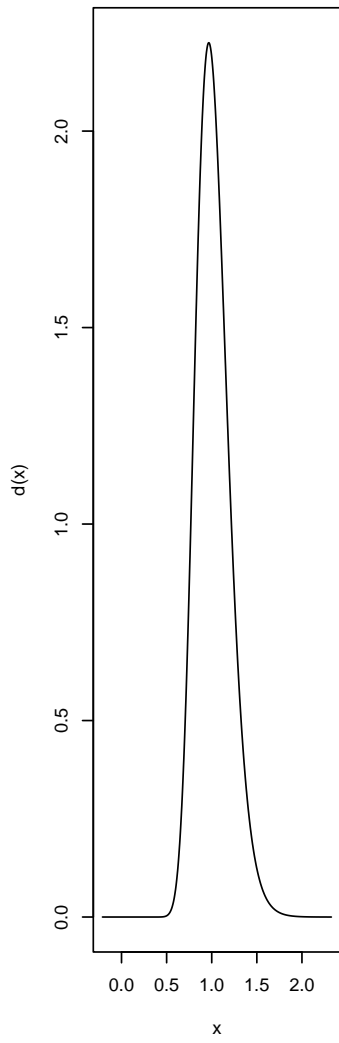
```
plot(B^A1)
```

CDF of UnivarLebDecDistribution      Quantile function of UnivarLebDecDistribution
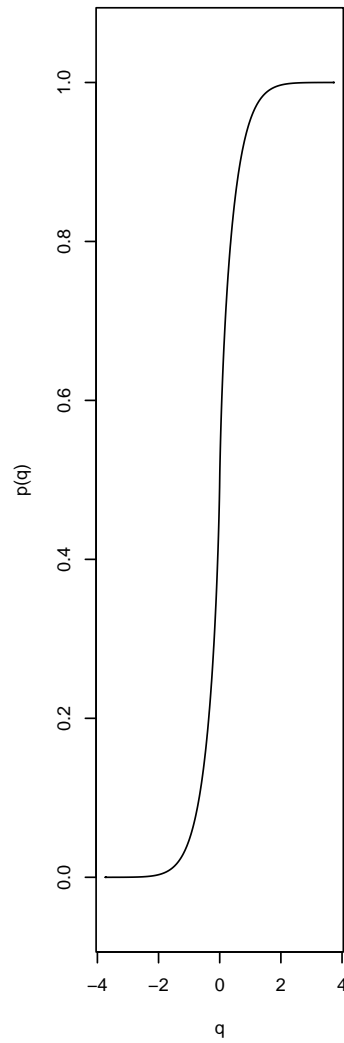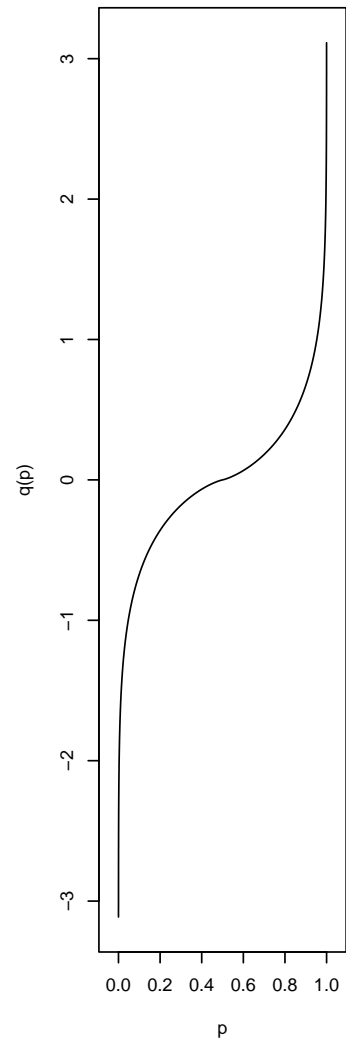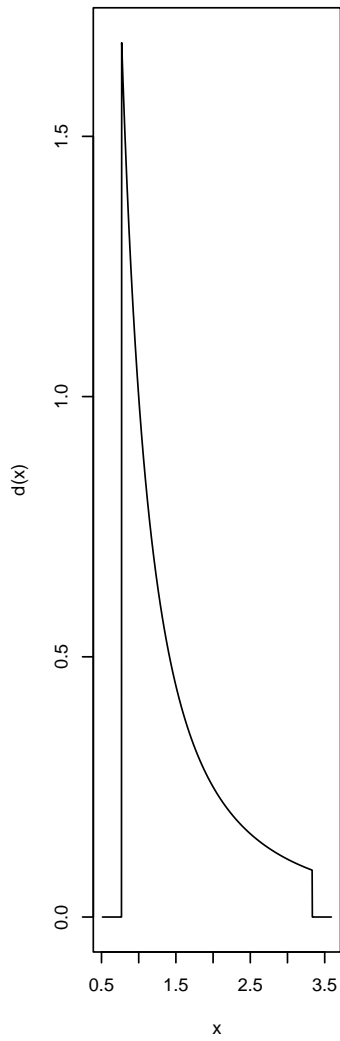
Density of AbscontDistribution    CDF of AbscontDistribution    Quantile function of AbscontDistribution
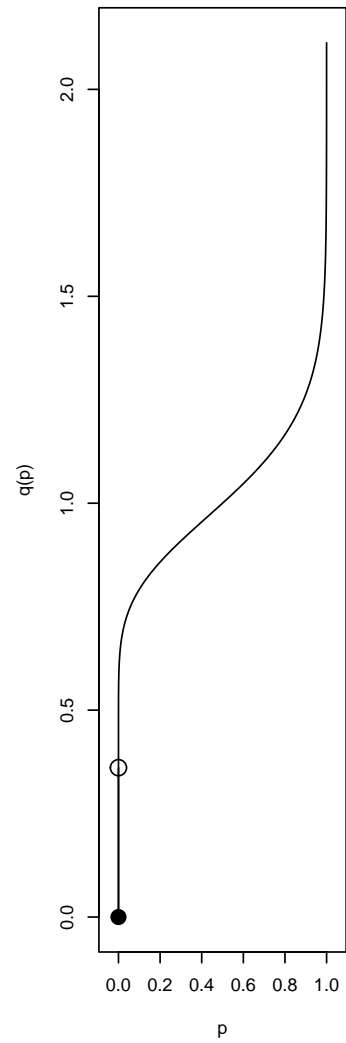
Probability function of Dirac(1)    CDF of Dirac(1)    Quantile function of Dirac(1)
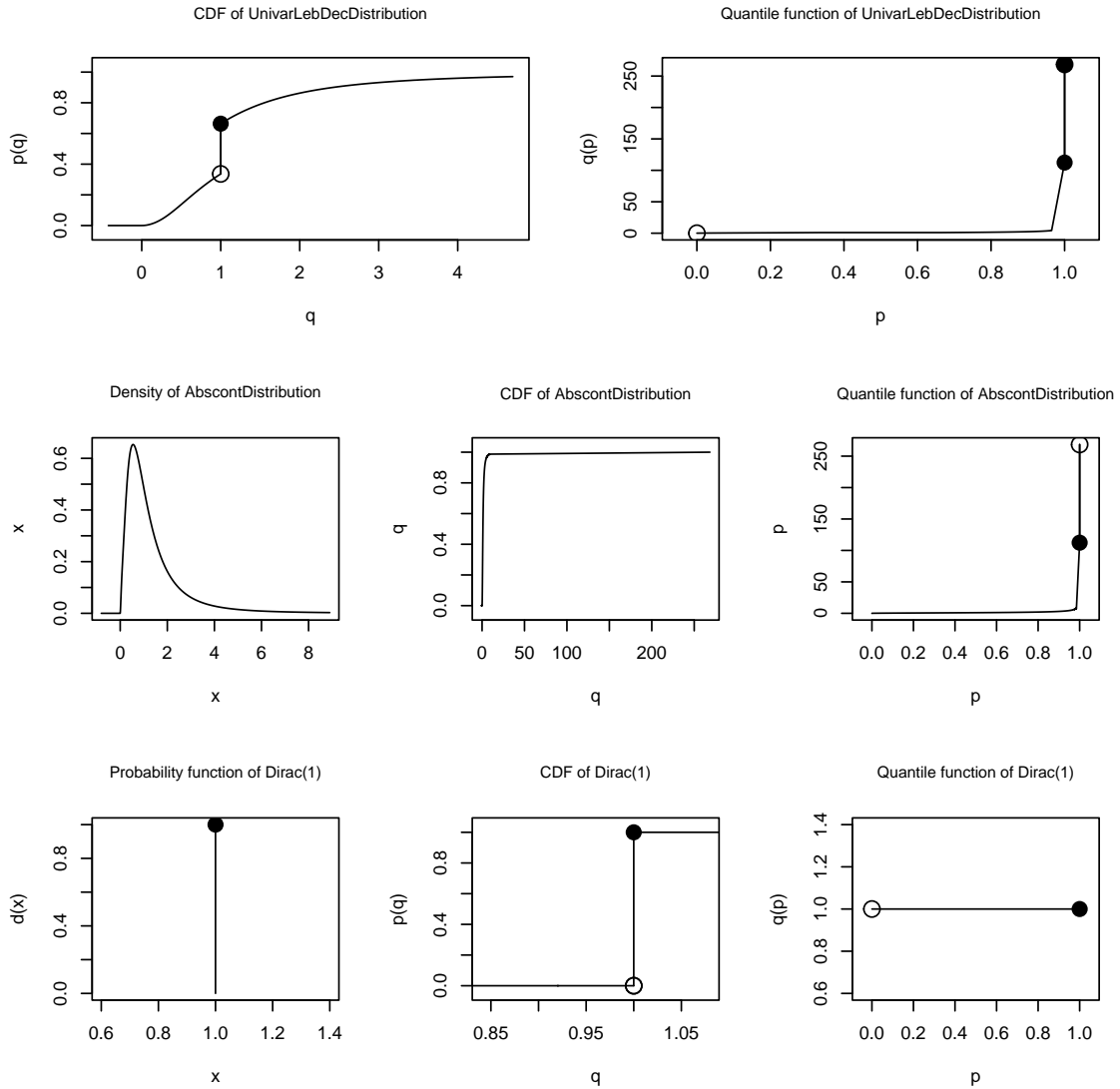
## 3.8    Truncation, Pairwise Minimum/Maximum, Huberization

Up to version 2.0, we have had truncation, Huberization and minimum and maximum of random variables as illustrating demos; in particular the last three could not be realized in a completely satisfactory manour, as Lebesgue decomposed distributions had not been available before. Now these illustrations have moved into the package itself:

```
H <- Huberize(Norm(),lower=-1,upper=2)
plot(H)
```

CDF of AffLinUnivarLebDecDistribution

Quantile function of AffLinUnivarLebDecDistribution

Density of AffLinAbscontDistribution

CDF of AffLinAbscontDistribution

Quantile function of AffLinAbscontDistribution

Probability function of AffLinDiscreteDistribution

CDF of AffLinDiscreteDistribution

Quantile function of AffLinDiscreteDistribution
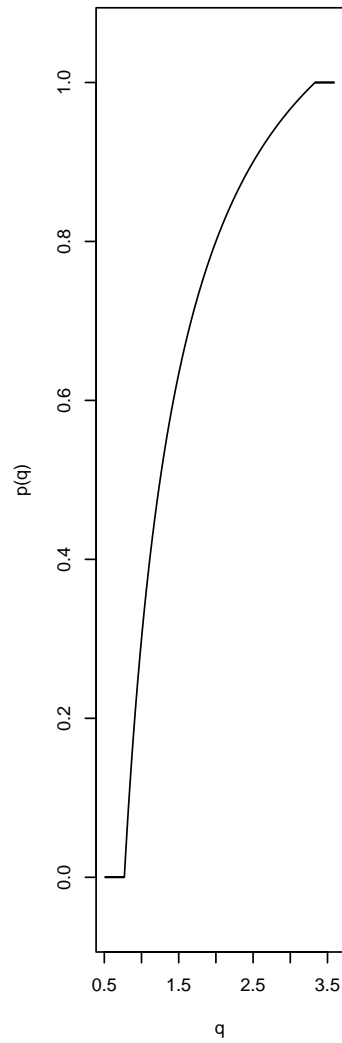
```
T <- Truncate(Norm(),lower=-1,upper=2)
plot(T)
```
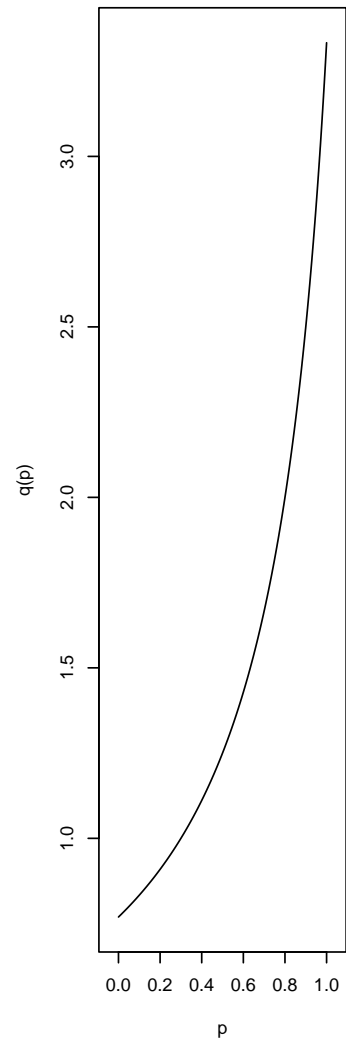
Density of AbscontDistribution    CDF of AbscontDistribution    Quantile function of AbscontDistribut
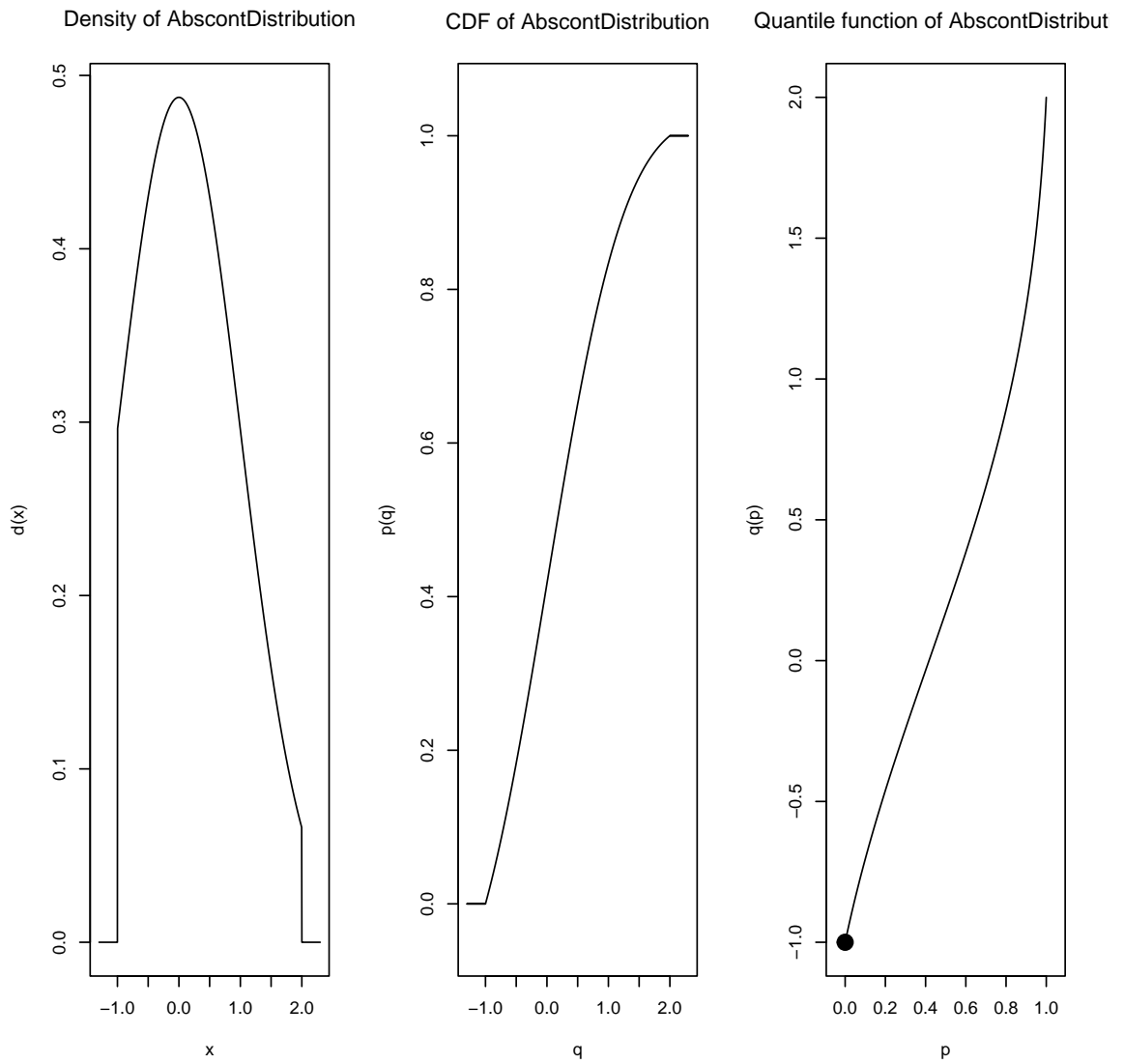
```
M1 <- Maximum(Unif(0,1), Minimum(Unif(0,1), Unif(0,1)))
plot(M1)
```
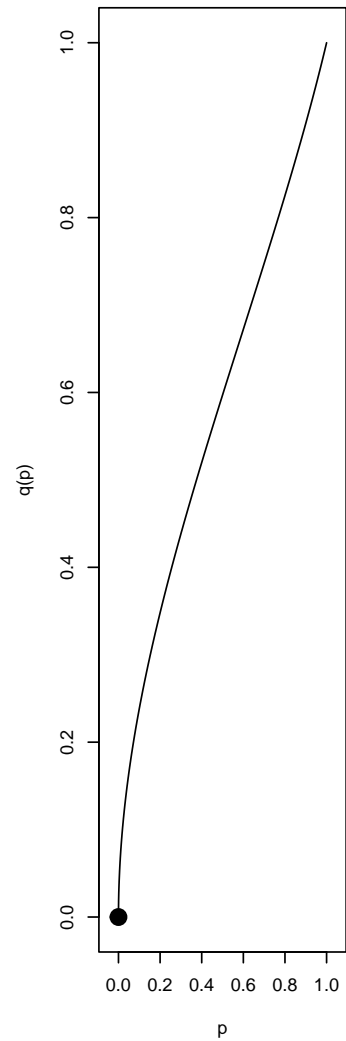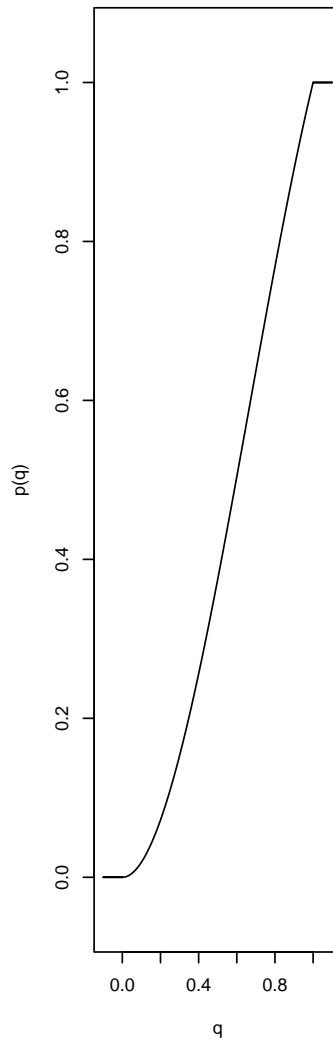
```
M2 <- Minimum(Exp(4),4)
plot(M2)
```
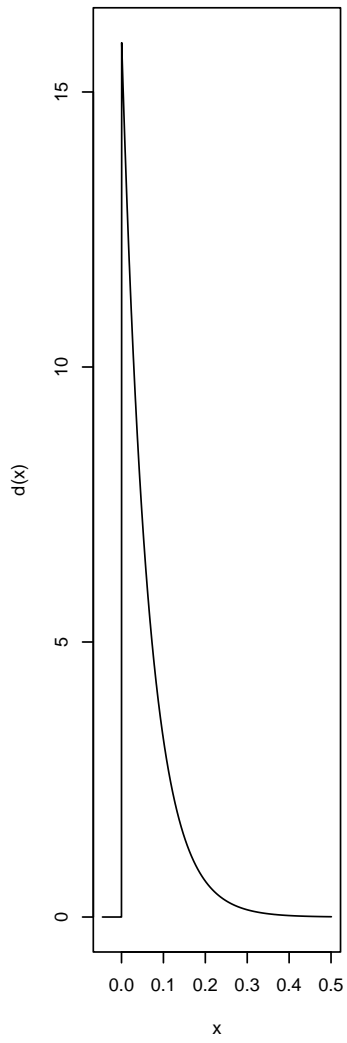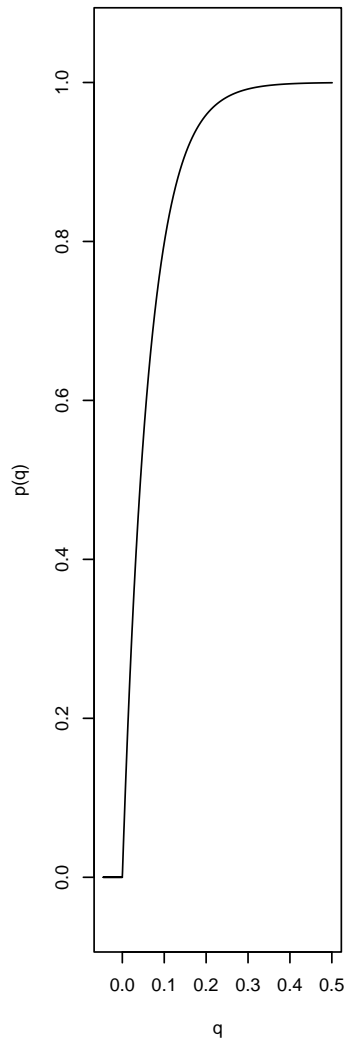
Density of AbscontDistribution    CDF of AbscontDistribution    Quantile function of AbscontDistribut
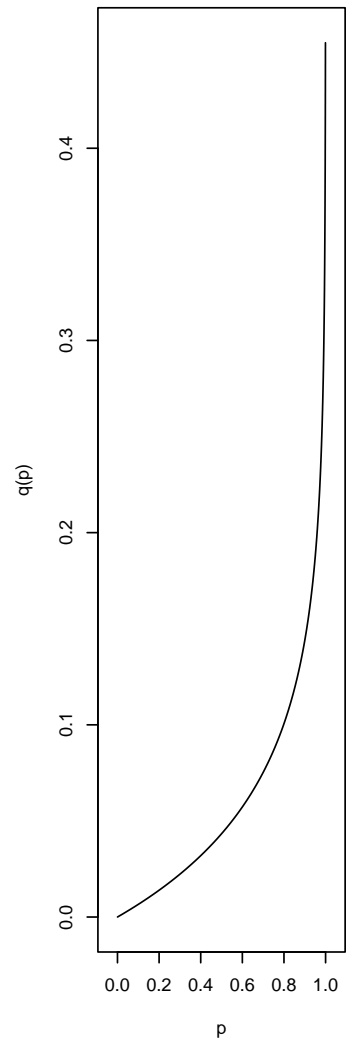
```
M3 <- Minimum(Norm(2,2), Pois(3))
plot(M3)
```
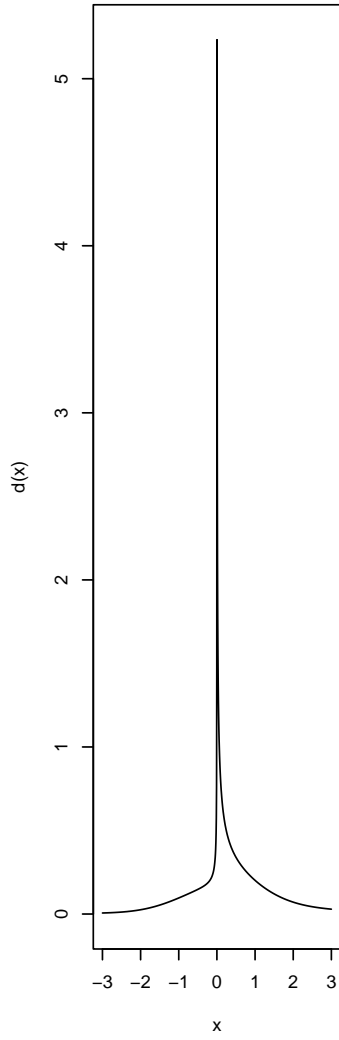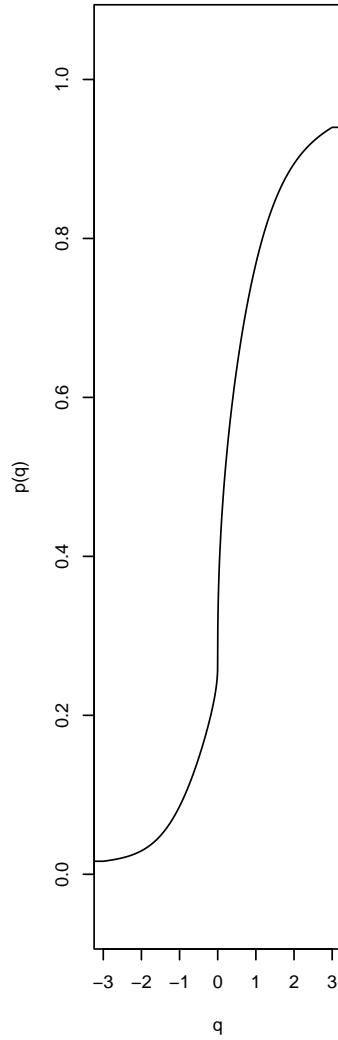
CDF of UnivarLebDecDistribution

