S4 Classes for Distributions—a manual for packages "distr", "distrSim", "distrTEst", version 1.8, "distrEx", version 0.4-4

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

Mathematisches Institut
Universität Bayreuth
D-95440 Bayreuth
Germany

e-Mail: peter.ruckdeschel@uni-bayreuth.de

November 23, 2006

Abstract

"distr" is a package for R from version 1.8.1 onwards that is distributed under GPL license 2.0. Its own current version is 1.8. The aim of this package is to provide a conceptual treatment of random variables (r.v.'s) by means of S4–classes. A mother 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 convolution of independent r.v.'s. From version 1.6 on, "distr" has been split up into the smaller packages "distr" (only distribution-classes and -methods), "distrSim" (standardized treatment of simulations, also under contaminations) and "distrTEst"
(classes and methods for evaluations of statistical procedures on such simulations).
The latter two of them require package "setRNG" by Paul Gilbert to be installed from CRAN.

*Universität Bayreuth
†SIRS-Lab GmbH, Jena
‡Universität Siegen ????? was soll ich hier machen ?????
§Universität Bayreuth

Additionally, mainly contributed by [4], in "distrEx" we extend the functionality of "distr", providing functionals like expectation or variance and distances for distributions. Also, this package contains some first steps to multivariate distributions, providing classes for discrete multivariate distributions and for factorized, conditional distributions.

Contents

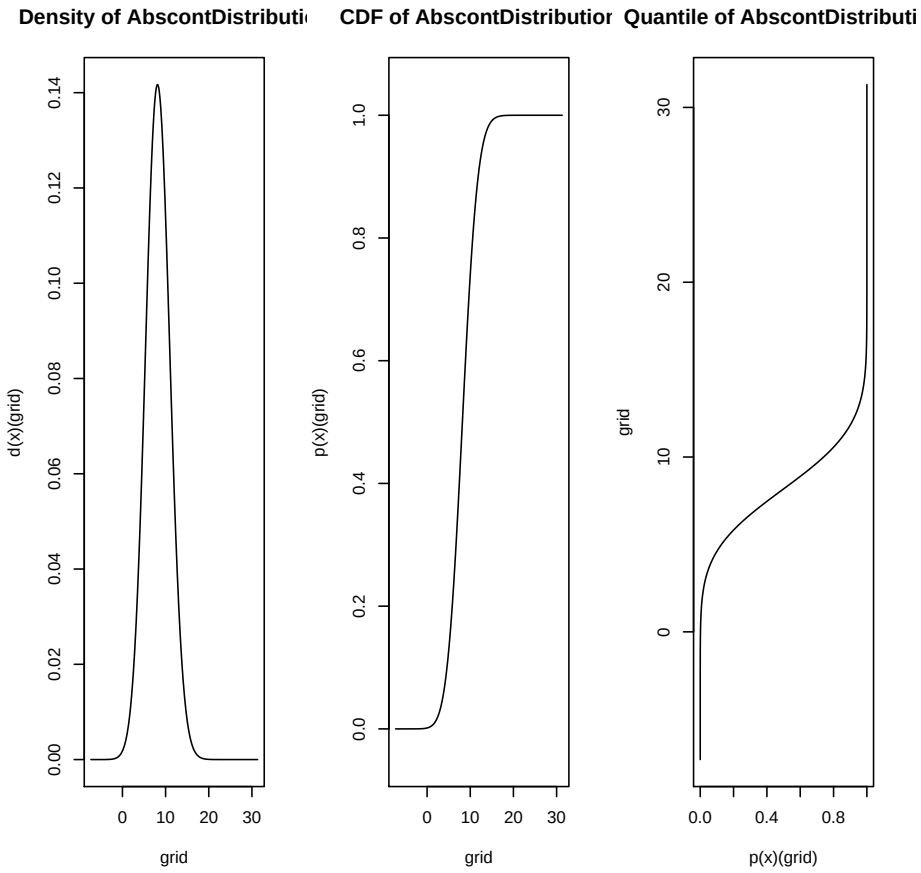
This document appeared in an abridged form in *R-News*, **6**(2) as "S4 Classes for Distributions", c.f. [8], which in its published form refers to package versions 1.6, resp. 0.4-2. This document takes into account the subsequent revisions and versions.

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 implementation. 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 $f\colon \mathbb{R}^2 \to \mathbb{R}$ which is already realized as a function in R.
This is possible with package `"distr"`. As an example, try

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

Distribution Object of Class: AbscontDistribution

> plot(Z)
> p(Z)(0.4)

[1] 0.002415384

> q(Z)(0.3)

[1] 6.70507

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

 [1]  7.530445 10.234964  5.722258  6.182583 14.103522  9.944960  9.980536
 [8]  6.771546  8.238850  9.947272  9.563634  7.654897  7.126974  9.242383
[15]  5.089696 12.339637 11.014562 13.604341  9.889180  5.543226  9.107229
```

```
[22]  6.450713 12.210669 11.416910  6.478102  9.050404  8.144519  8.922142
[29]  9.582026  8.250282  6.243961  7.143411  4.935602  7.580706  7.706190
[36]  6.398999 12.762741  7.569283  8.179069  3.820573  4.952510  8.185683
[43]  4.907038  4.071673  3.707401  4.667932  3.464704  2.616173  2.151091
[50]  9.930184
```



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`, $\mathcal{L}(\texttt{Z}) = \mathcal{L}(2 * \texttt{N} + 3 + \texttt{P})$, 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

[1] a solution to this "homework" may be found in the sources to `"distrEx"`

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

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

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 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 even more plan to introduce a further extension package to `"distr"` which will cover statistical models. In this package, this issue will be solved by requiring a map $\theta \mapsto P_\theta$ or, in `S`, a function `function(theta)....` which returns an object of class distribution or subclass, which realizes P_θ. So it will be up to the programmer or user how to realize this map.

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`.

2.1.1 Subclasses

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

[4] Upto version 0.4-4, we use a different mechanism to inspect/modify global options of `"distrEx"` (see section 4.2); corresponding functions `distrExoptions`, `getdistrExOption` for package `"distrEx"` will only be available from version 0.4-5 on, which is due for spring 2007.

latter 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 4.
As subclasses of these two 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. Operations to automatically generate new slots `r`, `d`, `p`, and `q` —induced by mathematical transformations— perhaps provide the most powerful use of our package. This is discussed in some detail in subsection 3.

2.1.2 Classes for multivariate distributions and for conditional distributions

In `"distrEx"`, we provide the following classes for handling multivariate 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.

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`.

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`.