Quantile function of UnivarLebDecDistribution

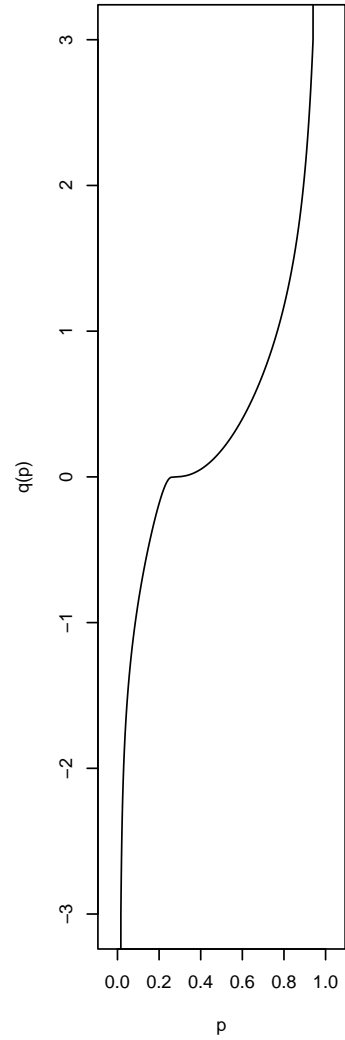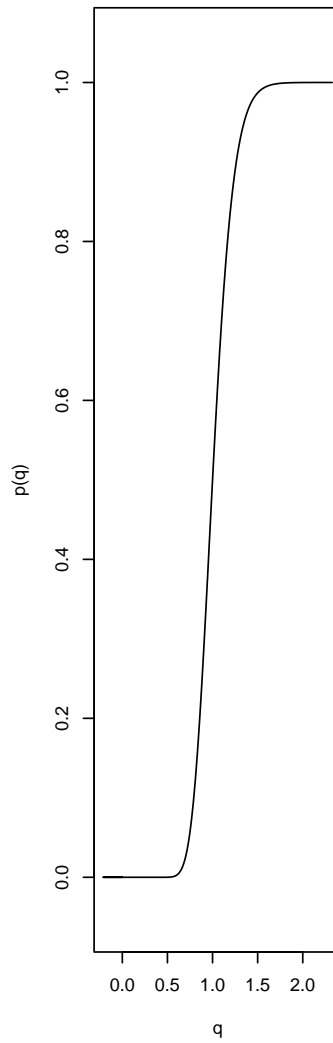Density of AbscontDistribution

CDF of AbscontDistribution

Quantile function of AbscontDistribution

Probability function of DiscreteDistribution

CDF of DiscreteDistribution

Quantile function of DiscreteDistribution
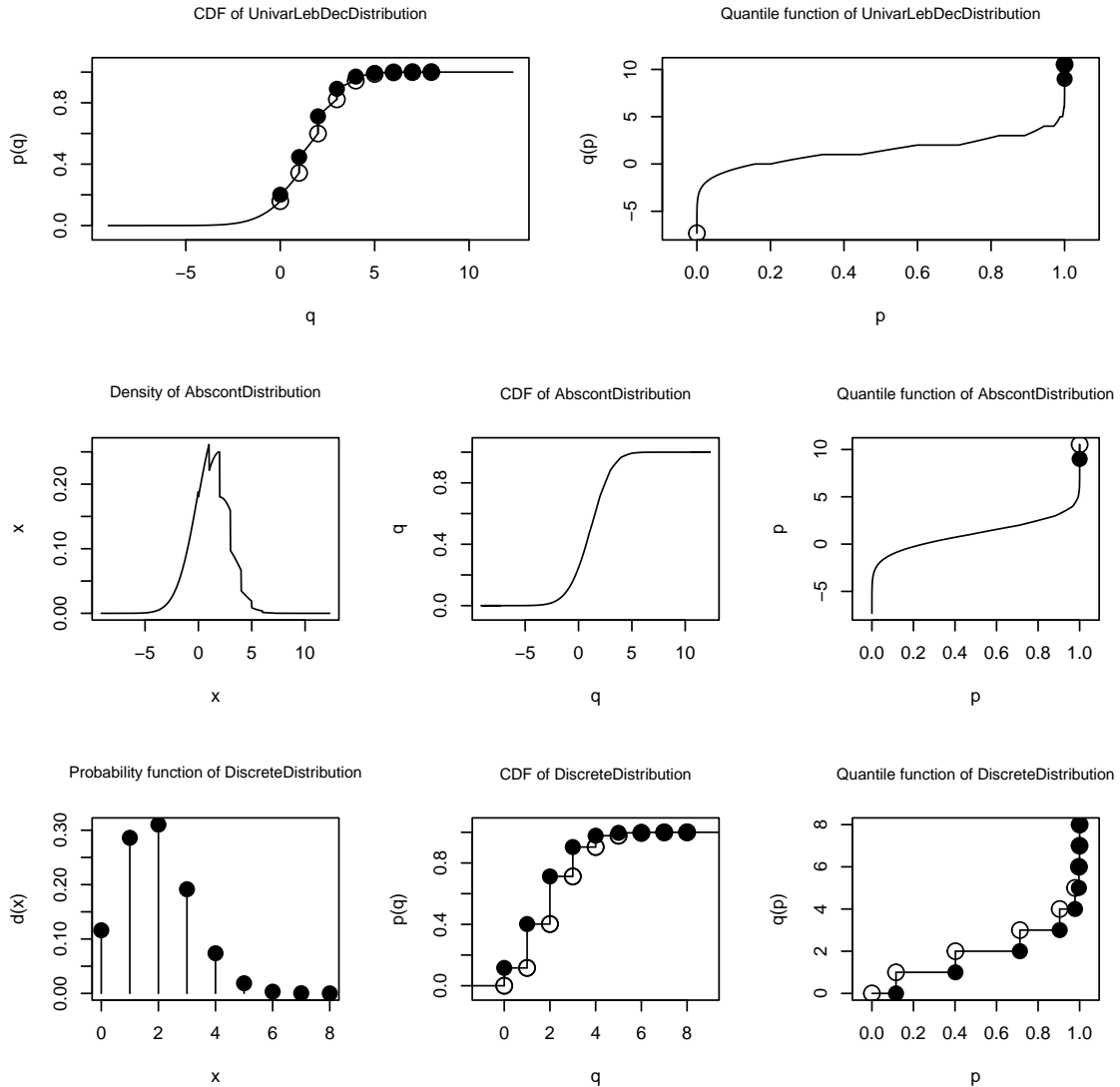
**Enhanced accuracy by using log scales:** Following an idea proposed by Peter Dalgaard on `r-help`, confer https://stat.ethz.ch/pipermail/r-help/2008-September/174295.html we use the **log**.p argument to be able to simulate from the far out tails. To this end, from version 2.1, our distribution classes gain internal slots .lowerExact and .logExact which control whether the corresponding slot functions p and q use particular code for computing on the log-scale respectively whether they use explicit representations for upper quantiles (instead of computing $1 - \text{lowerQuantile}$). If a corresponging distribution object has slots

.lowerExact and .logExact both TRUE, we may use this to produce accurate simulations, especially for respectively truncated distributions:

```
N <- Norm()
TN <- Truncate(N, 20,22)
r(TN)(20)  ## simulation accurate although :

##  [1] 20.05187 20.11175 20.21531 20.10694 20.07931 20.06208 20.01516 20.06959
##  [9] 20.07691 20.04166 20.00698 20.00950 20.00386 20.05449 20.06220 20.03736
## [17] 20.00870 20.03985 20.00361 20.09076

p(N)(20, lower.tail = FALSE) ## prob that N>=20

## [1] 2.753624e-89
```

## 3.9   Additional helper functions

From version 1.9 on, there are methods p.l and q.r available for DiscreteDistribution objects for the left-continuous variant of the cdf, i.e.; $t \mapsto \mathrm{p.l}(t) = P(X < t))$, and the right-continuous variant of the quantile function, i.e.;

$$s \mapsto \mathrm{q.r}(s) = \sup\{t \,|\, P(\texttt{object} \le t) \le s\}$$

From version 2.1 on, p.l and q.r, are also available for class AbscontDistribution (where q.r takes care about correct treatment of gaps using helper function modifyqgaps), for class UnivarLebDecDistribution, and for class UnivarMixingDistribution.

```
B <- Binom(5,0.5)
p(B)(3)

## [1] 0.8125

p.l(B)(3)

## [1] 0.5

q(B)(.5)

## [1] 2

q.r(B)(0.5)

## [1] 3
```

Again from version 2.1 on, class DiscreteDistribution has a helper method prob which returns vector of probabilities for the support points. More precisely, the return value is a numeric vector named by the values of support points. This method is also available for objects of class UnivarLebDecDistribution where it returns a two-row matrix where the column names are the values of the support points, and the first row, named "cond", contains the probabilities of the discrete part (summing up to 1), while the second row, named "abd" contains the probabilities of discrete part multiplied with discreteWeight; hence these values are the absolute probabilities of the support points.

Again for objects of class UnivarLebDecDistribution, we have methods p.ac, d.ac, p.discrete, d.discrete to give the density/probability and the cumulative distribution function of the discrete and absolutely continuous (ac) part of the distribution. All these methods have an extra argument CondOrAbs with default value "cond", which if it does not partially match "abs", returns exactly slot p (resp. d) the respective a.c./ discrete part of the object; else the return value is weighted by the respective weight of the part, i.e. acWeight/discreteWeight.

```
B0 <- as(Binom(5,0.5),"DiscreteDistribution")
    ## coercion necessary:
    ## otherwise slot "prob" of B0 will be returned
prob(B0)

##       0       1       2       3       4       5
## 0.03125 0.15625 0.31250 0.31250 0.15625 0.03125

HN <- Huberize(N, -2,1)
prob(HN)

##               -2          1
## cond 0.12541045 0.8745895
## abs  0.02275013 0.1586553
```

In order to convert arbitrary univariate distributions to AbscontDistribution from version 2.1 on, we have function makeAbscontDistribution which takes slot p and uses AbscontDistribution() to generate a corresponding smoothed version; to smear out mass points on the border of the support, these upper and lower bounds are somewhat enlarged. Note that in the result, slots p and q are not replaced but rather taken unchanged from the argument:

```
par(mfrow=c(2,3))
plot(makeAbscontDistribution(Nbinom(5,.5)),mfColRow=FALSE)

## Warning in .makeDNew(xx, dx, h = NULL, Cont = TRUE, standM = "integrate"):
'integrate()' threw an error --result may be inaccurate.
```

44

```
plot(makeAbscontDistribution(HN),mfColRow=FALSE)
```

```
## Warning in .makeDNew(xx, dx, h = NULL, Cont = TRUE, standM = "integrate"):
'integrate()' threw an error --result may be inaccurate.
```
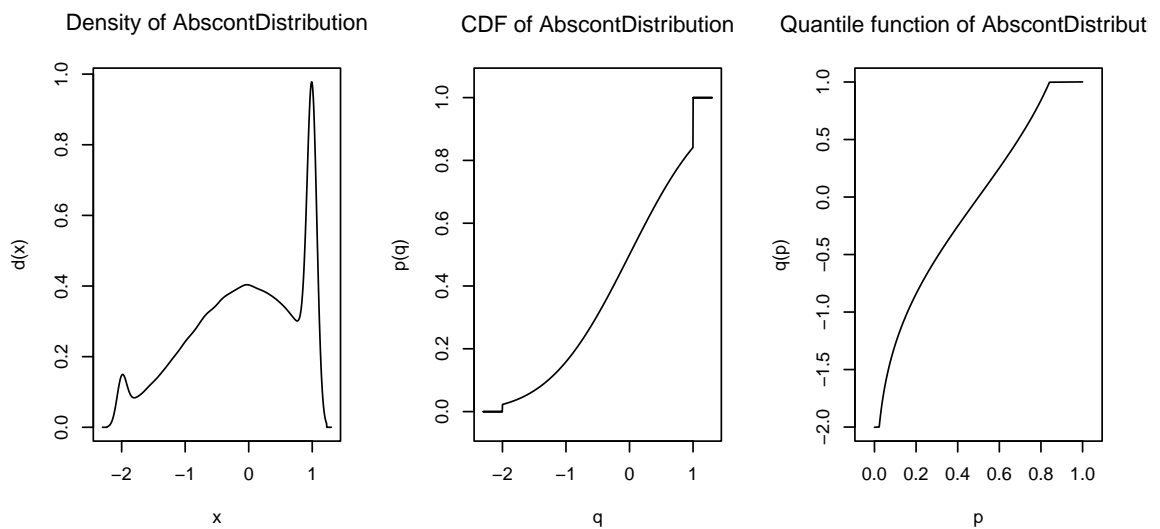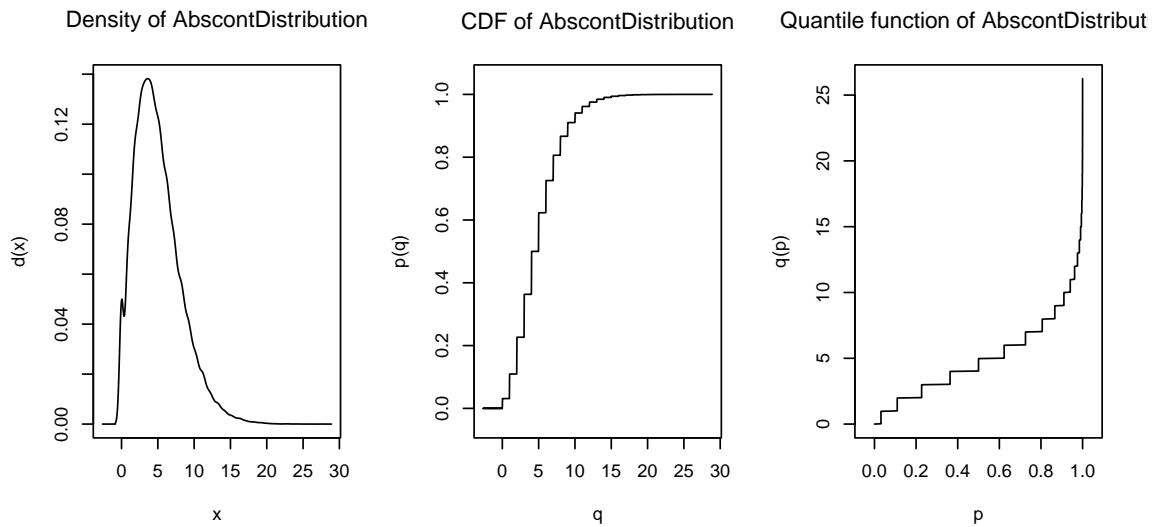


```
par(mfrow=c(1,1))
```

Methods getLow, getUp available for classes DiscreteDistribution, AbscontDistribution, UnivarLebDecDistribution, and UnivarMixingDistribution return "numerical" end points of the respective supports: if these

45

distributions have finite end points, these are returned, else a lower/upper eps-quantile is returned, where ep, by default, is set to global options TruncQuantile.

```
getLow(Nbinom(5,0.5))

## [1] 0

getUp(Nbinom(5,0.5))

## [1] 69

getLow(Norm(5,0.5))

## [1] 2.867555

getUp(Norm(5,0.5))

## [1] 7.132445
```

## 3.10 Overloaded generic functions

Methods **print**, **plot**, **show** and **summary** have been overloaded for classes Distribution, Dataclass, Simulation, ContSimulation, as well as Evaluation and EvaluationList to produce "pretty" output. More specifically there are also particular **show** methods for classes UnivarDistrList, UnivarMixingDistribution and UnivarLebDecDistribution. **print**, **plot**, **show** and **summary** have additional, optional arguments for plotting subsets of the simulations / results: index vectors for the dimensions, the runs, the observations, and the evaluations may be passed using arguments obs0, runs0, dims0, eval0, confer
**help**("<mthd>−methods",**package**=<pkg>) where <mthd> stands for **plot**, **show**, **print**, or **plot**, and <pkg> stands for either "distrSim" or "distrTEst".

## 3.11 Plotting

### 3.11.1 Plotting for Distribution objects

For an object of class Distribution, **plot** displays the density/probability function, the c.d.f. and the quantile function of a distribution. Note that all usual parameters of **plot** remain valid. For instance, you may increase the axis annotations and so on. **plot**() can also cope with **log**-arguments.

**xlim argument:** More importantly, you may also override the automatically chosen $x$-region by passing an xlim argument:

```
plot(Cauchy())
```

Density of Cauchy(0, 1)          CDF of Cauchy(0, 1)          Quantile function of Cauchy(0, 1)



```
plot(Cauchy(),xlim=c(-4,4))
```

Density of Cauchy(0, 1)     CDF of Cauchy(0, 1)     Quantile function of Cauchy(0, 1)
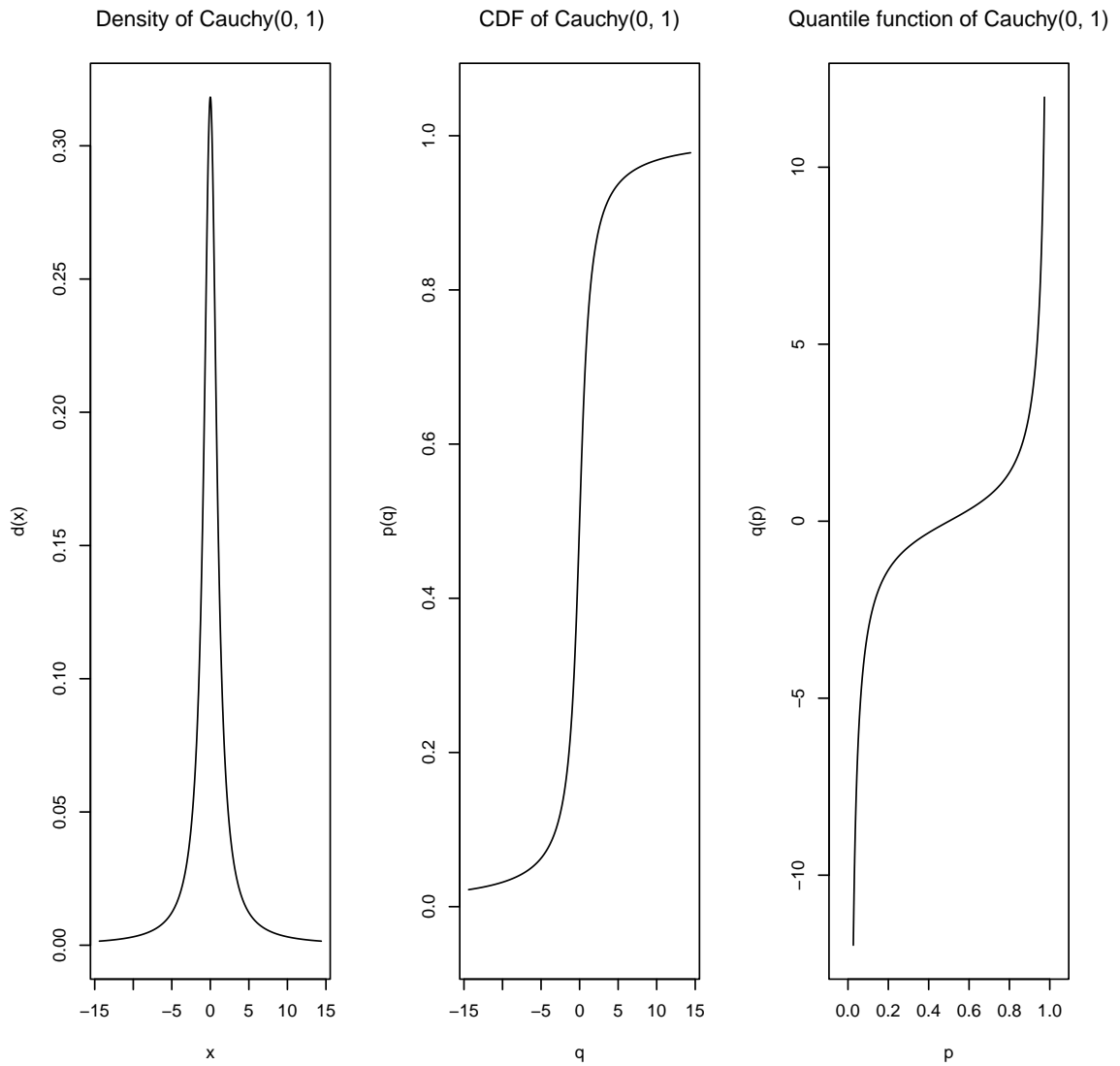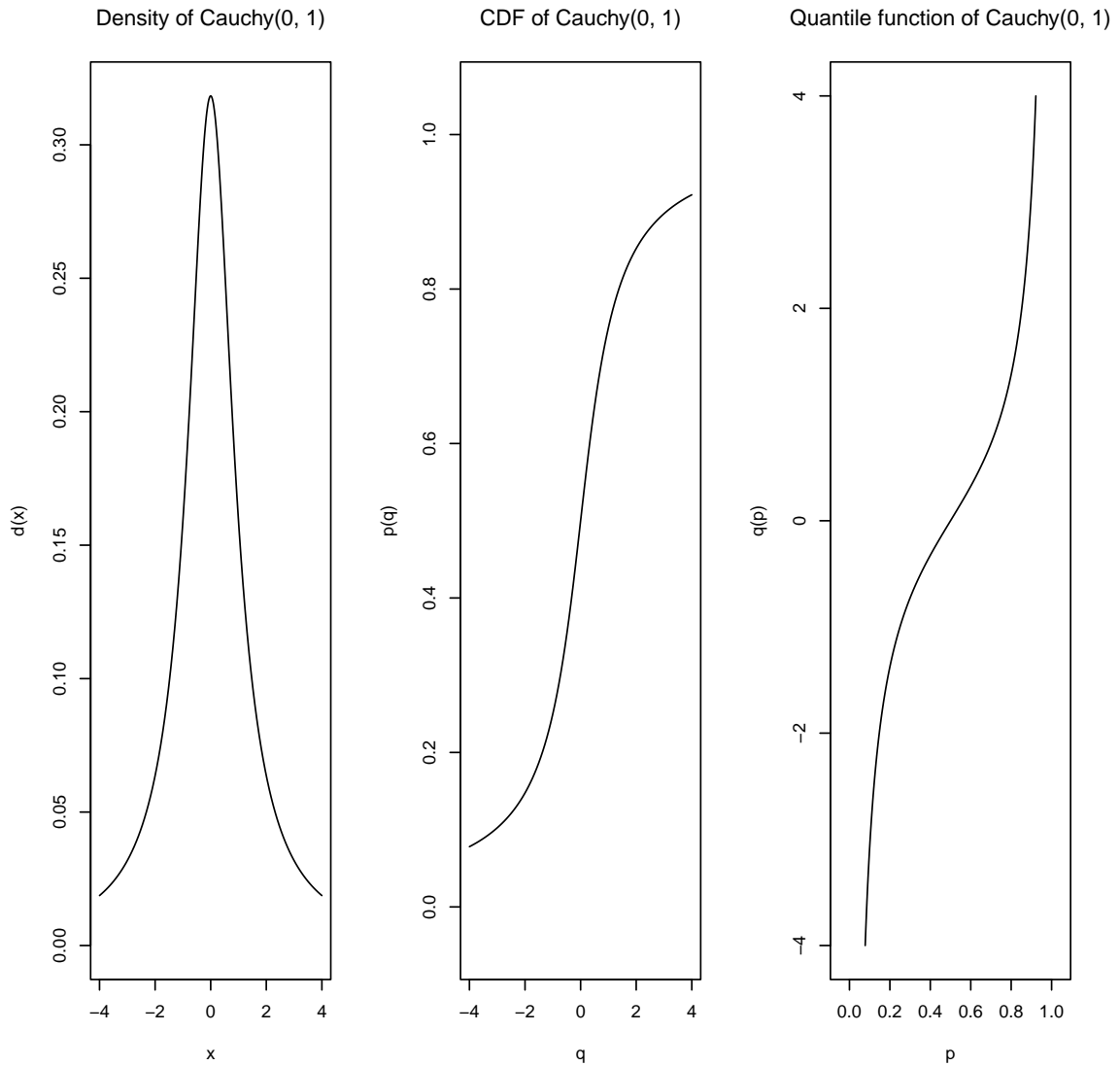
From version 2.1, the automatic choice of the x-range of **plot** for distributions has been enhanced: just as for expectation (see subsection 3.17.1) we use both a quantile and a scale based approach to get sensible values. Also argument ylim can now be matrix-valued to use different limits for the various panels (d, p, **q**, or in case of UnivarLebDecDistribution, p, **q**, d.ac, p.ac, **q**.ac, d. discrete, p. discrete, **q**. discrete).

**titles:** Moreover you may control optional main, inner titles and subtitles with arguments main / **sub** / inner. To this end there are preset strings substituted in both expression and character vectors (where in the following x denotes the argument with which **plot**() was

called)

%A  deparsed argument x

%C  class of argument x

%P  comma-separated list of parameter values of slot param of argument x

%Q  comma-separated list of parameter values of slot param of argument x in parenthesis unless this list is empty; then ""

%N  comma-separated `<name> = <value>` - list of parameter values of slot param of argument x

%D  time/date at which plot is/was generated

This substitution can be switched off by means of argument withSubst. As usual you may control title sizes and colors with cex.main / cex.inner / **cex.sub** respectively with **col** / **col**.main / **col**.inner / **col.sub**. Additionally it may be helpful to control top and bottom margins with arguments bmar, tmar.

**step-function features:** We provide different default symbols for unattained [pch.u] / attained [pch.a] one-sided limits, which may be overridden by corresponding arguments pch / pch.a / pch.u.

For objects of class AbscontDistribution, you may set the number of grid points used by an ngrid argument; also the "quantile"-panel takes care of finite left/right endpoints of support and optionally tries to identify constancy region of the p-slot.

For objects of class DiscreteDistributions, we use stepfun() from package `"base"` as far as possible and (also for panel "q" for AbscontDistributions) consequently take over its arguments **do.points**, verticals , **col.points** / **col**. vert / **col**.hor and cex.**points**.

As examples consider the following plots: The standard plot for a discrete distribution is shown in Figure 7.

Omitting the point symbols at jump points (**do.points** = FALSE) and the corresponding vertical lines ( verticals = FALSE) in the p and **q** panels gives Figure 8.

Instead we might use a somewhat enlarged (cex.main = 1.6) main title (main = TRUE) and omit the panel titles (inner = FALSE). To this end, we should increase the margin between main title and panels (by tmar = 6). This is shown in Figure 9.

Changing point sizes (cex.**points**) and line width (lwd), and using (default) panel titles (set TRUE by default) gives a somewhat different picture as in Figure 10.

Different colors for different plot elements can be used by arguments **col** (general), **col.points** (jump points), **col.sub** (sub-titles), **col**.inner (panel titles). In Figure 11, this is shown; the figure uses main (main = TRUE), inner (TRUE by default) and sub (**sub** = TRUE) titles, using the default titles respectively.

```
plot(Binom(size = 4, prob = 0.3))
```

Probability function of Binom(4, 0.3)     CDF of Binom(4, 0.3)     Quantile function of Binom(4, 0.3)
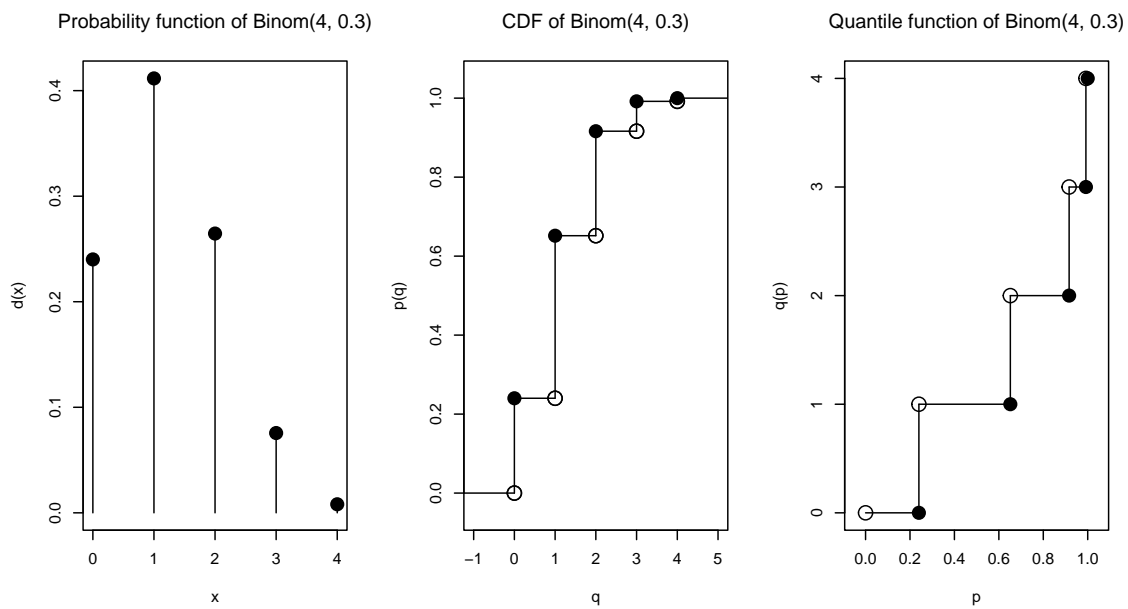
Figure 7: Standard plot for discrete distributions

```
plot(Binom(size = 4, prob = 0.3), do.points = FALSE, verticals = FALSE)
```
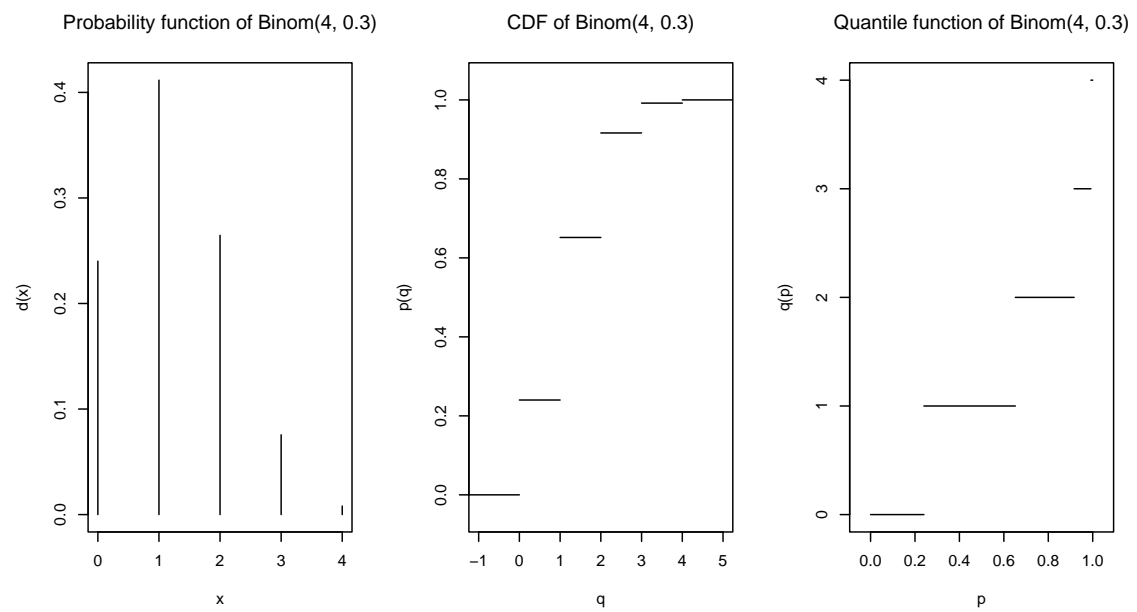


Figure 8: Plot for discrete distributions without extra symbols at jump points and vertical lines

```
plot(Binom(size = 4, prob = 0.3), main = TRUE, inner = FALSE, cex.main = 1.6,
     tmar = 6)
```

## Distribution Plot for Binom(size = 4, prob = 0.3)
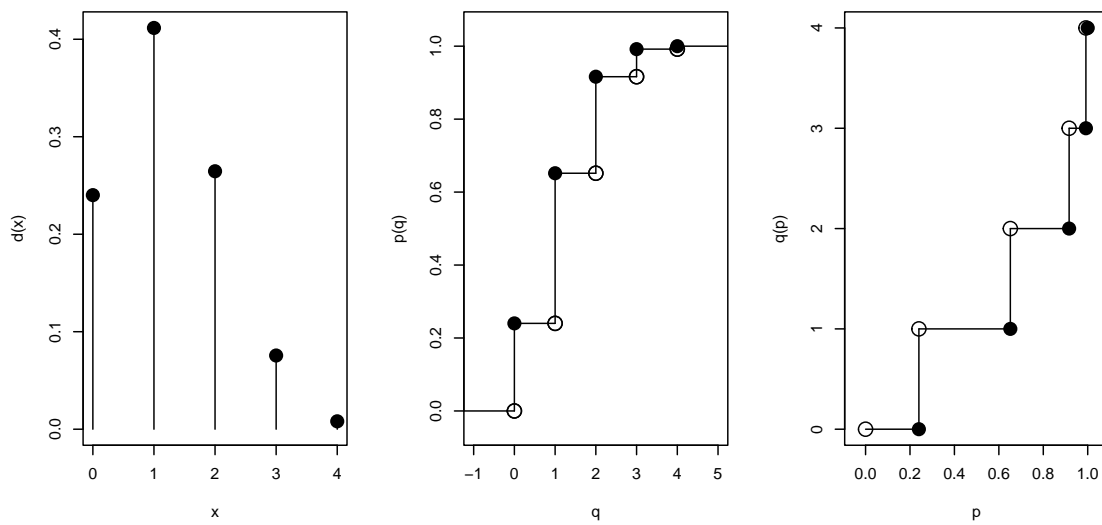


Figure 9: Plot for discrete distributions using a main title

```
plot(Binom(size = 4, prob = 0.3), cex.points = 1.2, pch = 20, lwd = 2)
```
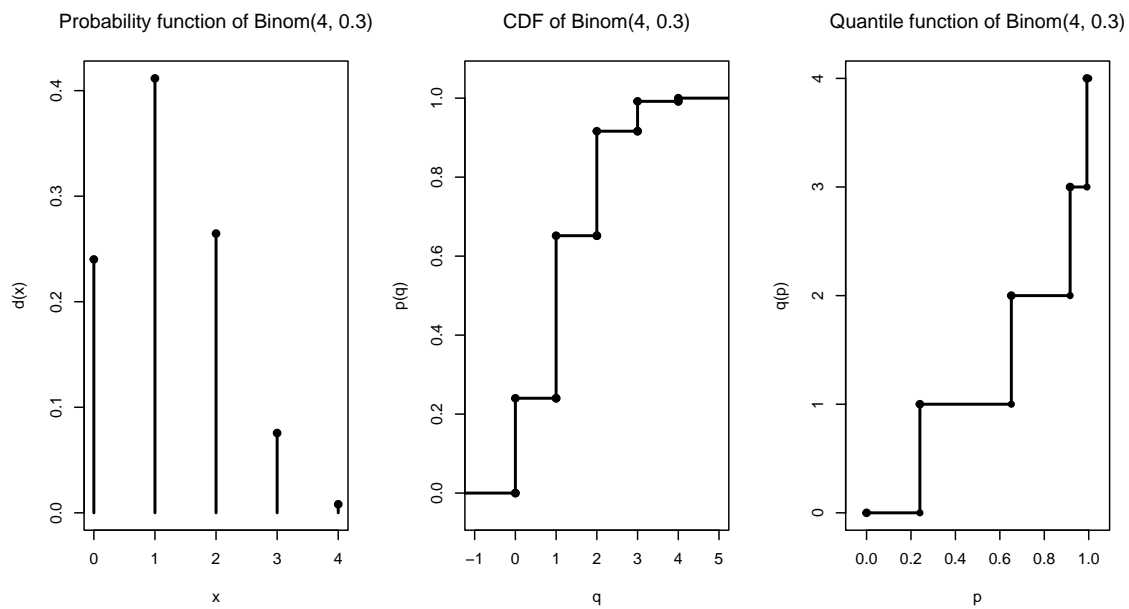


Figure 10: Plot for discrete distributions using panel titles and changed point sizes

```
B <- Binom(size = 4, prob = 0.3)
plot(B, col="red", col.points = "green", main = TRUE, col.main="blue",
     col.sub = "orange", sub = TRUE, cex.sub = 0.6, col.inner = "brown")
```
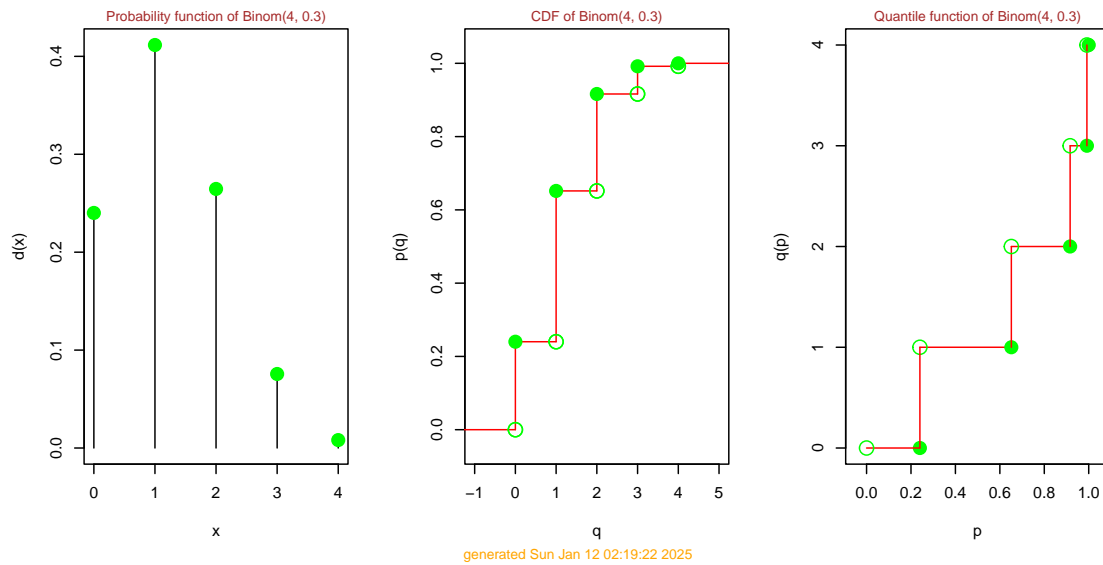


Figure 11: Plot for discrete distributions using main, panel and sub titles with changed colors

Changing plot size for marking the jump points by cex.**points** = 1.2 and selecting symbols for left limits (by pch.u = 20) and right values (by pch.a = 20) at jump points is exemplified in Figure 12.

Using log-scale for both axes by **log** = "xy" and a reduced number of grid points for plotting by ngrid = 200 is shown in Figure 13.

Changing line type by lty = 3 and color by **col** = "red", axis notation orientation by las = 2 and the number of grid points used for plotting by ngrid = 200 is shown in Figure 14.

You may also use hook functions in distribution plots from version 2.0 on, which is especially useful for plotting grids in the background as shown in Figure 15; this plot also has non-default titles using preset strings substitutions (%N, %C, %P, %A, %D).

Special care is take to correctly depict jumps in the quantile function / gaps in the support; from version 2.0 on you have function setgaps() to automatically such gaps, but in case of very small density values this may give some "false positives" as shown for the $\chi^2$ distribution in Figure 16

From version 2.0 on, you may override the given panel configuration using argument mfColRow=FALSE —see Figure 17.

Following a suggestion by Anthony Unwin, from version 2.1 you may also select the panels you like to plot, using argument to.draw.arg; the corresponding panels are named and may either be given by name or by number (the rank in drawing the default "complete" plot); for details see ?**plot**. As example for this panel selection, see Figure 18.

### 3.11.2 Plotting for Dataclass objects

For objects of class Dataclass —or of a corresponding subclass— **plot** plots the sample against the run index and in case of ContSimulation the contaminating variables are highlighted by a different color. Additional arguments controlling the plot as in the default **plot** command may be passed, confer **help**("plot−methods",**package**="distrSim").

### 3.11.3 Plotting for Evaluation objects

For an object of class Evaluation, **plot** yields a boxplot of the results of the evaluation. For an object of class EvaluationList, **plot** regroups the list according to the different columns/coordinates of the result of the evaluation; for each such coordinate, a boxplot is generated, containing possibly several procedures, and, if evaluated at a Contsimulation, the plots are also grouped into evaluations on ideal and real data. As for the usual **boxplot** function you may pass additional "**plot**-type" arguments to this particular **plot** method, confer **help**("plot−methods",**package**="distrTEst"). In particular, the **plot**-arguments main and ylim, however, may also be transmitted coordinatewise, i.e.; a vector of the same length as the dimension of the result `resDim` (e.g. parameter dimension), respectively a `2 x resDim` matrix, or they may be transmitted globally, using the usual `S` recycling rules.

55

```
plot(Nbinom(size = 4,prob = 0.3), cex.points = 1.2, pch.u = 20, pch.a = 10)
```
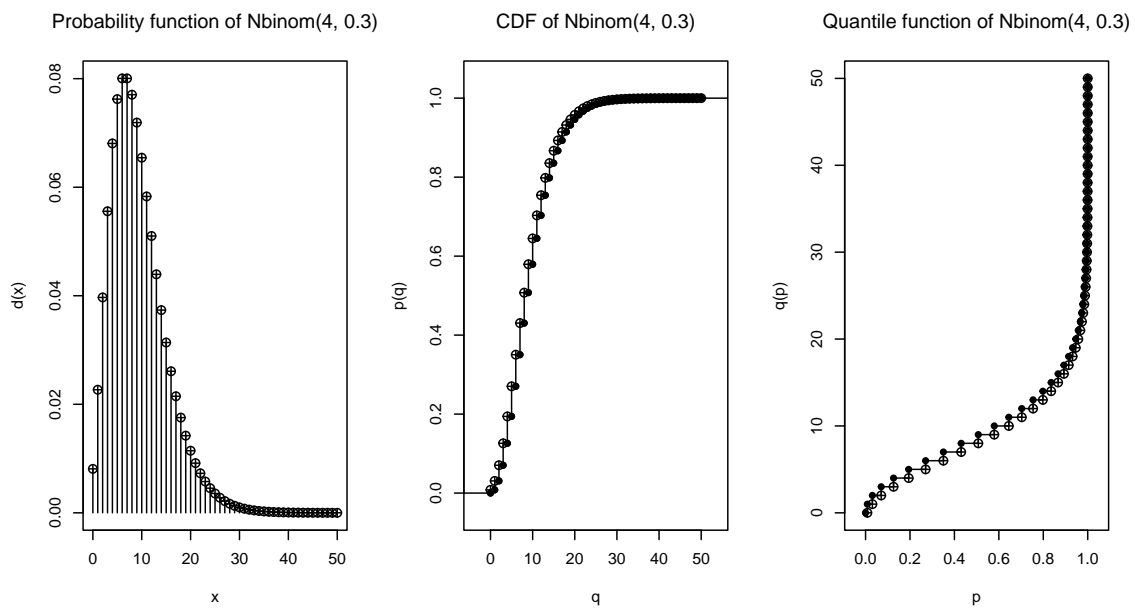


Figure 12: Plot for discrete distributions with different symbols for marking jump points

```
plot(Chisq(), log = "xy", ngrid = 100)
```
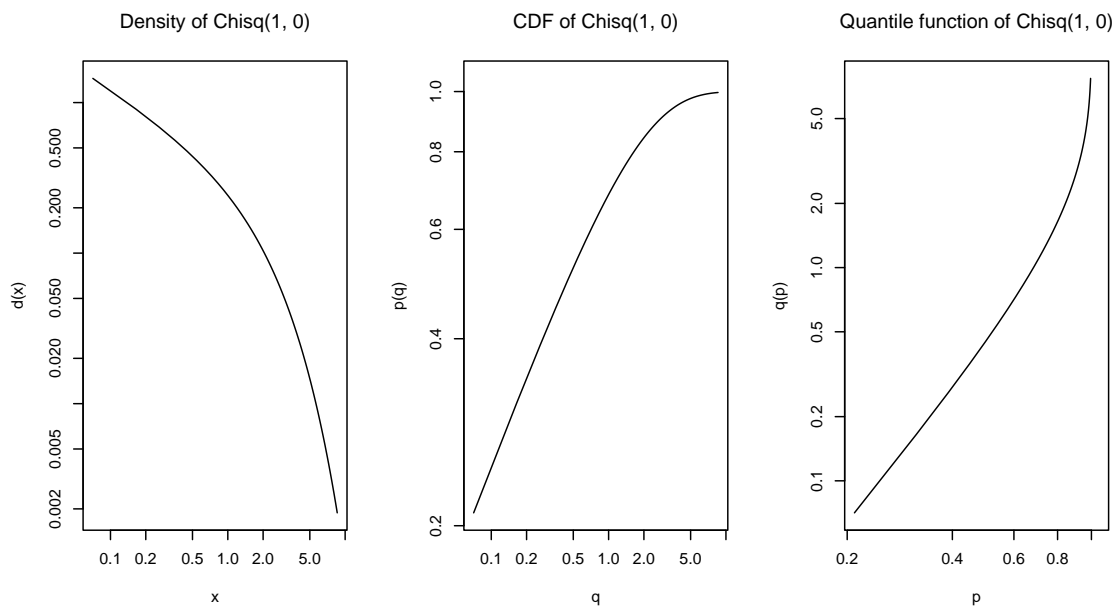


Figure 13: Plot for absolutely continuous distributions using log scales

```
plot(Norm(), lwd=3, col = "red", ngrid = 200, lty = 3, las = 2)
```
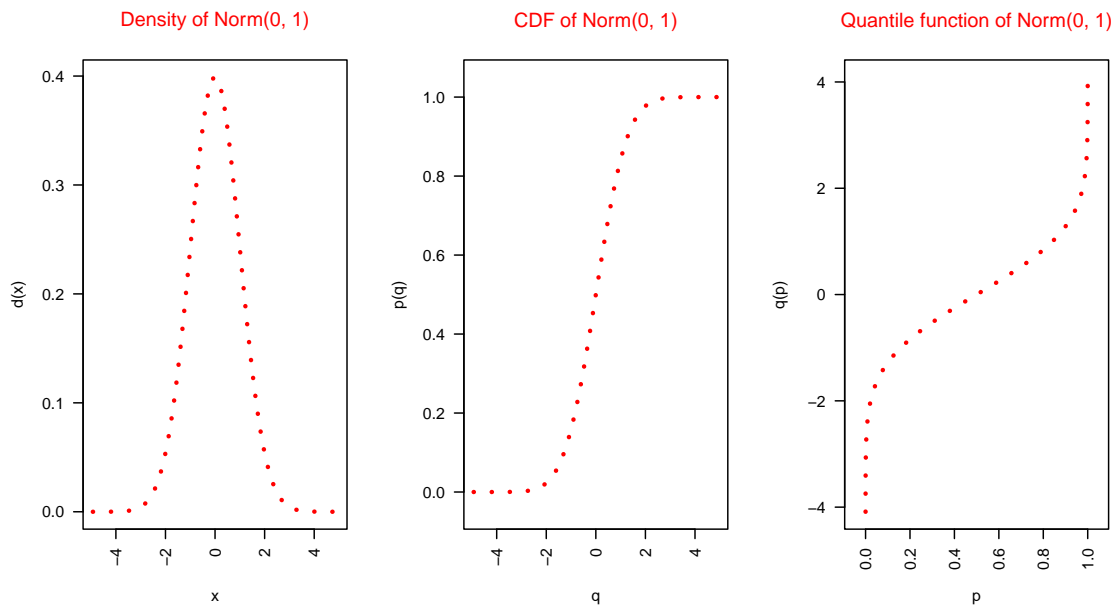


Figure 14: Plot for absolutely continuous distributions with different line type and reduced number of grid points

```
plot(Norm(), panel.first = grid(), main = "my Distribution: %A",
     inner = list(expression(paste(lambda, "-density of %C(%P)")), "CDF",
                  "Pseudo-inverse with param's %N"),
     sub = "this plot was correctly generated on %D",
     cex.inner = 0.9, cex.sub = 0.8)
```

my Distribution: Norm()
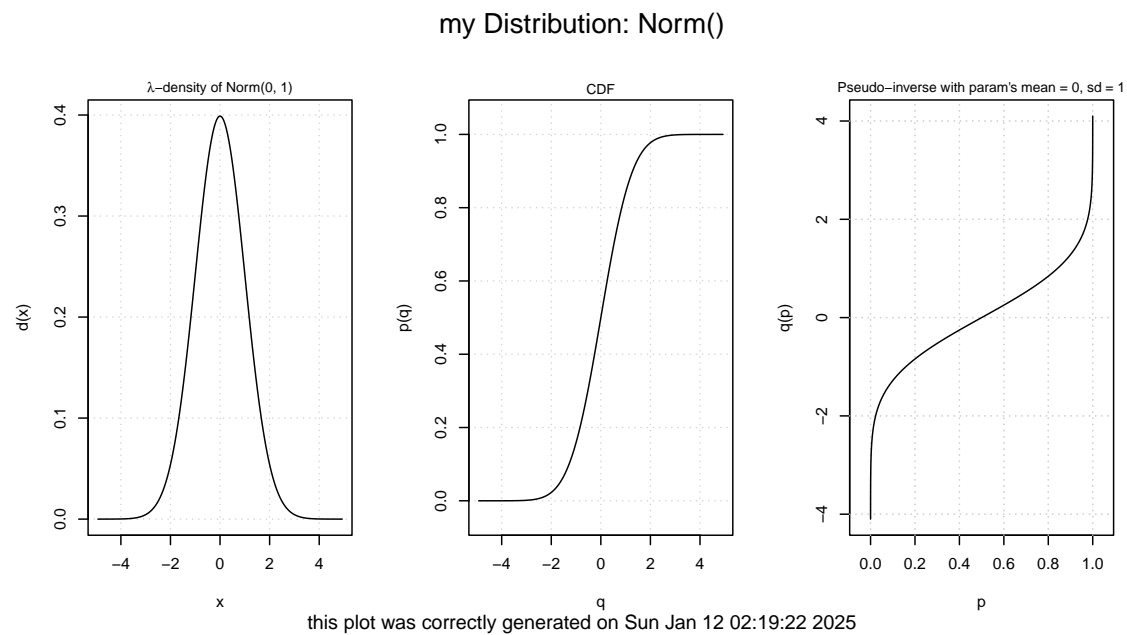


Figure 15: Plot for absolutely continuous distributions using non-standard titles and with a background grid

```
Ch <- Chisq(); setgaps(Ch, exactq = 3)
plot(Ch, cex = 1.2, pch.u = 20, pch.a = 10, col.points = "green",
     col.vert = "red")
```
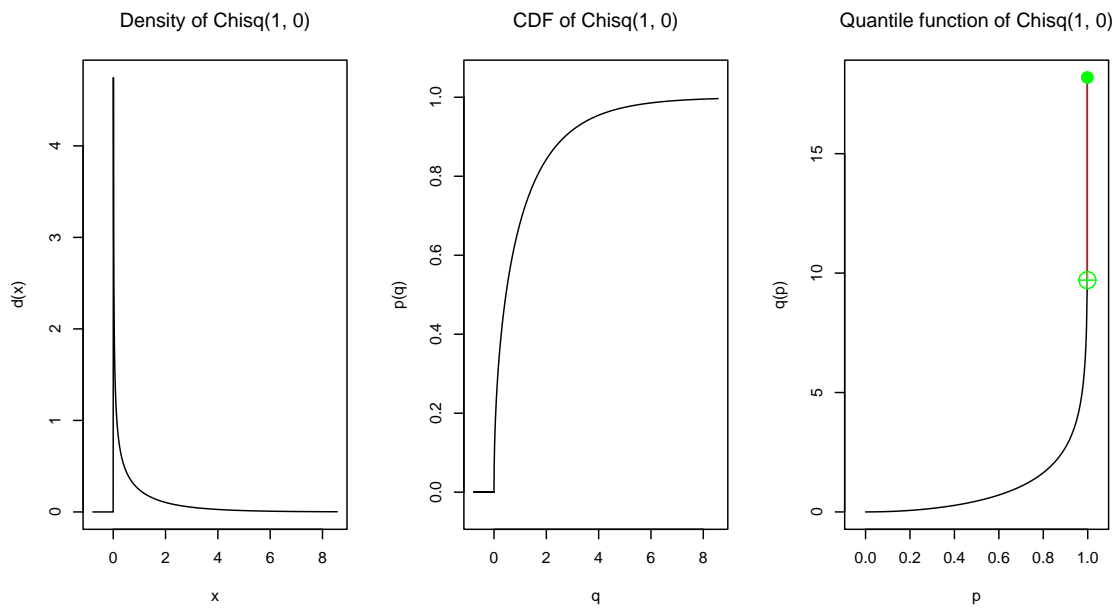


Figure 16: Plot for absolutely continuous distributions with automatic gap detection

```
layout(matrix(c(1,3,2,3), nrow=2))
plot(N, mfColRow = FALSE)
```
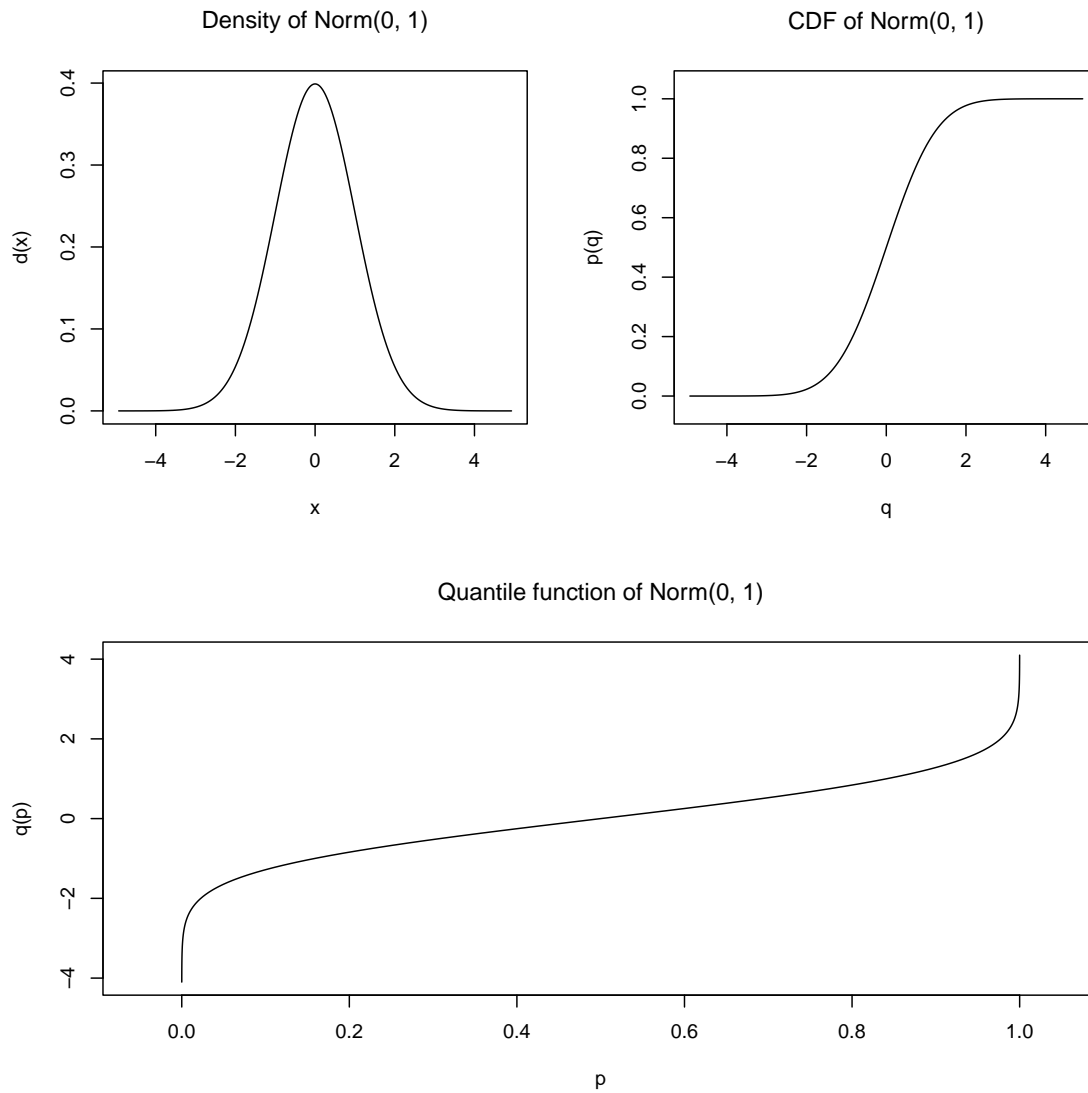


Figure 17: Plot for absolutely continuous distributions using non-standard panel configuration

```
layout(matrix(c(rep(1,6),2,2,3,3,4,4,5,5,5,6,6,6),
               nrow=3, byrow=TRUE))
plot(HN, mfColRow = FALSE,
     to.draw.arg=c("p","d.c","p.c","q.c", "p.d","q.d"))
```
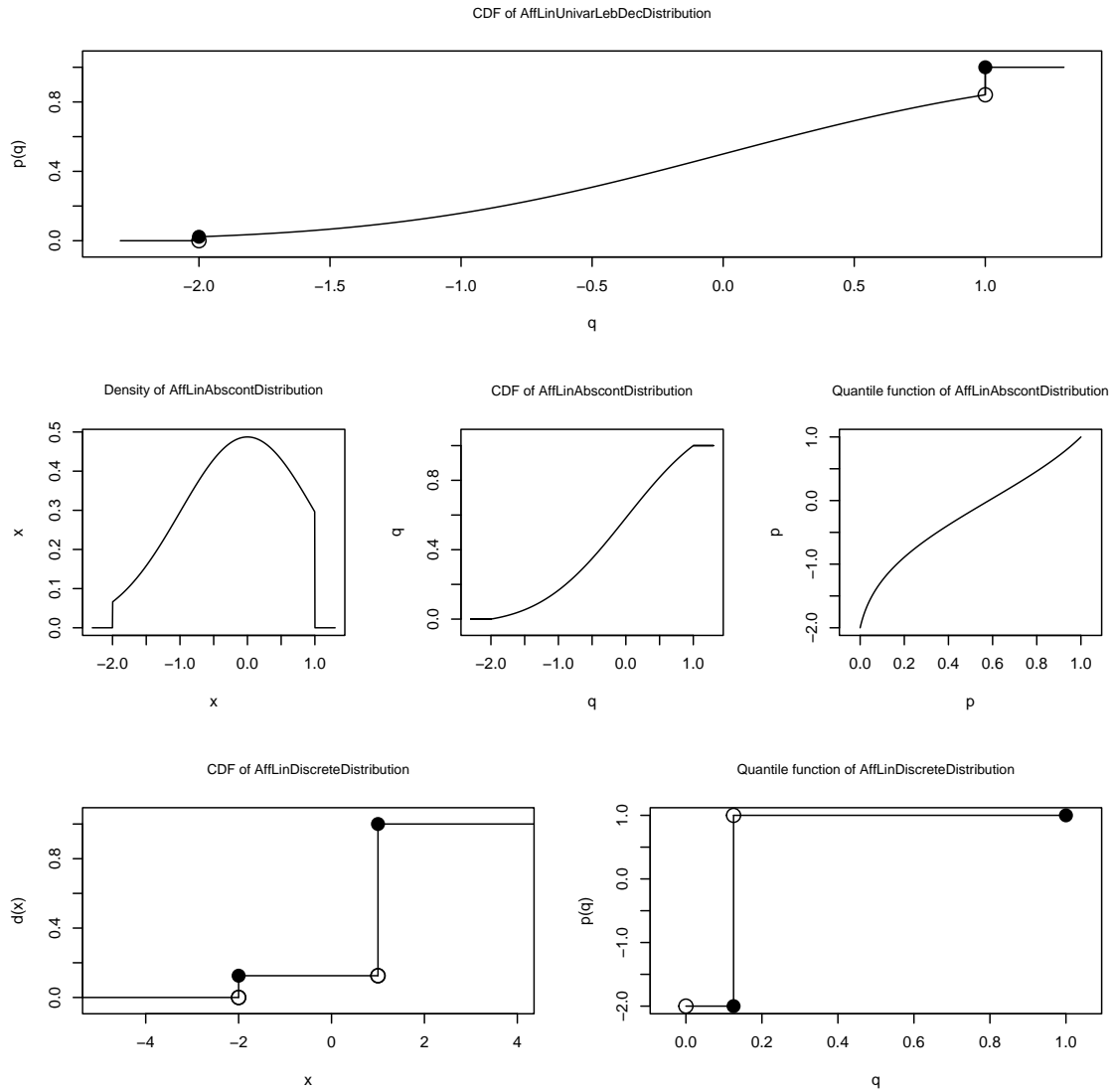


Figure 18: Plot for Lebesgue decomposed distributions with user-chosen selection of plotted panels

### 3.11.4 Plotting for `L2paramFamily` objects

In package `"distrMod"` we have an additional plotting method for class `L2paramFamily`; besides the underlying model distribution this also plots the coordinates of the $L_2$-derivative (scores function). From version 2.1 on, this plot is as flexible as the one for Distribution. You can select the panels to be plotted by argument to.draw.arg, and may use (almost) all arguments generally available for **plot**. In particular, also you may use an argument **panel**.first=**grid**() to produce a grid behind the panel.

### 3.12 liesInSupport

For all discrete distribution classes, we have methods liesInSupport to check whether a given vector/ a matrix of points lies in the support of the distribution. From version 2.8.0 on, we have better control on situations where the true support of the distribution is large/infinite to one or both sides—before it used support, which in this case is truncated to the relevant support points.

### 3.13 Simulation (in package `"distrSim"`)

From version 1.6 on, simulation is available in package `"distrSim"`.

For the classes Simulation and ContSimulation, we normally will not save the current values of the simulation, as they can easily be reproduced knowing the values of the other slots of this class. So when declaring a new object of either of the two classes, the slot Data will be empty (NULL). To fill it with the simulated values, we have to apply the method simulate to the object. This has to be redone whenever another slot of the object is changed. To guarantee reproducibility, we use the slot seed.
This slot is controlled and set through Paul Gilbert's `"setRNG"` package. By default, seed is set to setRNG(), which returns the current "state" of the random number generator (RNG). So the user does not need to specify a value for seed, and nevertheless may reproduce his samples: He simply uses simulate to fill the Data slot. If the user wants to, he may also set the seed explicitly via the replacement function seed(), but has to take care of the correct format himself, confer the documentation of setRNG. One easy way to fill the Data slot of a simulation X with "new" random numbers is

```
X <- Simulation()
seed(X) <- setRNG()
simulate(X)
Data(X)[1:10]

##  [1]  1.6349395 -1.5493719 -1.2960348  1.7561997  1.7621819  2.0428429
##  [7]  0.7235745 -1.2181955 -0.3295757  0.4025521
```

63

## 3.14 Evaluate (in package "distrTEst")

From version 1.6 on evaluate is available in "distrTEst".

In an object of class Evaluation we store relevant information about an evaluation of a statistical procedure (estimator/test/predictor) on an object of class Dataclass, including the concrete results of this evaluation. An object of class Evaluation is generated by an application of method evaluate which takes as arguments an object of class Dataclass and a procedure of type **function**. As an example, confer Example 13.8. For data of class Contsimulation, the result takes a slightly different, combining evalations on ideal and real data.

## 3.15 Is-Relations

By means of setIs, we have "told" R that a distribution object obj of class

- "Unif" with Min $\doteq$ 0 and Max $\doteq$ 1 also is a Beta distribution with parameters shape1 = 1, shape2 = 1

- "Geom" also is a negative Binomial distribution with parameters size = 1, prob = prob(obj)

- "Cauchy" with location $\doteq$ 0 and scale $\doteq$ 1 also is a T distribution with parameters **df** = 1, ncp = 0

- "Exp" also is a Gamma distribution with parameters shape = 1, **scale** = 1/rate(obj) and a Weibull distribution with parameters shape = 1, **scale** = 1/rate(obj)

- "Chisq" with non-centrality ncp $\doteq$ 0 also is a Gamma distribution with parameters shape = **df**(obj)/2, **scale** = 2

- "DiscreteDistribution" (from version 1.9 on) with an equally spaced support also is a "LatticeDistribution"

## 3.16 Further methods

When iterating/chaining mathematical operations on a univariate distribution, generation process of random variables can become clumsy and slow. To cope with this, we introduce a sort of "Forget-my-past"-method simplifyr that replaces the chain of mathematical operations in the r-method by drawing with replacement from a large sample ($10^{\texttt{RtoDPQ.e}}$) of these.

## 3.17 Functionals (in package "distrEx")

### 3.17.1 Expectation

The most important contribution of package "distrEx" is a general expectation operator. In basic statistic courses, the expectation E may come as $E[X]$, $E[f(X)]$, $E[X|Y = y]$, or

$\mathrm{E}\left[f(X)|Y=y\right]$. Our operator (or in S4-language "generic function") `E` covers all of these situations (or *signatures*).

**default call**   The most frequent call will be `E(X)` where `X` is an (almost) arbitrary distribution object. More precisely, if `X` is of a specific distribution class like `Pois`, it is evaluated exactly using analytic terms. Else if it is of class `DiscreteDistribution` we use a sum over the support of `X`, and if it is of class `AbscontDistribution` we use numerical integration[8]; for `X` of class `UnivarLebDecDistribution`, expectations for discrete and absolutely continuous part are evaluated separately and subsequently combined according to their respective weights. If we only know that `X` is of class `UnivariateDistribution` we use Monte-Carlo integration. This also is the default method in for class `MultivariateDistribution`, while for `DiscreteMVDistribution` we again use sums. For an object `Y` of a subclass of class union `AffLinDistribution`, we determine the expectation as `Y@a *E(Y@X0) + Y@b` and hence use analytic terms for `X0` if available.

**with a function as argument**   we proceed just as without:  if `X` is of class `DiscreteDistribution`, we use a sum over the support of `X`, and if `X` is of class `AbscontDistribution` we use numerical integration; else we use Monte-Carlo integration.

**in addition: with a condition as argument**   we simply use the corresponding `d` respective `r` slots with the additional argument `cond`.

**exact evaluation**   is available for `X` of class `Arcsine`, `Beta` (for noncentrality 0), `Binom`, `Cauchy`, `Chisq`, `Dirac`, `Exp`, `Fd`, `Gammad`, `Geom`, `Hyper`, `Logis`, `Lnorm`, `Nbinom`, `Norm`, `Pois`, `Td`, `Unif`.

**examples**

```
D4 <- LMCondDistribution(theta = 1)
D4  # corresponds to Norm(cond, 1)

## Distribution object of class: AbscontCondDistribution
##  theta: 1
##  intercept: 0
##  scale: 1
## ## cond:
## name: conditioning by an Euclidean space
## Range: Euclidean Space with dimension 1
```

---

[8]i.e., we first try (really(!): **try**) integrate and if this fails we use Gauß-Legendre integration according to [6], see also ?distrExIntegrate

```
N <- Norm(mean = 2)

E(D4, cond = 1)

## [1] 0.9999998

E(D4, cond = 1, useApply = FALSE)

## [1] 0.9999998

E(as(D4, "UnivariateCondDistribution"), cond = 1)

## Warning in 0 + cond %*% 1 + 1 * r(n, ...):  Recycling array of length 1 in
array-vector arithmetic is deprecated.
##   Use c() or as.vector() instead.

## [1] 0.9943829

E(D4, function(x){x^2}, cond = 2)

## [1] 4.999993

E(D4, function(x){x^2}, cond = 2, useApply = FALSE)

## [1] 4.999993

E(N, function(x){x^2})

## [1] 4.999993

E(as(N, "UnivariateDistribution"), function(x){x^2},
    useApply = FALSE) # crude Monte-Carlo

## [1] 4.990048

E(D4, function(x, cond){cond*x^2}, cond = 2,
  withCond = TRUE)

## [1] 9.999987

E(D4, function(x, cond){cond*x^2}, cond = 2,
  withCond = TRUE, useApply = FALSE)

## [1] 9.999987
```

```
E(N, function(x){2*x^2})

## [1] 9.999987

E(as(N, "UnivariateDistribution"), function(x){2*x^2},
  useApply = FALSE) # crude Monte-Carlo

## [1] 9.988575

Y <- 5 * Binom(4, .25) - 3
Y

## Distribution Object of Class: AffLinLatticeDistribution

E(Y)

## [1] 2
```

**Controlling integration range:** From version 2.1 on, `E` gains arguments `low` and `upp` to restrict evaluation of the integrand to a given integration domain; these arguments can also be passed through by other functionals based on expectation, like **var**, **sd**, skewness, and kurtosis.

```
E(Cauchy(), low=3, upp=5)

## [1] 0.03958342

var(Cauchy(), low=3, upp=5)

## [1] 0.5739097
```

**Controlling accuracy:** From version 2.1 on, our expectation methods gain explicit arguments to set accuracy locally; i.e.; the MC-methods have an argument `Nsim` defaulting to global option `MCIterations`, while the methods for class `AbscontDistribution` using numerical integration have an argument `rel.tol` defaulting to global option `ErelativeTolerance`. To obtain a sensible integration range automatically, these methods use both quantile and scale based methods; more precisely you may pass on arguments lowerTruncQuantile and upperTruncQuantile, defaulting to global options ElowerTruncQuantile, and EupperTruncQuantile, respectively, by means of these we determine lower and upper quantiles $l_0$, $u_0$. In addition,

we determine scale based bounds as median $\pm s_f$ IQR where $s_f$ is a scaling factor to be passed on as argument **IQR**.fac which defaults to global option **IQR**.fac.

```
E(N, function(x)x^2)

## [1] 4.999993

E(N, function(x)x^2,  lowerTruncQuantile = 1e-5)

## [1] 4.999933

var(Cauchy(), low =3, upperTruncQuantile = 1e-5,  IQR.fac = 10)

## [1] 4.979597

var(Cauchy(), low =3, upperTruncQuantile = 1e-10, IQR.fac = 20)

## [1] 11.0304
```

### 3.17.2  Variance

The next-common functional is the variance. In order to keep a unified notation we will use the same name as for the empirical variance, i.e., **var**.

**masking "stats"-method var**    To cope with the different argument structure of the empirical variance, i.e. **var**(x, y = NULL, **na.rm** = FALSE, use) and our functional variance, i.e., **var**(x, fun = **function**(t) {t}, cond, withCond = FALSE, useApply = TRUE, ...) we have to mask the original "stats"-method:

```
  var <- function(x , ...)
      {dots <- list(...)
       if(hasArg(y)) y <- dots$"y"
       na.rm <- ifelse(hasArg(na.rm), dots$"na.rm", FALSE)
       if(!hasArg(use))
            use <- ifelse (na.rm, "complete.obs","all.obs")
       else use <- dots$"use"
       if(hasArg(y))
          stats::var(x = x, y = y, na.rm = na.rm, use)
       else
          stats::var(x = x, y = NULL, na.rm = na.rm, use)
          }
```

before registering **var** as generic function. Doing so, if the x (or the first) argument of **var** is not of class UnivariateDistribution, **var** behaves identically to the **"stats"** package

**default method**  if x is of class UnivariateDistribution, **var** just returns the variance of distribution X — or of fun(X) if a function is passed as argument fun, or, if a condition argument cond (for $Y = y$), $\mathrm{Var}\,[X|Y = y]$ respectively $\mathrm{Var}\,[f(X)|Y = y]$ — just as for E. The same goes for corresponding arguments controlling the accuracy of E locally from version 2.1 on (see paragraph "Controlling accuracy"): These may simply passed through in a call to **var**.

For an object Y of a subclass of class union AffLinDistribution, we determine the variance as Y@a\textasciicircum2 *var(Y@X0) and hence use analytic terms for X0 if available.

**exact evaluation**  is provided for specific distributions if no function and no condition argument is given: this is available for X of class Arcsine, Beta (for noncentrality 0), Binom, Cauchy, Chisq, Dirac, Exp, Fd, Gammad, Geom, Hyper, Logis, Lnorm, Nbinom, Norm, Pois, Unif, Td.

### 3.17.3   Further functionals

By the same techniques we provide the following functionals for univariate distributions:

- standard deviation: **sd**

- skewness: skewness (code contributed by G. Jay Kerns, `gkerns@ysu.edu`)

- kurtosis: kurtosis (code contributed by G. Jay Kerns, `gkerns@ysu.edu`)

- median: **median** (not for function arguments)

- median of absolute deviations: **mad** (not for function/condition arguments)

- interquartile range: **IQR** (not for function arguments)

For details, see ?skewness.

## 3.18   Truncated moments (in package **"distrEx"**)

For Robust Statistics, the first two truncated moments are very useful. These are realized as generic functions m1df and m2df: They use the expectation operator for general univariate distributions, but are overloaded for most specific distributions:

- Binom

- Pois

- Norm

- Exp

- Chisq

## 3.19 Distances (in package "distrEx")

For several purposes like Goodness-of-fit tests or minimum-distance estimators, distances between distributions are useful. This applies in particular to Robust Statistics. In package "distrEx", we provide the follwoing distances:

- Kolmogoroff distance (KolmogorovDist)

- total variation distance (TotalVarDist)

- Hellinger distance (HellingerDist)

- Cramér von Mises distance (CvMDist) with an additional argument for the weighting measure $\mu$ (defaulting to second operand $Q$):

$$d_\mu(P,Q)^2 = \int \Big(Q((-\infty;t]) - P((-\infty;t])\Big)^2 \mu(dt)$$

- convex-contamination "distance" (asymmetric!) (ContaminationSize) defined as

$$d(Q,P) := \inf\{r > 0 \,|\, \exists \text{ probability } H : \quad Q = (1-r)P + rH\}$$

- from version 2.1 on: an asymmetric version of total variation distance (AsymTotalVarDist): to given ratio $\rho \geq 0$ of negative to positive part of the deviation we set

$$d_{v;\rho}(Q,P) := \int (dQ - c\,dP)_+$$

where $c \in \mathbb{R}$ is such that

$$\rho \int (dQ - c\,dP)_+ = \int (dQ - c\,dP)_-$$

- minimal total variation distance (OAsymTotalVarDist):

$$d_{v;\text{opt}}(Q,P) := \min_c \int (dQ - c\,dP)_+ + (dQ - c\,dP)_-$$

Methods using numerical integration use a similar technique as for the expectation mentioned in subsection 3.17.1, combining scale and quantile based methods to obtain a sensible integration range automatically.

## 3.20 Functions for demos (in package "distrEx")

To illustrate the possibilities with packages "distr" and "distrEx" we include two major demos to "distrEx", each with extra code to it — one for the CLT and one for the LLN.

From version 2.0 on, we have started a new package "distrTeach", which is to use the capabilities of packages "distr" and "distrEx" for illustrating topics of Stochastics and Statistics as taught in secondary school. So far we have moved the illustrations for the CLT and the LLN just mentioned to it.

### 3.20.1 CLT for arbitrary summand distribution

By means of our convolution algortithm as well as with the operators E and sd an illustration for the CLT is readily written: function illustrateCLT, respectively demo illustCLT. For plotting, we have particular methods for discrete and absolute continuous distributions. The user may specify a given summand distribution, an upper limit for the consecutive sums to be considered and a pause between the corresponding plots in seconds. From version 1.9 on, we also include a TclTk-based version of this demo, where the user may enter the distribution argument (i.e.; the summands' distribution) into a text line and control the sample size by a slider in some widget: illustCLT\_tcl From version 2.0 on, this functionality has moved to package "distrTeach".

### 3.20.2 LLN for arbitrary summand distribution

From version 1.9 on, similarly, we provide an illustration for the LLN: function illustrateLLN, respectively demo illustLLN. The user may specify a vector of sample sizes to be considered, the number of replicates to be drawn and a pause between the corresponding plots in seconds, also, optionally, the limiting expectation (in case of class Cauchy: the non-limiting median) is drawn as a line and Chebyshev/CLT-based (pointwise) confidence bands and their respective empirical coverages are displayed. From version 2.0 on, this functionality has moved to package "distrTeach".

### 3.20.3 Deconvolution example

To illustrate conditional distributions and their implementation in "distrEx", we consider the following situation: We consider a signal $X \sim P^X$ which is disturbed by noise $\varepsilon \sim P^\varepsilon$, independent from $X$; in fact we observe $Y = X + \varepsilon$ and want to reconstruct $X$ by means of $Y$. By means of the generating function PrognCondDistribution of package "distrEx", for absolutely continuous $P^X, P^\varepsilon$, we may determine the factorized conditional distribution $P^{X|Y=y}$, and based on this either its (posterior) mode oder (posterior) expectation; also see **demo**(Prognose, **package**="distrEx").

# 4   Package `distrMod`

The package `"distrMod"` aims for an object orientated (S4-styple) implementation of probability models and introduces several new S4-classes for this purpose. Moreover, it includes functions to compute minimum criterion estimators – in particular, minimum distance and maximum likelihood (i.e., minimum negative log-likelihood) estimators.

## 4.1   Symmetry Classes

As symmetry is a property which usually cannot be proven via numerical computations, we introduce the S4-class Symmetry and corresponding subclasses which may serve as slots which indicate that there exists a certain symmetry. So far, we have subclasses for the symmetry of distributions as well as for the symmetry of functions; confer Figure 19.



Figure 19: Inheritance relations and slots of the corresponding (sub-)classes for Symmetry where we do not repeat inherited slots

## 4.2 Model Classes

Based on class Distribution and its subclasses we define classes for families of probability measures. So far, we specialised this to parametric families of probability measures; confer Figure 20. But it would also be possible to derive subclasses for other (e.g., semiparametric) families of probability measures. In case of $L_2$-differentiable (i.e., smoothly parameterized) parametric families we introduce several additional slots, in particular the slot L2deriv which is of class EuclRandVarList. Hence, package "distrMod" depends on package "RandVar" [4]. Note that for general central distribution in the one-dimensional location and scale model,
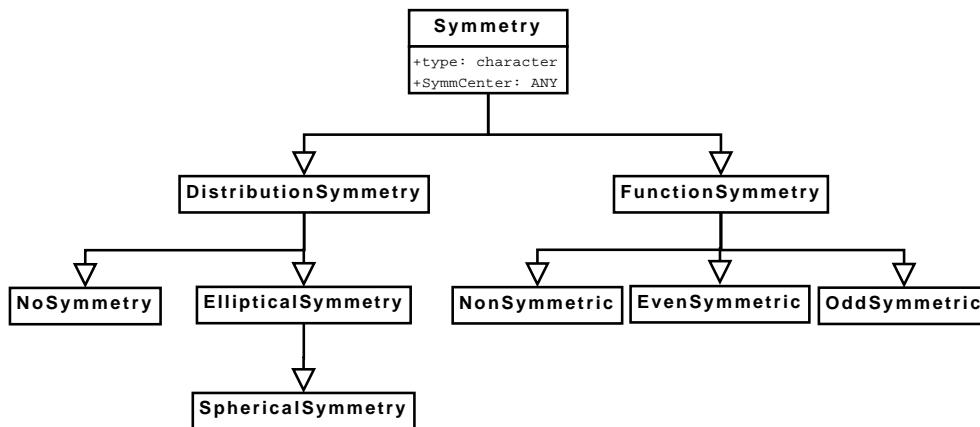
Figure 20: Inheritance relations and slots of the corresponding (sub-)classes for ProbFamily where we do not repeat inherited slots

we need to determine a consistency factor for the MAD (when used as scale estimator). From version 2.1 on this consistency factor is determined automatically.

73

## 4.3  Parameter in a parametric family: class ParamFamParameter

In many applications, it is not the whole parameter of a parametric family which is of interest, but rather parts of it, while the rest of it either is known and fixed or has to be estimated as a nuisance parameter; in other situations, we are interested in a (smooth) transformation of the parameter. This all is realized in a class design for the parameter of a parametric family —class ParamFamParameter, the formal class of a slot of class ParamFamily. It has slots name (the name of the parameter), main (the interesting aspect of the parameter),

```
┌─────────────────────────────┐
│        Parameter            │
├─────────────────────────────┤
│ +name: character            │
└─────────────────────────────┘
```

```
┌─────────────────────────────┐
│    ParamFamParameter        │
├─────────────────────────────┤
│ +main: numeric              │
│ +nuisance: OptionalNumeric  │
│ +fixed: OptionalNumeric     │
│ +trafo: MatrixorFunction    │
└─────────────────────────────┘
```
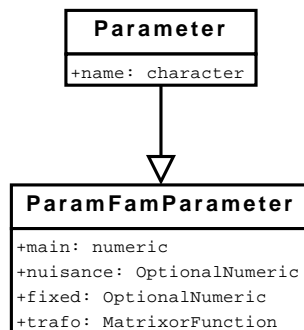
Figure 21:  Inheritance relations and slots of ParamFamParameter where we do not repeat inherited slots

nuisance an unknown part of the parameter of secondary interest, but which has to be estimated, for instance for confidence intervals, and fixed a known and fixed part of the parameter. Besides these it also has a slot trafo which also sort of arises in class Estimate.

trafo realizes partial influence curves; i.e.; we are only interested is some possibly lower dimensional smooth (not necessarily linear or even coordinate-wise) aspect/transformation $\tau$ of the parameter $\theta$.

To be coherent with the corresponding *nuisance* implementation, we make the following convention:

The full parameter $\theta$ is split up coordinate-wise in a main parameter $\theta'$ and a nuisance parameter $\theta''$ (which is unknown, too, hence has to be estimated, but only is of secondary interest) and a fixed, known part $\theta'''$.

Without loss of generality, we restrict ourselves to the case that transformation $\tau$ only acts on the main parameter $\theta'$ — if we want to transform the whole parameter, we only have to assume both nuisance parameter $\theta''$ and fixed known part of the parameter $\theta'''$ have length 0.

**To the implementation:**  Slot trafo can either contain a (constant) matrix $D_\theta$ or a function

$$\tau : \Theta' \to \tilde{\Theta}, \qquad \theta \mapsto \tau(\theta)$$

mapping main parameter $\theta'$ to some range $\tilde{\Theta}$.

If *slot value* trafo is a function, besides $\tau(\theta)$, it will also return the corresponding derivative matrix $\frac{\partial}{\partial\theta}\tau(\theta)$. More specifically, the return value of this function theta is a list with entries fval, the function value $\tau(\theta)$, and **mat**, the derivative matrix.

In case trafo is a matrix $D$, we interpret it as such a derivative matrix $\frac{\partial}{\partial\theta}\tau(\theta)$, and, correspondingly, $\tau(\theta)$ is the linear mapping $\tau(\theta) = D\,\theta$.

According to the signature, *method* trafo will return different return value types. For signatures Estimate,**missing**, Estimate,ParamFamParameter, and ParamFamily,ParamFamParameter, it will return a list with entries fct, the function $\tau$, and **mat**, the matrix $\frac{\partial}{\partial\theta}\tau(\theta)$. function $\tau$ will then return the list **list**(fval, **mat**) mentioned above. For signatures ParamFamily,**missing** and ParamFamParameter,**missing**, it will just return the corresponding matrix.

From version 2.1 on, there are helper functions trafo.fct() (see ?trafo.fct) and trafoEst. While trafo.fct() allows to access "function" aspect of the transformation, returning the corresponding function, trafoEst transforms an existing estimator of class estimate consistently (i.e.; with corresponding untransfromed.estimate and untransformed.asvar information and transformed asvar) by a "trafo" function; see ?trafoEst.

## 4.4   Risk Classes

The risk classes are up to now (i.e, version 2.0) not used inside of the distr-family. They are however used in the RobASt-family [4]. We distinguish between various finite-sample and asymptotic risks; confer Figure 24. The bias and norm classes given in Figure 22 and Figure 23, respectively, occur as slots of the risk classes.
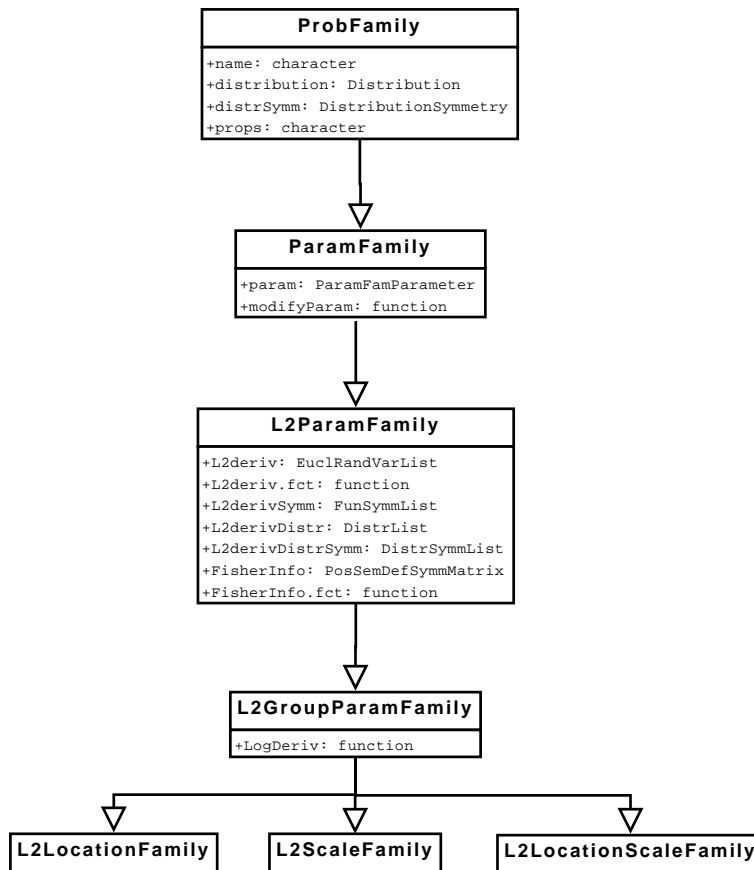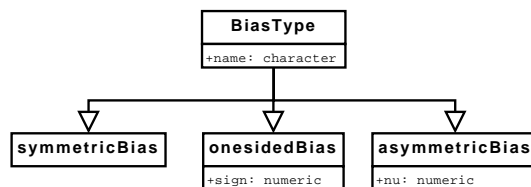


Figure 22: Inheritance relations and slots of the corresponding (sub-)classes for BiasType where we do not repeat inherited slots
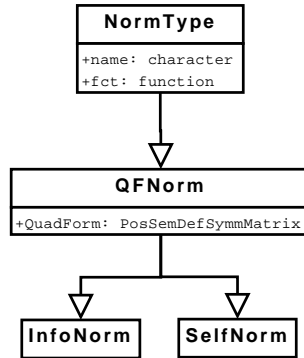
Figure 23: Inheritance relations and slots of the corresponding (sub-)classes for NormType where we do not repeat inherited slots
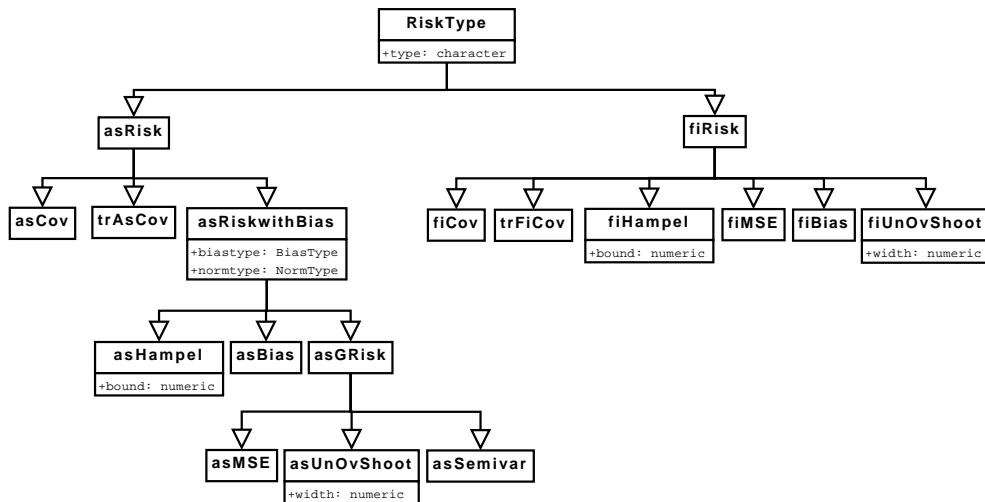


Figure 24: Inheritance relations and slots of the corresponding (sub-)classes for RiskType where we do not repeat inherited slots

## 4.5 Minimum Criterion Estimation

The S4-classes and methods defined inside of our distr-family enable us to define general functions for the computation of minimum criterion estimators – in particular, minimum distance and maximum likelihood (i.e., minimum negative log-likelihood) estimators. The main function for this purpose is MCEstimator. As an example we can use the negative log-likelihood as criterion; i.e., compute the maximum likelihood estimator.

```
library(distrMod)
x <- rgamma(50, scale = 0.5, shape = 3)
G <- GammaFamily(scale = 1, shape = 2)
negLoglikelihood <- function(x, Distribution){
    res <- -sum(log(Distribution@d(x)))
    names(res) <- "Negative Log-Likelihood"
    return(res)
}
MCEstimator(x = x, ParamFamily = G, criterion = negLoglikelihood)

## Evaluations of Minimum criterion estimate:
## ------------------------------------------
## An object of class "MCEstimate"
## generated by call
##   MCEstimator(x = x, ParamFamily = G, criterion = negLoglikelihood)
## samplesize:   50
## estimate:
##     scale      shape
## 0.4234807 3.5919495
## Criterion:
##
## 54.98167
```

The user can specialize the behavior of MCEstimator on two layers: instance-individual or class-individual.

Using the first layer, we may specify model-individual starting values / search intervals by slot startPar of class ParamFamily, pass on special control parameters to functions **optim** / **optimize** by a ... argument in function MCEstimator, and we may enforce valid parameter values by specifying function slot makeOKPar of class ParamFamily; also one can specify a penalty value penalizing invalid parameter values. E.g.; in case of the censored Poisson distribution family in demo `censoredPois` to this package these functions are defined as

```
    ## search interval for reasonable parameters
    startPar <- function(x,...) c(.Machine$double.eps,max(x))

    ## what to do in case of leaving the parameter domain
    makeOKPar <- function(param) {if(param<=0) return(.Machine$double.eps)
                                    return(param)}
```

In some situations, one would rather like to define rules for groups of models or to be even more flexible; this can be achieved using the class-individual layer: We may use method dispatch to find the "right" function to determine the MC estimator; to this end subclasses to class L2ParamFamily have to be defined, which has alread been done, e.g. in case of class PoisFamily. In general these sub classes will not have any new slots. E.g.; the code to define class PoisFamily simply is

```
setClass("PoisFamily", contains = "L2ParamFamily")
```

For group models, like the location scale model, there may be additional slots and intermediate classes. E.g.,

```
setClass("NormLocationFamily", contains = "L2LocationFamily")
```

Then, for these subclasses, particular methods may be defined; so far, in package "distrMod" we have particular validParameter methods for classes ParamFamily, L2ScaleFamily, L2LocationFamily, and L2LocationScaleFamily. E.g.; the code to signature L2ScaleFamily simply is

```
setMethod("validParameter", signature(object = "L2ScaleFamily"),
          function(object, param, tol=.Machine$double.eps){
              if(is(param,"ParamFamParameter"))
                  param <- main(param)
              if(!all(is.finite(param))) return(FALSE)
              if(length(param)!=1) return(FALSE)
              return(param > tol)})
```

To move the whole model from one parameter value to the other, so far we have modifyModel methods for classes L2ParamFamily, L2LocationFamily, L2ScaleFamily, L2LocationScaleFamily, GammaFamily, and ExpScaleFamily, where the second argument to dispatch on so for has to be of class ParamFamParameter. E.g.; the code to signature **model**="GammaFamily" is

```
setMethod("modifyModel", signature(model = "GammaFamily",
            param = "ParamFamParameter"),
          function(model, param, ...){
             M <- modifyModel(as(model, "L2ParamFamily"), param = param,
                                 .withCall = FALSE)
             M@L2derivSymm <- FunSymmList(OddSymmetric(SymmCenter =
                                                      prod(main(param))),
                                          NonSymmetric())
             class(M) <- class(model)
             return(M)
          })
```

We also allow for particular methods within function MCEstimator, as therein we call method mceCalc; so far there only is a method for signature(x="numeric", PFam="ParamFamily") Similarly, and more important, the same technique is applied for the wrapper function MLEstimator.

In case of the maximum likelihood estimator as well as in case of minimum distance (MD) estimation there are the function MLEstimator and MDEstimator which provide user-friendly interfaces to MCEstimator. Hence, the maximum likelihood estimator and for instance the Kolmogorov MD estimator can more easily be computed as follows.

```
    MLEstimator(x = x, ParamFamily = G)

## Evaluations of Maximum likelihood estimate:
## --------------------------------------------
## An object of class "MLEstimate"
## generated by call
##   MLEstimator(x = x, ParamFamily = G)
## samplesize:   50
## estimate:
##      scale        shape
##   0.42348070   3.59194948
##  (0.08701168) (0.68764060)
## asymptotic (co)variance (multiplied with samplesize):
##            scale     shape
## scale  0.3785516 -2.787382
## shape -2.7873816 23.642480
## Criterion:
## negative log-likelihood
##                54.98167
```

```
    MDEstimator(x = x, ParamFamily = G, distance = KolmogorovDist)

## Evaluations of Minimum Kolmogorov distance estimate  :
## ------------------------------------------------------
## An object of class "MDEstimate"
## generated by call
##   MDEstimator(x = x, ParamFamily = G, distance = KolmogorovDist)
## samplesize:   50
## estimate:
##     scale      shape
## 0.4118344 3.8438999
## Criterion:
## Kolmogorov distance
##          0.06141999
```

Within MLEstimator, we call method mleCalc, which then dispatches according to its arguments x and PFam as in case of method mceCalc. So far x must inherit from class **numeric**, and there are particular methods for argument PFam of classes ParamFamily, BinomFamily, PoisFamily, NormLocationFamily, NormScaleFamily, and NormLocationScaleFamily. More specifically, mleCalc must have an extra ... argument to cope with different callings from MLEstimator; additional arguments are possible of course. The return value must be a list with prescribed structure; to this end function meRes() is helpful which produces this structure. E.g. the mleCalc-method for signature(x="numeric", PFam="NormScaleFamily") is

```
setMethod("mleCalc", signature(x = "numeric", PFam = "NormScaleFamily"),
          function(x, PFam, ...){
          n <- length(x)
          theta <- sqrt((n-1)/n)*sd(x); mn <- mean(distribution(PFam))
          ll <- -sum(dnorm(x, mean=mn, sd = theta, log=TRUE))
          names(ll) <- "neg.Loglikelihood"
          crit.fct <- function(sd)
                        -sum(dnorm(x, mean=mn, sd = sd, log=TRUE))
          param <- ParamFamParameter(name = "scale parameter",
                              main = c("sd"=theta))
          if(!hasArg(Infos)) Infos <- NULL
          return(meRes(x, theta, ll, param, crit.fct, Infos = Infos))
})
```

We also provide a coercion to class mle from package **"stats4"**, hence making profiling by the **profile**-method therein possible. In order to be able to do so, we need to fill a functional slot criterion.fct of class MCEstimate. In many examples this is straightforward,

80

but in higher dimensions, helper function **get**.criterion.fct can be useful, e.g. it handles the general case for signature(PFam="ParamFamily").

The results of our computations in functionsMCEstimator, MDEstimator, and MLEstimator are objects of S4-class MCEstimate which inherits from S4-class Estimate. The definitions are given in Figure 25. For class MCEstimate, we have a method confint, which produces
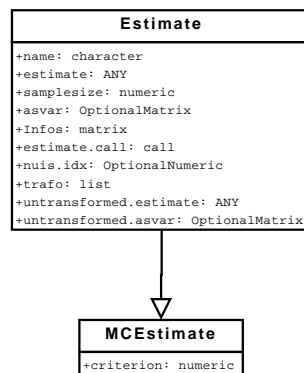


Figure 25: Inheritance relations and slots of the corresponding (sub-)classes for Estimate where we do not repeat inherited slots

confidence intervals (of class Confint). For class Confint as well as for class Estimate we have particular **show** and **print** methods where you may scale the output by setting global options with distrModOptions, see also subsection 5.3. As example consider the following:

```r
require(distrMod)
## some transformation
mtrafo <- function(x){
     nms0 <- c("scale","shape")
     nms <- c("shape","rate")
     fval0 <- c(x[2], 1/x[1])
     names(fval0) <- nms
     mat0 <- matrix( c(0, -1/x[1]^2, 1, 0), nrow = 2, ncol = 2,
                     dimnames = list(nms,nms0))
     list(fval = fval0, mat = mat0)}

set.seed(124)
x <- rgamma(50, scale = 0.5, shape = 3)

## parametric family of probability measures
G <- GammaFamily(scale = 1, shape = 2, trafo = mtrafo)
## MLE
```

```
res <- MLEstimator(x = x, ParamFamily = G)
print(res, digits = 4, show.details="maximal")

## Evaluations of Maximum likelihood estimate:
## --------------------------------------------
## An object of class "MLEstimate"
## generated by call
##   MLEstimator(x = x, ParamFamily = G)
## samplesize:   50
## estimate:
##    shape     rate
##   3.1519   2.3789
##  (0.6000) (0.4909)
## asymptotic (co)variance (multiplied with samplesize):
##       shape  rate
## shape 18.00 13.59
## rate  13.59 12.05
## untransformed estimate:
##     scale     shape
##   0.42036   3.15193
##  (0.08675) (0.60002)
## asymptotic (co)variance of untransformed estimate (multiplied with samplesize):
##         scale  shape
## scale  0.3763 -2.401
## shape -2.4008 18.001
## Transformation of main parameter:
## function(x){
##      nms0 <- c("scale","shape")
##      nms <- c("shape","rate")
##      fval0 <- c(x[2], 1/x[1])
##      names(fval0) <- nms
##      mat0 <- matrix( c(0, -1/x[1]^2, 1, 0), nrow = 2, ncol = 2,
##                      dimnames = list(nms,nms0))
##      list(fval = fval0, mat = mat0)}
## <bytecode: 0x0000022a2734ca10>
## Trafo / derivative matrix:
##        scale shape
## shape  0.000     1
## rate  -5.659     0
## Criterion:
## negative log-likelihood
```

```
##                      50.59

print(res, digits = 4, show.details="medium")

## Evaluations of Maximum likelihood estimate:
## -------------------------------------------
## An object of class "MLEstimate"
## generated by call
##   MLEstimator(x = x, ParamFamily = G)
## samplesize:   50
## estimate:
##    shape     rate
##   3.1519   2.3789
##  (0.6000) (0.4909)
## asymptotic (co)variance (multiplied with samplesize):
##       shape  rate
## shape 18.00 13.59
## rate  13.59 12.05
## Criterion:
## negative log-likelihood
##                     50.59

print(res, digits = 4, show.details="minimal")

## Evaluations of Maximum likelihood estimate:
## -------------------------------------------
##    shape     rate
##   3.1519   2.3789
##  (0.6000) (0.4909)

ci <- confint(res)
print(ci, digits = 4, show.details="maximal")

## A[n] asymptotic (CLT-based) confidence interval:
##       2.5 % 97.5 %
## shape 1.976  4.328
## rate  1.417  3.341
## Type of estimator: Maximum likelihood estimate
## samplesize:   50
## Call by which estimate was produced:
## MLEstimator(x = x, ParamFamily = G)
```

```
## Transformation of main parameter by which estimate was produced:
## function(x){
##      nms0 <- c("scale","shape")
##      nms <- c("shape","rate")
##      fval0 <- c(x[2], 1/x[1])
##      names(fval0) <- nms
##      mat0 <- matrix( c(0, -1/x[1]^2, 1, 0), nrow = 2, ncol = 2,
##                      dimnames = list(nms,nms0))
##      list(fval = fval0, mat = mat0)}
## <bytecode: 0x0000022a2734ca10>
## Trafo / derivative matrix at which estimate was produced:
##        scale shape
## shape  0.000     1
## rate   -5.659     0

print(ci, digits = 4, show.details="medium")

## A[n] asymptotic (CLT-based) confidence interval:
##       2.5 % 97.5 %
## shape 1.976  4.328
## rate  1.417  3.341
## Type of estimator: Maximum likelihood estimate
## samplesize:   50
## Call by which estimate was produced:
## MLEstimator(x = x, ParamFamily = G)

print(ci, digits = 4, show.details="minimal")

## A[n] asymptotic (CLT-based) confidence interval:
##       2.5 % 97.5 %
## shape 1.976  4.328
## rate  1.417  3.341

## some profiling
par(mfrow=c(2,1))
plot(profile(res))
```
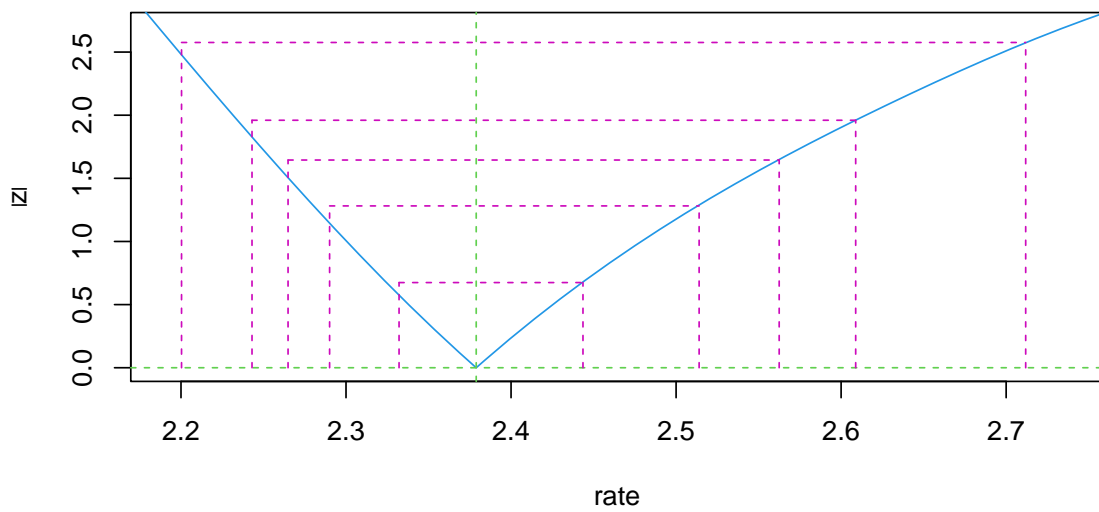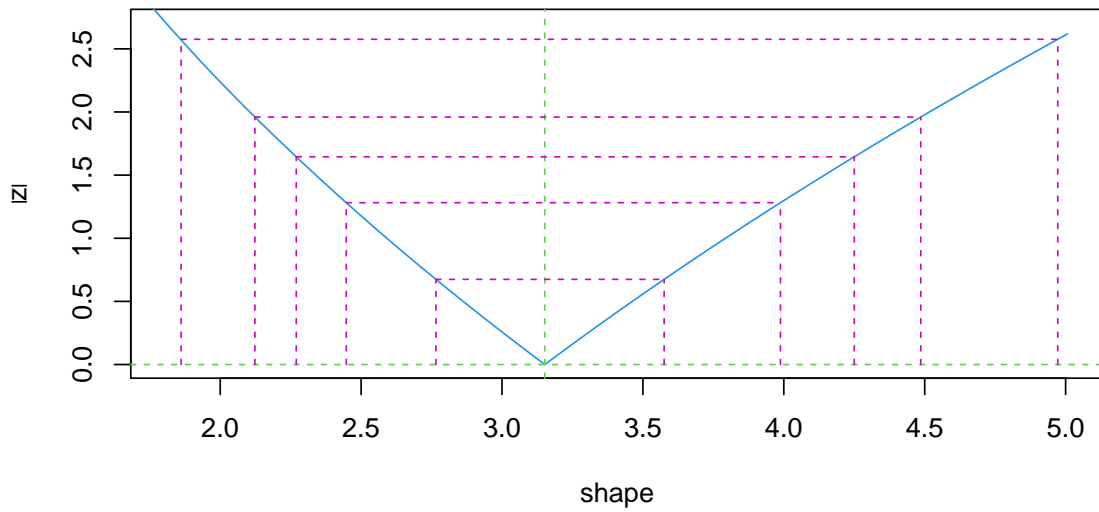
# 5 Options

## 5.1 Options for "distr"

Analogously to the **options** command in R you may specify a number of global "constants" to be used within the package. These include

- DefaultNrFFTGridPointsExponent: the binary logarithm of the number of grid-points used

in FFT —default 12

- DefaultNrGridPoints: number of grid-points used for a continuous variable —default 4096

- DistrResolution: the finest step length that is permitted for a grid for a discrete variable —default 1e–06

- RtoDPQ.e: For simulational determination of d, p and **q**, $10^{\text{RtoDPQ.e}}$ random variables are simulated —default 5

- TruncQuantile: to work with compact support, random variables are truncated to their lower/upper TruncQuantile-quantile —default 1e–05.
  From version 1.9 on, for $\varepsilon = $ TruncQuantile, we use calls of form **q**(X)(eps, **lower**.tail = FALSE) instead of **q**(X)(1−eps) to gain higher precision.

- warningSim: controls whether a warning issued at printing/showing a Distribution object the slots of which have been filled starting with simulations —default TRUE

- warningArith: controls whether a warning issued at printing/showing a Distribution object produced by arithmetics operating on distributions —default TRUE

- withgaps: controls whether in the return value of arithmetic operations the slot gaps of an the AbscontDistribution part is filled automatically based on empirical evaluations via setgaps —default TRUE

- simplifyD: controls whether in the return value of arithmetic operations there is a call to simplifyD or not —default TRUE

- DistrCollapse: logical; shall support points with distance smaller than DistrResolution be collapsed; default value: TRUE

- withSweave: logical; is code run in Sweave (then no new graphic devices are opened); default value: FALSE

- DistrCollapse.Unique.Warn: logical; shall a warning be issued upon collapsing? default value: TRUE

- use.generalized.**inverse.by.default** which is a logical variable giving the default value for argument generalized of our method **solve** in package "distrMod". This argument decides whether our method **solve** is to use generalized inverses if the original **solve**-method from package "base" fails; if the option is set to FALSE, in case of failure, and unless argument generalized is not explicitly set to TRUE, **solve** will throw an error as is the "base"-method behavior. The default value of thie option is TRUE.

All current options may be inspected by distroptions() and modified by distroptions("<options−name>"=<value>).
As **options**, distroptions("<options−name>") returns a list of length 1 with the value of the corresponding option, so here, just as **getOption**, getdistrOption("<options−name>") will be preferable, which only returns the value.

## 5.2 Options for `"distrEx"`

Up to version `0.4-4` we used the function distrExOptions(arg = "missing", value = −1) to manage some global options for `"distrEx"`, i.e.:
distrExOptions() returns a list of these options, distrExOptions(arg=x) returns option x, and distrExOptions(arg=x,value=y) sets the value of option x to y.
From version `1.9` on, we use a mechanism analogue to the distroptions/getdistrOption commands: You may specify certain global output options to be used within the package with distrExoptions/getdistrExOption. These include

- MCIterations: number of Monte-Carlo iterations used for crude Monte-Carlo integration; defaults to 1e5.

- GLIntegrateTruncQuantile: If integrate fails and there are infinite integration limits, the function GLIntegrate is called inside of distrExIntegrate with the corresponding quantiles GLIntegrateTruncQuantile resp. 1−GLIntegrateTruncQuantile as finite integration limits; defaults to 10∗.**Machine$double**.eps.

- GLIntegrateOrder: The order used for the Gauß-Legendre integration inside of distrExIntegrate; defaults to 500.

- ElowerTruncQuantile: The lower limit of integration used inside of E which corresponds to the ElowerTruncQuantile-quantile; defaults to 1e−7.

- EupperTruncQuantile: The upper limit of integration used inside of E which corresponds to the (1−ElowerTruncQuantile)-quantile; defaults to 1e−7.

- ErelativeTolerance: The relative tolerance used inside of E when calling distrExIntegrate; defaults to .**Machine$double**.eps^0.25.

- m1dfLowerTruncQuantile: The lower limit of integration used inside of m1df which corresponds to the m1dfLowerTruncQuantile-quantile; defaults to 0.

- m1dfRelativeTolerance: The relative tolerance used inside of m1df when calling distrExIntegrate; defaults to .**Machine$double**.eps^0.25.

- m2dfLowerTruncQuantile: The lower limit of integration used inside of m2df which corresponds to the m2dfLowerTruncQuantile-quantile; defaults to 0.

- m2dfRelativeTolerance: The relative tolerance used inside of m2df when calling distrExIntegrate; defaults to .**Machine$double**.eps^0.25.

- nDiscretize: number of support values used for the discretization of objects of class "AbscontDistribution"; defaults to 100.

- hSmooth: smoothing parameter to smooth objects of class "DiscreteDistribution". This is done via convolution with the normal distribution Norm(**mean** = 0, **sd** = hSmooth); defaults to 0.05.

- **IQR**.fac: for determining sensible integration ranges, we use both quantile and scale based methods; for the scale based method we use the median of the distribution ± **IQR**.fac × the IQR; defaults to 15.

## 5.3  Options for "distrMod"

Just as with to the distroptions/getdistrOption commands you may specify certain global output options to be used within the package with distrModoptions/ getdistrModOption. These include

- **show**.details which controls the detailedness for method **show** for objects of classes of the "distrXXX" family of packages. Possible values are

  - "maximal": all information is shown
  - "minimal": only the most important information is shown
  - "medium": somewhere in the middle; see actual **show**-methods for details.

  The default value is "maximal".

## 5.4  Options for "distrSim"

Just as with to the distroptions/getdistrOption commands you may specify certain global output options to be used within the package with distrSimoptions/ getdistrSimOption. These include

- MaxNumberofPlottedObs the maximal number of observation plotted in a plot of an object of class Dataclass; defaults to 4000

- MaxNumberofPlottedObsDims: the maximum number of observations to be plotted in a plot of an object of class Dataclass and descendants; defaults to 6.

- MaxNumberofPlottedRuns: the maximum number of runs to be plotted in a plot of an object of class Dataclass and descendants (one run/panel); defaults to 6.

- MaxNumberofSummarizedObsDims: the maximum number of observations to be summarized of an object of class Dataclass and descendants; defaults to 6.

- MaxNumberofSummarizedRuns: the maximum number of runs to be summarized of an object of class Dataclass and descendants; defaults to 6.

## 5.5  Options for "distrTEst"

Just as with to the distroptions/getdistrOption commands you may specify certain global output options to be used within the package with distrTEstoptions/ getdistrTEstOption. These include

- MaxNumberofPlottedEvaluations: the maximal number of evaluations to be plotted in a plot of an object of class EvaluationList; defaults to 6

- MaxNumberofPlottedEvaluationDims: the maximal number of evaluation dimensions to be plotted in a plot of an object of class Evaluation; defaults to 6

- MaxNumberofSummarizedEvaluations: the maximal number of evaluations to be summarized of an object of class EvaluationList; defaults to 15

- MaxNumberofPrintedEvaluations: the maximal number of evaluations printed of an object of class EvaluationList; defaults to 15

# 6  Further Documentation

## 6.1  Help pages

Additional information can be obtained during an R session, using help files and startup messages.

**Startup messages**  Upon loading packages from the distrXXX family of packages with require or library, by default startup messages pop up giving some starting points where to look for further information; howto scale/switch off these startup messages will be discussed in the next section.

**Package help file**  A starting point is doubtless the package help files, called e.g. by ?distr. We give an ASCII form of a class graph with all the S4 classes in the corresponding package, and indicate the implemented methods and functions. From version 2.1 this help file for "distr" has a new section "Extension packages" pointing to the various extensions of this package.

**Individual help files**   Of course, for looking up the syntax of a function the help pages available with ? / **help** are extremely useful. Note that for our packages, help pages are also available for internal (i.e.; non exported) functions.

## 6.2   `NEWS` file

To get more details on the implementation process you may also consult a corresponding `NEWS` file where all relevant changes are listed at the package version where they have been implemented. You may inspect this `NEWS` file with NEWS(<pkgname>).

## 6.3   Vignettes

Besides this large (or huge) vignette, there are also some minor vignettes available, assembling documentation somewhat more coherent than in the help pages. From version 2.1 on, e.g., there is a new vignette "How to generate new distributions in packages distr, distrEx" in package `"distr"` which is to encourage the implementation of new distributions and distribution classes by third parties.

## 6.4   Articles

Of course we plan to publish some of our findings in peer-reviewed journals; so the part of this vignette dealing with package `"distrMod"` will form the basis for an article on this package by the present authors.

# 7   Startup Messages

For the management of startup messages, from version 1.7, we use package `"startupmsg"`: When loading/attaching packages `"distr"`, `"distrEx"`, `"distrSim"`, or `"distrTEst"` for each package a disclaimer is displayed.

You may suppress these start-up banners/messages completely by setting **options**("StartupBanner"="off") somewhere before loading this package by **library** or **require** in your R-code / R-session.

If option "StartupBanner" is not defined (default) or setting **options**("StartupBanner"= NULL) or **options**("StartupBanner"= "complete") the complete start-up banner is displayed.

For any other value of option "StartupBanner" (i.e., not in **c**(NULL, "off", "complete")) only the version information is displayed.

The same can be achieved by wrapping the **library** or **require** call into either onlytypeStartupMessages(<code>, atypes="version") or suppressStartupMessages(<code>).

# 8 System/version requirements, license, etc.

## 8.1 System requirements

As our package is completely written in R, there are no dependencies on the underlying OS; of course, there is the usual speed gain possible on recent machines. We have tested our package on a Pentium II with 233 MHz, on Pentium III's with 0.8–2.1 GHz, and on an Athlon with 2.5 GHz giving a reasonable performance.

## 8.2 Required version of R

Contrary to the hardware required, if you want to use library or require to use "distr" within R code, you need at least R Version 1.8.1, as we make use of name space operations only available from that version on; also, the command setClassUnion, which is used in some sources, is only available from that version on.
On the other hand, if the package may be pasted in by source, the code works with R from version 1.7.0 on —but without using name-spaces, which is only available from 1.8.0 on. Due to some changes in R from version 1.8.1 to 1.9.0 and from 1.9.1 to 2.0.0, we have to provide different zip/tar.gz-Files for these versions.
Versions of "distr" from version number 1.5 onwards are only supplied for R Version 2.0.1 patched and later. After a reorganization, versions of "distr" from version number 1.6 onwards are only supplied for R Version 2.2.0 patched and later.

## 8.3 Dependencies

In package "distr", from version 2.0, we make use of D1ss from Martin Mächler's package "sfsmisc". In package "distrEx", up to version 2.4, we needed Alec Stephenson's package "evd" for the extreme value distributions implemented therein, as well as Vincent Goulet's and Mathieu Pigeon's package "actuar" for the (single parameter) Pareto distribution (from "distrEx" version 2.1 on) which has been ported to our framework by Nataliya Horbenko. From version 2.4 on, this infrastructure has moved to package "RobExtremes".

In package "distrSim", and conseqently also in package "distrTEst" we use Paul Gilbert's package "setRNG" to be installed from CRAN for the control of the seed of the random number generator in our simulation classes. More precisely, for our version $\leq$ 1.6 we need his version $<$ 2006.2-1, and for our version $\geq$ 1.7 we need his version $\geq$ 2006.2-1.

From package version 1.7/0.4-3 on, we also need package "startupmsg" by the first of the present authors, which also is available on CRAN.

## 8.4 License

This software is distributed under the terms of the GNU GENERAL PUBLIC LICENSE Version 2, June 1991, more specifically under LGPL-3 confer

## 9   Details to the implementation

- As the normal distribution is closed under affine transformations, we have overloaded the corresponding methods.

- For the general convolution algorithm for univariate probability distribution functions/densities by means of FFT, which we use in the overloaded "+"-operator, confer [5].

- Exact convolution methods are implemented for the normal, the Poisson, the binomial, the negative binomial, the Gamma (and the Exp), and the $\chi^2$ distribution

- Exact formulae for scale transformations are implemented for the Exp-/Gamma-distribution, the Weibull and the log-normal distribution (the latter two from version 1.9 on).

- Exact formulae for affine linear transformations are available for the normal, the logistic and the Cauchy distribution (the latter two from version 1.9 on).

- Instances of any class transparent to the user are initialized by `<classname>([<slotname>=<value>,...])` where except for class DataClass in package "distrSim" all classes have default values for all their slots; in DataClass, the slot Data has to be specified.

- Multiplication (and Division) is implemented as corresponding exponentials of the convolution of the logarithms (evaluated separately for positive and negative parts).

- Exponentiation also uses the exp-log trick.

- Multiplication, Exponentation, and Min/Maximum of an AbscontDistribution and a DiscreteDistribution as an intermediate step produce a UnivarMixingDistribution, with one mixing component for each element of the support of the DiscreteDistribution. As a last step, this UnivarMixingDistribution is then "flattened".

- As suggested in [3] all slots are accessed and modified by corresponding accessor- and replacement functions —templates for which were produced by standardMethods.

  **We strongly discourage the use of the @-operator to modify or even access slots r, d, p, and q, confer Example 13.7.**

## 10  A general utility

Following [3], the programmer of S4-classes should provide accessor and replacement functions for the inspection/modification of any newly introduced slot. This can be quite a task when you have a lot of classes/slots. As these functions all have the same structure, it would be nice to automatically generate templates for them. Faced with this problem in developing this package, Thomas Stabla has written such a utility, standardMethods —which the authors of this package recommend for any developer of S4-classes. For more details, see ?standardMethods.

## 11  Odds and Ends

### 11.1  What should be done and what we could do —for version >2.9

- application of analytic FourierTransforms instead of FFT where appropriate —perhaps also to be controlled by a parameter/option

- use the q-slot applied to runif in simplifyr for continuous distributions

- further exact formulae for binary arithmetic operations like "*"

- goodness of fit tests for distribution-objects

- defining a subgroup of Math2 of invertible binary operators

### 11.2  What should be done but for which we lack the know-how

- multivariate distributions

- conditional distributions

- copula

## 12  Acknowledgement

In order to give our acknowledgements their due place in the manual, we insert them before some rather extensive presentation of examples, because otherwise they would probably get lost or overseen by most of the readers.

We thank Martin Mächler and Josef Leydold for their helpful suggestions in conceiving the package. John Chambers also gave several helpful hints and insights when responding to our requests concerning the S4-class concept in r-devel/ r-help. We got stimulating replies to an RFC on r-devel by Duncan Murdoch and Gregory Warnes. We also thank Paul Gilbert for drawing our attention to his package setRNG and making it available as

stand-alone version. In the last few days before the release on `CRAN`, Kurt Hornik and Uwe Ligges were very kind, helping us to find the clue how to pass all necessary checks by `R CMD check`. We also thank G. Jay Kerns for contributing code for the skewness and kurtosis functionals.

Last not least a big "thank you" to Torsten Hothorn as editor of `R-News`, for his patience with our endless versions until we finally came to a publishable version.

# 13 Examples

## 13.1 12-fold convolution of uniform $(0, 1)$ variables

This example shows how easily we may get the distribution of the sum of 12 i.i.d. ufo$(0, 1)$–variables minus 6— which was used as a fast generator of $\mathcal{N}(0, 1)$–variables in times when evaluations of exp, log, sin and tan were expensive, confer [7], example C, p. 163. The user should not be confused by expressions like `U+U`: this *does not* mean `2U` but rather convolution of two independent identically distributed random variables.

```
require(distr)

N <- Norm(0,1)
U <- Unif(0,1)
U2 <- U + U
U4 <- U2 + U2
U8 <- U4 + U4
U12 <- U4 + U8
NormApprox <- U12 - 6

x <- seq(-4,4,0.001)

opar <- par(no.readonly = TRUE)
par(mfrow = c(2,1))

plot(x, d(NormApprox)(x),
     type = "l",
     xlab = "",
     ylab = "Density",
     main = "Exact and approximated density")
lines(x, d(N)(x),
      col = "red")
legend("topleft",
       legend = c("NormApprox", "Norm(0,1)"),
```

```
        fill = c("black", "red"))

plot(x, d(NormApprox)(x) - d(N)(x),
     type = "l",
     xlab = "",
     ylab = "\"black\" - \"red\"",
     col = "darkgreen",
     main = "Error")
lines(c(-4,4), c(0,0))
```

**Exact and approximated density**
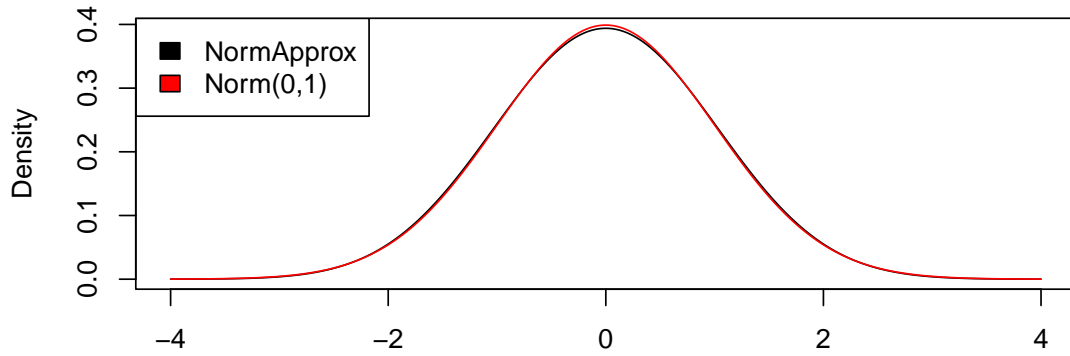


**Error**
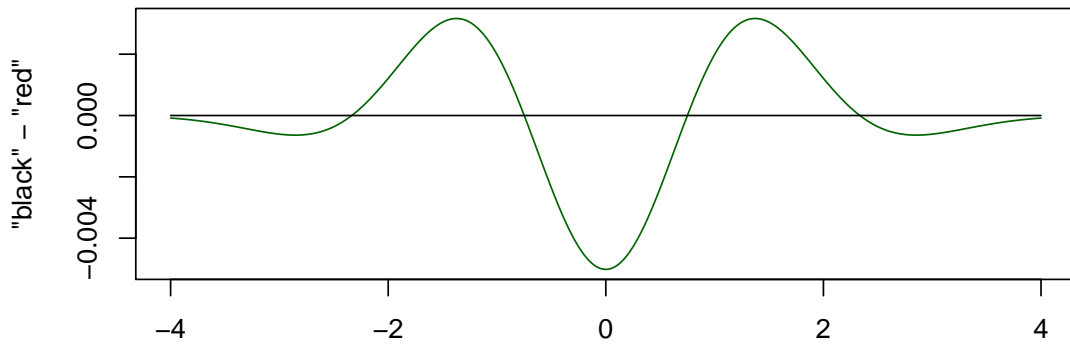
```
par(opar)
```

## 13.2 Comparison of exact convolution to FFT for normal distributions

This example illustrates the exactness of the numerical algorithm used to compute the convolution: We know that $\mathcal{L}(\mathtt{A} + \mathtt{B}) = \mathcal{N}(5, 13)$ — if the second argument of $\mathcal{N}$ is the variance

```r
require(distr)

## initialize two normal distributions
A <- Norm(mean=1, sd=2)
B <- Norm(mean=4, sd=3)

## convolution via addition of moments
AB <- A+B

## casting of A,B as absolutely continuous distributions
## that is, ``forget'' that A,B are normal distributions
A1 <- as(A, "AbscontDistribution")
B1 <- as(B, "AbscontDistribution")

## for higher precision we change the global variable
## "TruncQuantile" from 1e-5 to 1e-8
oldeps <- getdistrOption("TruncQuantile")
eps <- 1e-8
distroptions("TruncQuantile" = eps)
## support of A1+B1 for FFT convolution is
## [q(A1)(TruncQuantile),
##   q(B1)(TruncQuantile, lower.tail = FALSE)]

## convolution via FFT
AB1 <- A1+B1

############################
## plots of the results
############################
par(mfrow=c(1,3))
low <- q(AB)(1e-15)
upp <- q(AB)(1e-15, lower.tail = FALSE)
x <- seq(from = low, to = upp, length = 10000)

## densities
plot(x, d(AB)(x), type = "l", lwd = 5)
lines(x , d(AB1)(x), col = "orange", lwd = 1)
title("Densities")
legend("topleft", legend=c("exact", "FFT"),
        fill=c("black", "orange"))
```

```r
## cdfs
plot(x, p(AB)(x), type = "l", lwd = 5)
lines(x , p(AB1)(x), col = "orange", lwd = 1)
title("CDFs")
legend("topleft", legend=c("exact", "FFT"),
       fill=c("black", "orange"))

## quantile functions
x <- seq(from = eps, to = 1-eps, length = 1000)
plot(x, q(AB)(x), type = "l", lwd = 5)
lines(x , q(AB1)(x), col = "orange", lwd = 1)
title("Quantile functions")
legend("topleft", legend=c("exact", "FFT"),
       fill=c("black", "orange"))
```

**Densities** **CDFs** **Quantile functions**
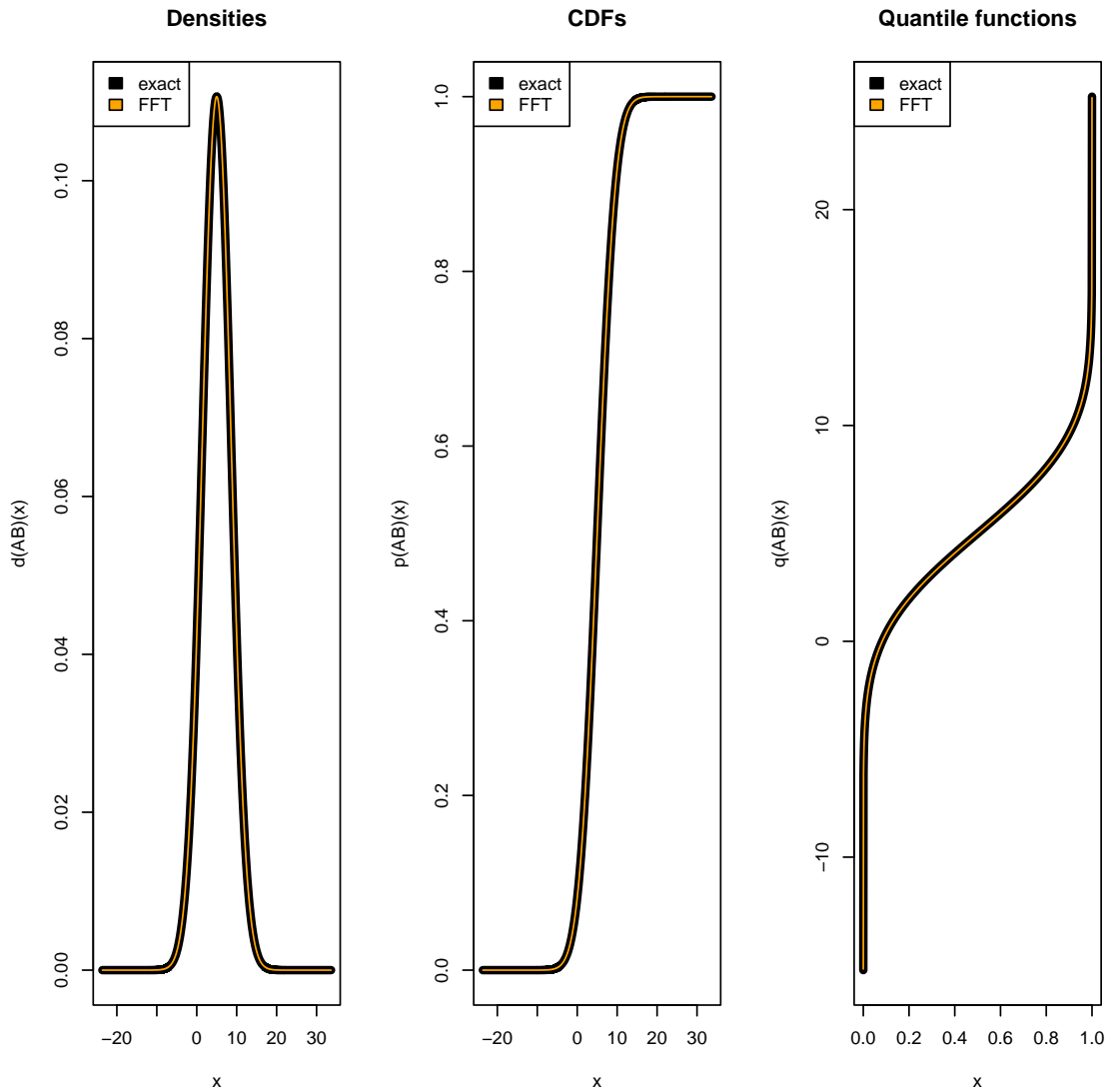
```
## Since the plots of the results show no
## recognizable differencies, we also compute
## the total variation distance of the densities
## and the Kolmogorov distance of the cdfs

## total variation distance of densities
total.var <- function(z, N1, N2){
    0.5*abs(d(N1)(z) - d(N2)(z))
```

```
}
dv <- integrate(total.var, lower=-Inf, upper=Inf, rel.tol=1e-8, N1=AB, N2=AB1)
cat("Total variation distance of densities:\t")

## Total variation distance of densities:

print(dv) # 4.25e-07

## 4.250016e-07 with absolute error < 1.8e-09

### meanwhile realized in package "distrEx"
### as TotalVarDist(N1,N2)

## Kolmogorov distance of cdfs
## the distance is evaluated on a random grid
z <- r(Unif(Min=low, Max=upp))(1e5)
dk <- max(abs(p(AB)(z)-p(AB1)(z)))
cat("Kolmogorov distance of cdfs:\t", dk, "\n")

## Kolmogorov distance of cdfs:  7.26889e-07

# 2.03e-07

### meanwhile realized in package "distrEx"
### as KolmogorovDist(N1,N2)

## old distroptions
distroptions("TruncQuantile" = oldeps)
```

## 13.3  Comparison of FFT to `RtoDPQ`

This example illustrates the exactness (or rather not–so–exactness) of the simulational default algorithm used to compute the distribution of transformations of group `math`.

```
require(distr)

###############################
## Comparison 1 - FFT and RtoDPQ
###############################
```

```
N1 <- Norm(0,3)
N2 <- Norm(0,4)
rnew1 <- function(n) r(N1)(n) + r(N2)(n)

X <- N1 + N2
    # exact formula -> N(0,5)
Y <- N1 + as(N2, "AbscontDistribution")
    # appoximated with FFT
Z <- new("AbscontDistribution", r = rnew1)
    # appoximated with RtoDPQ

# density-plot

x <- seq(-15,15,0.01)
plot(x, d(X)(x),
    type = "l",
    lwd = 3,
    xlab = "",
    ylab = "density",
    main = "Comparison 1",
    col = "black")
lines(x, d(Y)(x),
    col = "yellow")
lines(x, d(Z)(x),
    col = "red")
legend("topleft",
  legend = c("Exact", "FFT-Approximation",
             "RtoDPQ-Approximation"),
      fill = c("black", "yellow", "red"))
```

**Comparison 1**
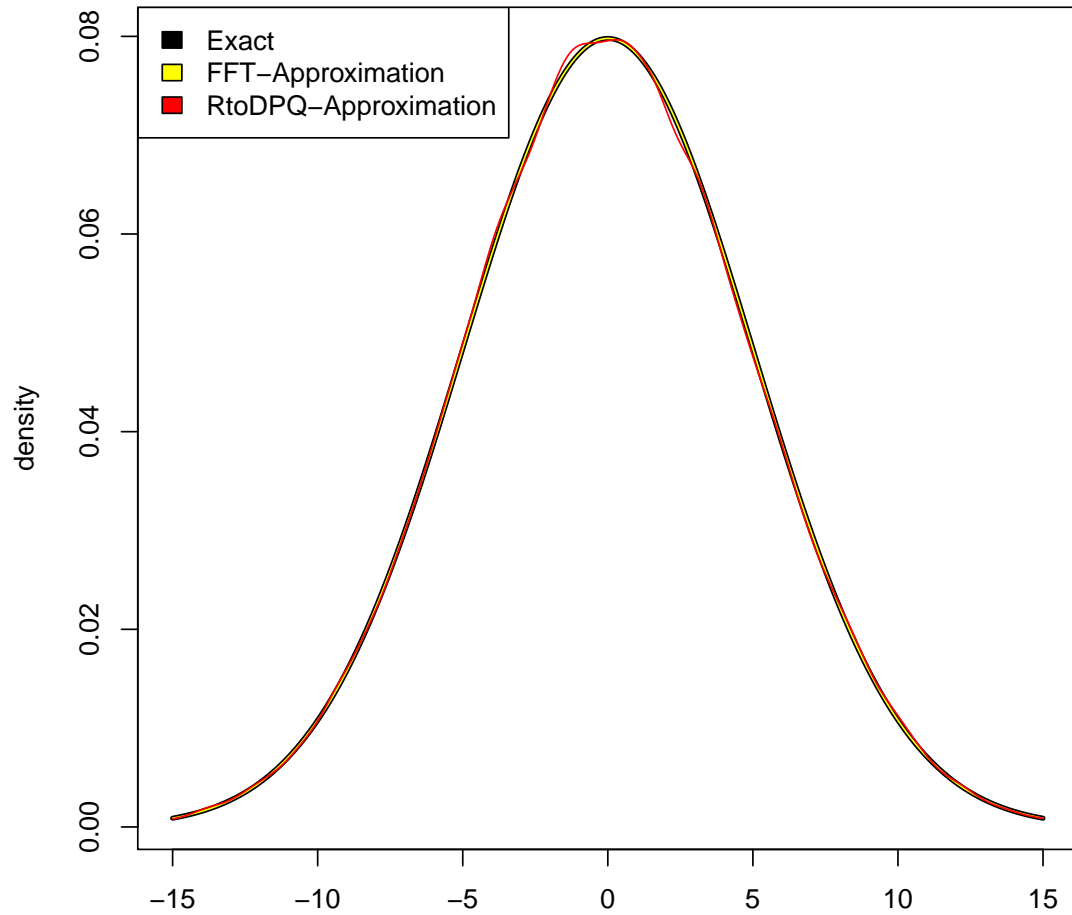
```
#########################################
## Comparison 2 - "Exact" Formula and RtoDPQ
#########################################

B <- Binom(size = 6, prob = 0.5) * 10
N <- Norm()
rnew2 <- function(n) r(B)(n) + r(N)(n)
```

```r
Y <- B + N
    # "exact" formula
Z <- new("AbscontDistribution", r = rnew2)
    # appoximated with RtoDPQ

# density-plot

x  <- seq(-5,65,0.01)
plot(x, d(Y)(x),
     type = "l",
     xlab = "",
     ylab = "density",
     main = "Comparison 2",
     col = "black")
lines(x, d(Z)(x),
     col = "red")
legend("topleft",
        legend = c("Exact", "RtoDQP-Approximation"),
        fill = c("black", "red"))
```

**Comparison 2**
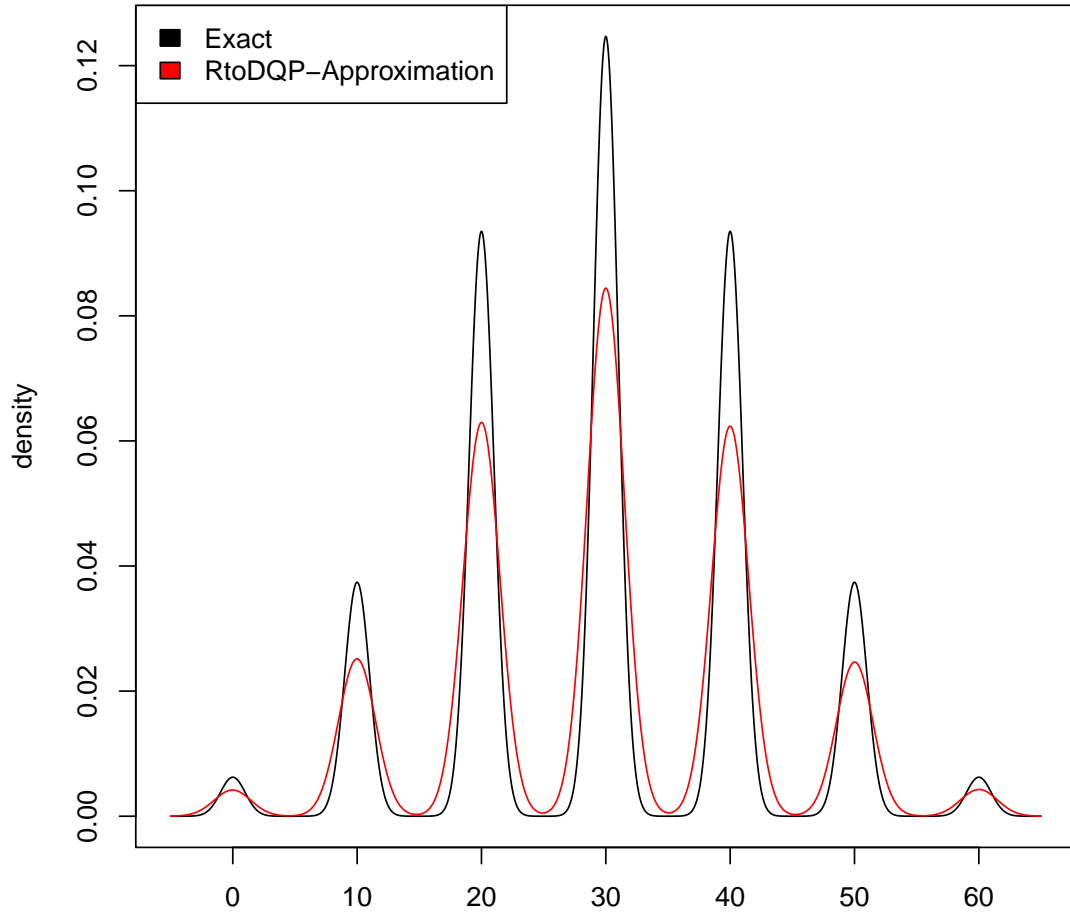


## 13.4 Comparison of exact and approximate stationary regressor distribution

Another illustration for the use of package `"distr"`. In case of a stationary AR(1)–model, for non–normal innovation distribution, the stationary distribution of the observations must be approximated by finite convolutions. That these approximations give fairly good results for approximations down to small orders is exemplified by the Gaussian case where we may compare the

approximation to the exact stationary distribution.

```r
require(distr)

## Approximation of the stationary regressor
## distribution of an AR(1) process
##        X_t = phi X_{t-1} + V_t
## where V_t i.i.d N(0,1) and phi\in(0,1)
## We obtain
##     X_t = \sum_{j=1}^\infty phi^j V_{t-j}
## i.e., X_t \sim N(0,1/(1-phi^2))
phi <- 0.5

## casting of V as absolutely continuous distributions
## that is, ``forget'' that V is a normal distribution
V <- as(Norm(), "AbscontDistribution")

## for higher precision we change the global variable
## "TruncQuantile" from 1e-5 to 1e-8
oldeps <- getdistrOption("TruncQuantile")
eps <- 1e-8
distroptions("TruncQuantile" = eps)

## Computation of the approximation
##      H=\sum_{j=1}^n phi^j V_{t-j}
## of the stationary regressor distribution
## (via convolution using FFT)
H <- V
n <- 15
## may take some time
### switch off warnings [would be issued due to
###  very unequal variances...]
old.warn <- getOption("warn")
options("warn" = -1)
for(i in 1:n){Vi <- phi^i*V; H <- H + Vi }
options("warn" = old.warn)

## the stationary regressor distribution (exact)
X <- Norm(sd=sqrt(1/(1-phi^2)))

###########################
```

```r
## plots of the results
############################
par(mfrow=c(1,3))
low <- q(X)(1e-15)
upp <- q(X)(1e-15, lower.tail = FALSE)
x <- seq(from = low, to = upp, length = 10000)

## densities
plot(x, d(X)(x),type = "l", lwd = 5)
lines(x , d(H)(x), col = "orange", lwd = 1)
title("Densities")
legend("topleft", legend=c("exact", "FFT"),
       fill=c("black", "orange"))

## cdfs
plot(x, p(X)(x),type = "l", lwd = 5)
lines(x , p(H)(x), col = "orange", lwd = 1)
title("CDFs")
legend("topleft", legend=c("exact", "FFT"),
       fill=c("black", "orange"))

## quantile functions
x <- seq(from = eps, to = 1-eps, length = 1000)
plot(x, q(X)(x),type = "l", lwd = 5)
lines(x , q(H)(x), col = "orange", lwd = 1)
title("Quantile functions")
legend( "topleft",
       legend=c("exact", "FFT"),
       fill=c("black", "orange"))
```

## Densities

## CDFs

## Quantile functions
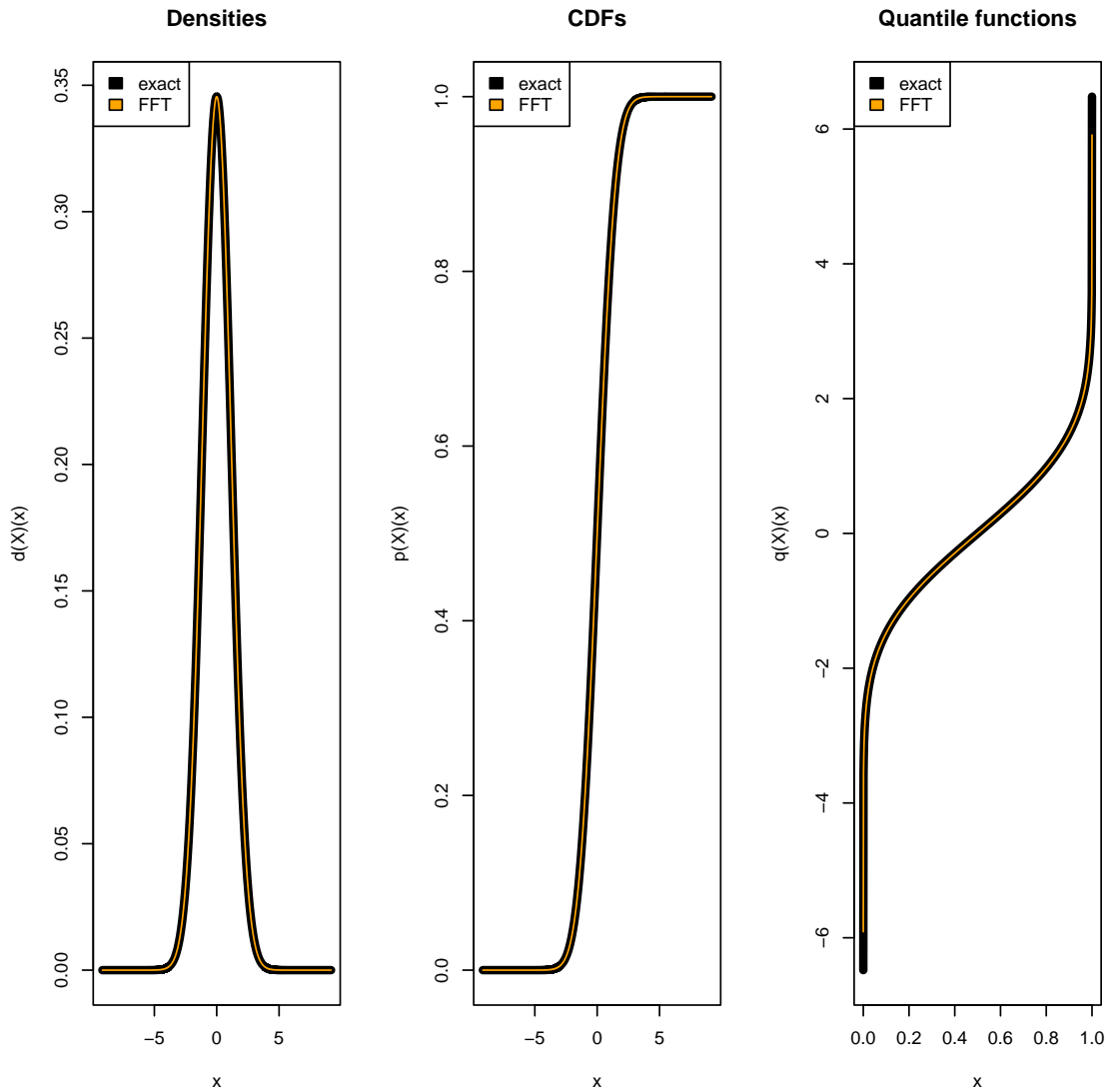


```
## Since the plots of the results show no
## recognizable differencies, we also compute
## the total variation distance of the densities
## and the Kolmogorov distance of the cdfs

## total variation distance of densities
total.var <- function(z, N1, N2){
    0.5*abs(d(N1)(z) - d(N2)(z))
```

```
}
dv <- integrate(f = total.var, lower = -Inf,
                upper = Inf, rel.tol = 1e-7,
                N1=X, N2=H)
cat("Total variation distance of densities:\t")

## Total variation distance of densities:

print(dv) # ~ 5.0e-06

## 2.091698e-05 with absolute error < 5.7e-08

### meanwhile realized in package "distrEx"
### as TotalVarDist(N1,N2)


## Kolmogorov distance of cdfs
## the distance is evaluated on a random grid
z <- r(Unif(Min=low, Max=upp))(1e5)
dk <- max(abs(p(X)(z)-p(H)(z)))
cat("Kolmogorov distance of cdfs:\t", dk, "\n")

## Kolmogorov distance of cdfs:   1.12417e-05

# ~2.5e-06

### meanwhile realized in package "distrEx"
### as KolmogorovDist(N1,N2)


## old distroptions
distroptions("TruncQuantile" = oldeps)
```

## 13.5 Truncation and Huberization/winsorization

has been integrated to the package itself, see section 3.8

## 13.6 Distribution of minimum and maximum of two independent random variables

has been integrated to the package itself, see section 3.8

## 13.7   Instructive destructive example

```
#########################################################
## Demo: Instructive destructive example
#########################################################
require(distr)

## package "distr" encourages
## consistency but does not
## enforce it---so in general
## d o    n o t    m o d i f y
## slots d,p,q,r!

N <- Norm()
B <- Binom()
N@d <- B@d
plot(N, lwd = 3)
```

Density of Norm(0, 1)　　　　CDF of Norm(0, 1)　　　　Quantile function of Norm(0, 1)
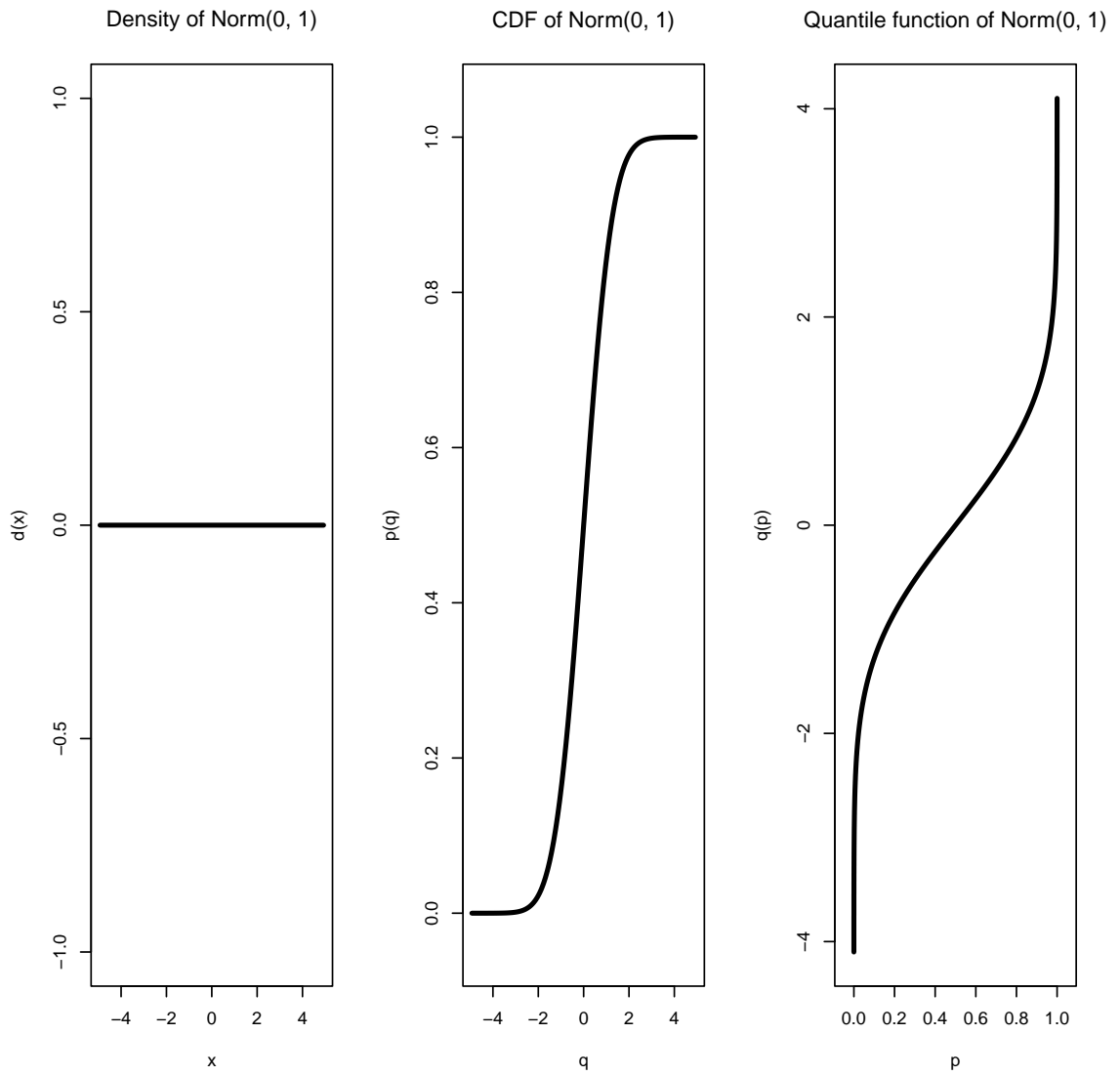


## 13.8   A simulation example

needs packages "distrSim"/"distrTEst"

```r
require(distrTEst)
    ### also loads distrSim
sim <- new("Simulation",
           seed = setRNG(),
           distribution = Norm(mean = 0, sd = 1),
           filename="sim_01",
           runs = 1000,
           samplesize = 30)

contsim <- new("Contsimulation",
               seed = setRNG(),
               distribution.id = Norm(mean = 0, sd = 1),
               distribution.c = Norm(mean = 0, sd = 9),
               rate = 0.1,
               filename="contsim_01",
               runs = 1000,
               samplesize = 30)

simulate(sim)
simulate(contsim)

sim

## filename of Simulation: sim_01
## Seed:   Kind: Mersenne-Twister
##         Normal Kind: Inversion
##         first 6 numbers: -0389363363  0850082084 -1226019697
##                          -0599765454 -0379619375 -1103098716
## number of runs: 1000
## dimension of the observations: 1
## size of sample: 30
## object was generated by version: 1.9
## Distribution:
## Distribution Object of Class: Norm
##  mean: 0
##  sd: 1

summary(contsim)

## name of simulation: contsim_01
## rate of contamination: 0.100000
```

```
## real Data:
## dimension of the observations: 1
## number of runs: 1000
## size of sample: 30
## , , 1
##
##                [,1]
## Min.    -2.33653712
## 1st Qu. -0.88422755
## Median  -0.08545112
## Mean     0.30888133
## 3rd Qu.  0.66490292
## Max.     7.36461912
##
## , , 2
##
##                [,1]
## Min.    -3.4141891
## 1st Qu. -0.9002957
## Median  -0.1397413
## Mean     0.0230223
## 3rd Qu.  0.6146120
## Max.     9.0219574
##
## , , 3
##
##                [,1]
## Min.    -2.0086545
## 1st Qu. -0.5478159
## Median   0.1515300
## Mean     0.7377169
## 3rd Qu.  0.6945067
## Max.    16.5871009
##
## , , 4
##
##                [,1]
## Min.    -4.61269357
## 1st Qu. -0.87213899
## Median   0.17070796
```

```
## Mean     -0.03209197
## 3rd Qu.   0.47466974
## Max.      4.32216887
##
## , , 5
##
##                  [,1]
## Min.    -2.78195041
## 1st Qu. -0.95992099
## Median  -0.03409165
## Mean    -0.08187346
## 3rd Qu.  0.74824274
## Max.     2.36619347
##
## , , 6
##
##                  [,1]
## Min.   -13.62790113
## 1st Qu.  -0.51659510
## Median   0.43284614
## Mean    -0.07329117
## 3rd Qu.  0.74732946
## Max.     7.17196684
```

```
plot(contsim)
```
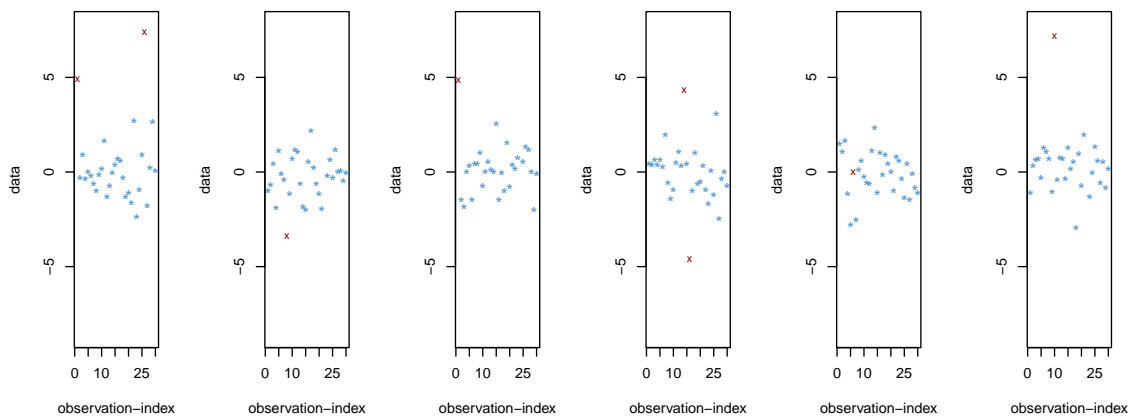
```
require(distrTEst)
psim <- function(theta,y,m0){
  mean(pmin(pmax(-m0, y - theta), m0))
  }
mestimator <- function(x, m = 0.7) {
  uniroot(f = psim,
          lower = -20,
          upper = 20,
          tol = 1e-10,
          y = x,
          m0 = m,
          maxiter = 20)$root
  }


result.id.mean <- evaluate(sim, mean)
result.id.mest <- evaluate(sim, mestimator)
result.id.median <- evaluate(sim, median)



result.cont.mean <- evaluate(contsim, mean)
result.cont.mest <- evaluate(contsim, mestimator)
result.cont.median <- evaluate(contsim, median)


elist <- EvaluationList(result.cont.mean,
                        result.cont.mest,
                        result.cont.median)


elist

## An EvaluationList Object
## name of Evaluation List: a list of "Evaluation" objects
## name of Dataobject: object
## name of Datafile: contsim_01
## --------------------------------
## An Evaluation Object
## estimator: mean
## Result: 'data.frame': 1000 obs. of  2 variables:
##  $ mean.id: num  -0.1342 -0.1564 -0.021 -0.0722 -0.078 ...
##  $ mean.re: num  0.3089 0.023 0.7377 -0.0321 -0.0819 ...
## --------------------------------
## An Evaluation Object
```

```
## estimator: mestimator
## Result: 'data.frame': 1000 obs. of  2 variables:
##  $ mstm.id: num  -0.2308 -0.1202 0.0288 -0.0801 -0.0186 ...
##  $ mstm.re: num  -0.0839 -0.1553 0.1994 -0.0153 -0.0275 ...
## --------------------------------
## An Evaluation Object
## estimator: median
## Result: 'data.frame': 1000 obs. of  2 variables:
##  $ medn.id: num  -0.1623 -0.1397 0.0348 0.0584 -0.0112 ...
##  $ medn.re: num  -0.0855 -0.1397 0.1515 0.1707 -0.0341 ...

summary(elist)

## name of Evaluation List: a list of "Evaluation" objects
## name of Dataobject: object
## name of Datafile: contsim_01
## --------------------------------
## name of Evaluation: object
## estimator: mean
## Result:
##      mean.id              mean.re
##  Min.    :-0.589662   Min.    :-2.048397
##  1st Qu.:-0.117384    1st Qu.:-0.335711
##  Median : 0.002348    Median :-0.003919
##  Mean    :-0.002694   Mean    : 0.009400
##  3rd Qu.: 0.108640    3rd Qu.: 0.372806
##  Max.    : 0.768604   Max.    : 2.462496
## --------------------------------
## name of Evaluation: object
## estimator: mestimator
## Result:
##      mstm.id              mstm.re
##  Min.    :-0.636670   Min.    :-0.731594
##  1st Qu.:-0.127130    1st Qu.:-0.146029
##  Median :-0.003956    Median :-0.007322
##  Mean    :-0.003831   Mean    :-0.002743
##  3rd Qu.: 0.122138    3rd Qu.: 0.147472
##  Max.    : 0.730241   Max.    : 0.842396
## --------------------------------
## name of Evaluation: object
## estimator: median
```

```
## Result:
##     medn.id              medn.re
##  Min.   :-0.8041779   Min.   :-0.873215
##  1st Qu.:-0.1501175   1st Qu.:-0.169997
##  Median :-0.0000101   Median :-0.002162
##  Mean   :-0.0030775   Mean   :-0.003911
##  3rd Qu.: 0.1552393   3rd Qu.: 0.164322
##  Max.   : 0.6484906   Max.   : 0.734665

plot(elist, cex = 0.7, las = 2)
```

**1. coordinate**
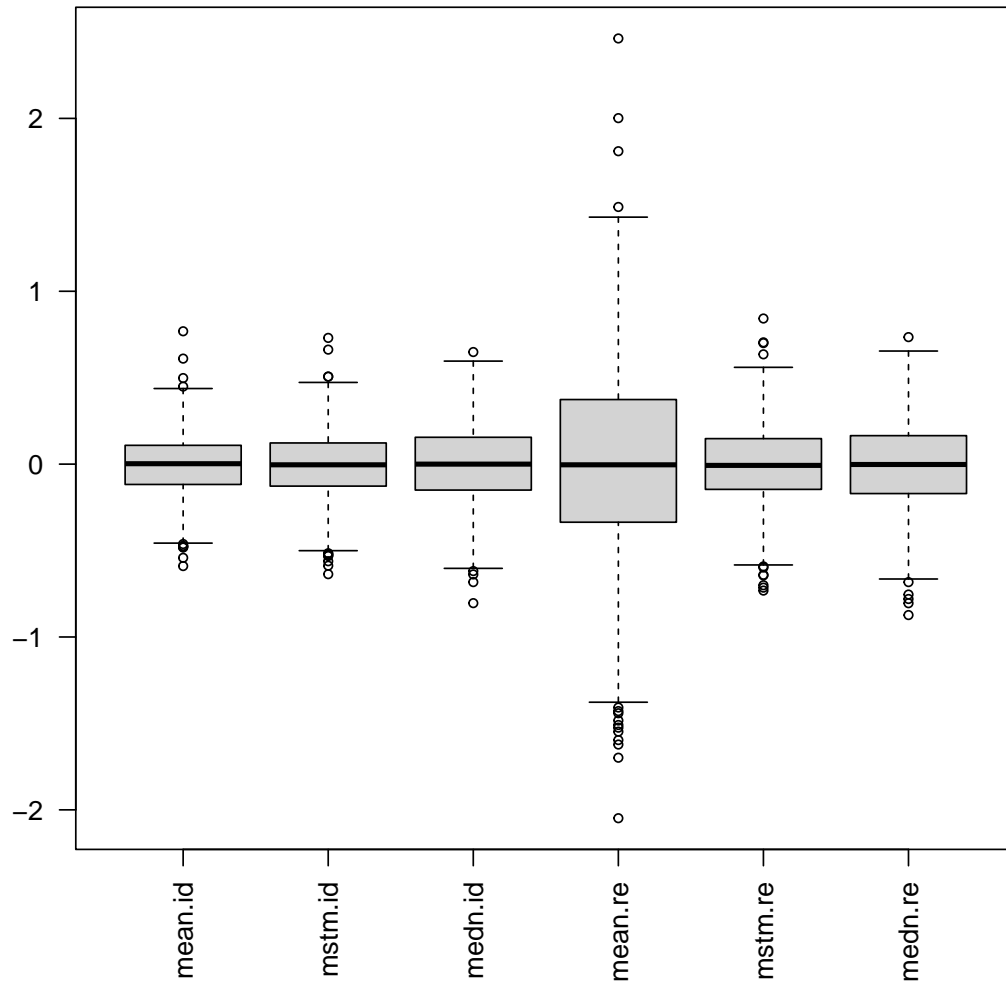


Output by **plot**/**show**-method for an object of class Evaluation

```
result.cont.mest

## An Evaluation Object
## name of Dataobject: object
## name of Datafile: contsim_01
## estimator: mestimator
## Result: 'data.frame': 1000 obs. of  2 variables:
##  $ mstm.id: num  -0.2308 -0.1202 0.0288 -0.0801 -0.0186 ...
```

```
##  $ mstm.re: num  -0.0839 -0.1553 0.1994 -0.0153 -0.0275 ...
```

Output by **summary**-method for an object of class EvaluationList

```
summary(elist)

## name of Evaluation List: a list of "Evaluation" objects
## name of Dataobject: object
## name of Datafile: contsim_01
## ---------------------------------
## name of Evaluation: object
## estimator: mean
## Result:
##     mean.id            mean.re
##  Min.   :-0.589662   Min.   :-2.048397
##  1st Qu.:-0.117384   1st Qu.:-0.335711
##  Median : 0.002348   Median :-0.003919
##  Mean   :-0.002694   Mean   : 0.009400
##  3rd Qu.: 0.108640   3rd Qu.: 0.372806
##  Max.   : 0.768604   Max.   : 2.462496
## ---------------------------------
## name of Evaluation: object
## estimator: mestimator
## Result:
##     mstm.id            mstm.re
##  Min.   :-0.636670   Min.   :-0.731594
##  1st Qu.:-0.127130   1st Qu.:-0.146029
##  Median :-0.003956   Median :-0.007322
##  Mean   :-0.003831   Mean   :-0.002743
##  3rd Qu.: 0.122138   3rd Qu.: 0.147472
##  Max.   : 0.730241   Max.   : 0.842396
## ---------------------------------
## name of Evaluation: object
## estimator: median
## Result:
##     medn.id            medn.re
##  Min.   :-0.8041779  Min.   :-0.873215
##  1st Qu.:-0.1501175  1st Qu.:-0.169997
##  Median :-0.0000101  Median :-0.002162
##  Mean   :-0.0030775  Mean   :-0.003911
##  3rd Qu.: 0.1552393  3rd Qu.: 0.164322
##  Max.   : 0.6484906  Max.   : 0.734665
```

In this example we present a standard robust simulation study that — in variations — arises in almost every paper on Robust Statistics. We do this with the tools provided by our package. . .

## 13.9  Expectation of a given function under a given distribution

This code is for illustration only; in the mean-time, the expectation- and variance operators implemented in this example have been included to package `"distrEx"` where their functionality has further been extended.

118

As in examples 13.5 and 13.6, we illustrate the use of package `"distr"` by implementing a general evaluation of expectation and variance under a given distribution.

```r
require("distrEx")
# Example
id <- function(x) x
sq <- function(x) x^2

# Expectation and Variance of Binom(6,0.5)
B <- Binom(6, 0.5)
E(B, id)

## [1] 3

E(B, sq) - E(B, id)^2

## [1] 1.5

# Expectation and Variance of Norm(1,1)
N <- Norm(1, 1)
E(N, id)

## [1] 0.9999998

E(N, sq) - E(N, id)^2

## [1] 0.9999944
```

## 13.10   $n$-fold convolution of absolutely continuous distributions

Might be useful for teaching the CLT: a straightforward implementation of the $n$–fold convolution of an arbitrary implemented absolutely continuous distribution — to show accuracy of our method we compare it to the exact formula valid for $n$-fold convolution of normal distributions.
From version 1.9 this is integrated to package `"distr"`.

```r
#######################################################
## Demo: n-fold convolution of absolutely continuous
##      probability distributions
#######################################################
require(distr)
```

```r
if(!isGeneric("convpow"))
    setGeneric("convpow",
    function(D1,...) standardGeneric("convpow"))

#########################################################
## Function for n-fold convolution
## -- absolute continuous distribution --
#########################################################

##implentation of Algorithm 3.4. of
# Kohl, M., Ruckdeschel, P., Stabla, T. (2005):
#   General purpose convolution algorithm for distributions
#   in S4-Classes by means of FFT.
# Technical report, Feb. 2005. Also available in
# http://www.uni-bayreuth.de/departments/math/org/mathe7/
#       /RUCKDESCHEL/pubs/comp.pdf


setMethod("convpow",
          signature(D1 = "AbscontDistribution"),
          function(D1, N){
            if((N < 1)||(!identical(floor(N), N)))
              stop("N has to be a natural greater than 0")

            m <- getdistrOption("DefaultNrFFTGridPointsExponent")

    ##STEP 1

            lower <- ifelse((q(D1)(0) > - Inf), q(D1)(0),
                    q(D1)(getdistrOption("TruncQuantile")))
            upper <- ifelse((q(D1)(1) < Inf), q(D1)(1),
                    q(D1)(getdistrOption("TruncQuantile"), lower.tail = FALSE))

    ##STEP 2

            M <- 2^m
            h <- (upper-lower)/M
            if(h > 0.01)
              warning(paste("Grid for approxfun too wide, ",
              "increase DefaultNrFFTGridPointsExponent", sep=""))
```

120

```r
        x <- seq(from = lower, to = upper, by = h)
        p1 <- p(D1)(x)

##STEP 3

        p1 <- p1[2:(M + 1)] - p1[1:M]

##STEP 4

        ## computation of DFT
        pn <- c(p1, numeric((N-1)*M))
        fftpn <- fft(pn)

##STEP 5

        ## convolution theorem for DFTs
        pn <- Re(fft(fftpn^N, inverse = TRUE)) / (N*M)
        pn <- (abs(pn) >= .Machine$double.eps)*pn
        i.max <- N*M-(N-2)
        pn <- c(0,pn[1:i.max])
        dn <- pn / h
        pn <- cumsum(pn)

##STEP 6(density)

        ## density
        x <- c(N*lower,seq(from = N*lower+N/2*h,
              to = N*upper-N/2*h, by=h),N*upper)
        dnfun1 <- approxfun(x = x, y = dn, yleft = 0, yright = 0)

##STEP 7(density)

        standardizer <- sum(dn[2:i.max]) + (dn[1]+dn[i.max+1]) / 2
        dnfun2 <- function(x) dnfun1(x) / standardizer

##STEP 6(cdf)

        ## cdf with continuity correction h/2
        pnfun1 <- approxfun(x = x+0.5*h, y = pn,
                    yleft = 0, yright = pn[i.max+1])
```

```r
    ##STEP 7(cdf)

            pnfun2 <- function(x) pnfun1(x) / pn[i.max+1]


            ## quantile with continuity correction h/2
            yleft <- ifelse(((q(D1)(0) == -Inf)|
                             (q(D1)(0) == -Inf)),
                            -Inf, N*lower)
            yright <- ifelse(((q(D1)(1) == Inf)|
                              (q(D1)(1) == Inf)),
                             Inf, N*upper)
            w0 <- options("warn")
            options(warn = -1)
            qnfun1 <- approxfun(x = pnfun2(x+0.5*h),
                        y = x+0.5*h, yleft = yleft, yright = yright)
            qnfun2 <- function(x){
            ind1 <- (x == 0)*(1:length(x))
            ind2 <- (x == 1)*(1:length(x))
            y <- qnfun1(x)
            y <- replace(y, ind1[ind1 != 0], yleft)
            y <- replace(y, ind2[ind2 != 0], yright)
            return(y)
            }
            options(w0)

            rnew = function(N) apply(matrix(r(e1)(n*N),
                                     ncol=N), 1, sum)

            return(new("AbscontDistribution", r = rnew,
                       d = dnfun1, p = pnfun2, q = qnfun2))
})


## initialize a normal distribution
A <- Norm(mean=0, sd=1)

## convolution power
N <- 10
```

```r
## convolution via FFT
AN <- convpow(as(A,"AbscontDistribution"), N)
##  ... for the normal distribution , 'convpow' has an "exact"
##      method by version 1.9 so the as(.,.)  is needed to
##      see how the algorithm above works

## convolution exact
AN1 <- Norm(mean=0, sd=sqrt(N))

## plots of the results
eps <- getdistrOption("TruncQuantile")
par(mfrow=c(1,3))
low <- q(AN1)(eps)
upp <- q(AN1)(eps, lower.tail = FALSE)
x <- seq(from = low, to = upp, length = 10000)

## densities
plot(x, d(AN1)(x), type = "l", lwd = 5)
lines(x , d(AN)(x), col = "orange", lwd = 1)
title("Densities")
legend("topleft", legend=c("exact", "FFT"),
        fill=c("black", "orange"))

## cdfs
plot(x, p(AN1)(x), type = "l", lwd = 5)
lines(x , p(AN)(x), col = "orange", lwd = 1)
title("CDFs")
legend("topleft", legend=c("exact", "FFT"),
        fill=c("black", "orange"))

## quantile functions
x <- seq(from = eps, to = 1-eps, length = 1000)
plot(x, q(AN1)(x), type = "l", lwd = 5)
lines(x , q(AN)(x), col = "orange", lwd = 1)
title("Quantile functions")
legend("topleft",
        legend = c("exact", "FFT"),
        fill = c("black", "orange"))
```
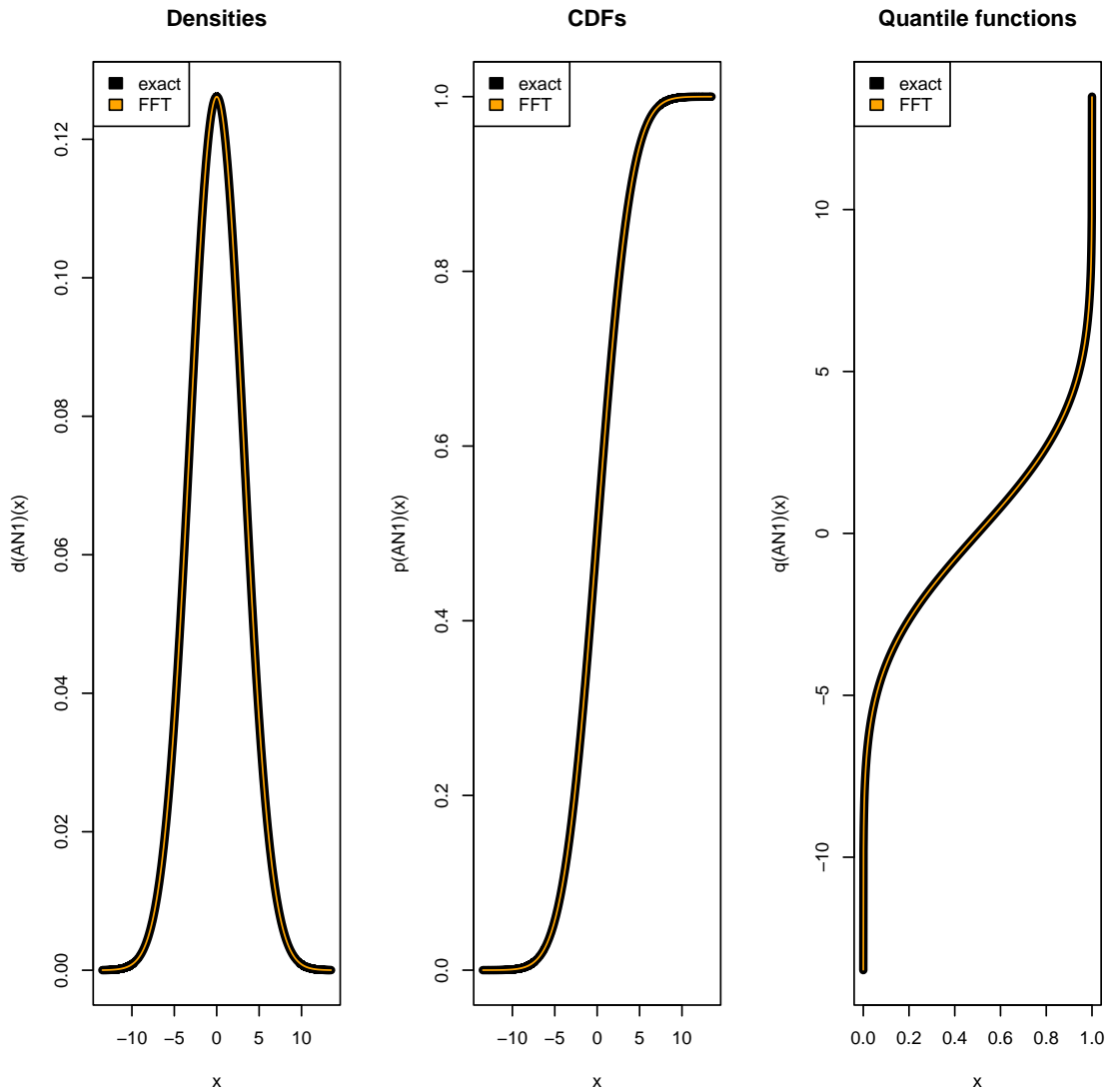
**Densities**   **CDFs**   **Quantile functions**



# References

[1] Bengtsson H. (2003): The R.oo package - object-oriented programming with references using standard R code. In: Hornik K., Leisch F. and Zeileis A. (Eds.) *Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003)*. Vienna, Austria. Published as http://www.ci.tuwien.ac.at/Conferences/DSC-2003/ 9

[2] Chambers J.M. (1998): *Programming with data. A guide to the S language*. Springer. https://statweb.stanford.edu/ jmc4/Sbook/ 9

[3] Gentleman R. (2003): *Object Orientated Programming. Slides of a Short Course held in Auckland*. http://www.stat.auckland.ac.nz/S-Workshop/Gentleman/Methods.pdf 92, 93

[4] Kohl M. (2005): *Numerical Contributions to the Asymptotic Theory of Robustness*. Dissertation, Universität Bayreuth. See also http://stamats.de/ThesisMKohl.pdf 73, 75

[5] Ruckdeschel P. and Kohl, M. (2014): General Purpose Convolution Algorithm for Distributions in S4-Classes by means of FFT. *J. Statist. Software*, **59**(4): 1–25. 9, 32, 92

[6] Press W.H., Teukolsky S.A., Vetterling W.T. and Flannery B.P. (1992): *Numerical recipes in C. The art of scientific computing*. Cambridge Univ. Press, 2. Aufl. 65

[7] Rice J.A. (1988): *Mathematical statistics and data analysis*. The Wadsworth & Brooks/-Cole Statistics/Probability Series. Wadsworth & Brooks/Cole Advanced Books & Software, Pacific Grove, California. 94

[8] Ruckdeschel P., Kohl M., Stabla T., and Camphausen F. (2006): S4 Classes for Distributions. *R-News*, **6**(2): 10–13. https://CRAN.R-project.org/doc/Rnews/Rnews_2006-2.pdf 4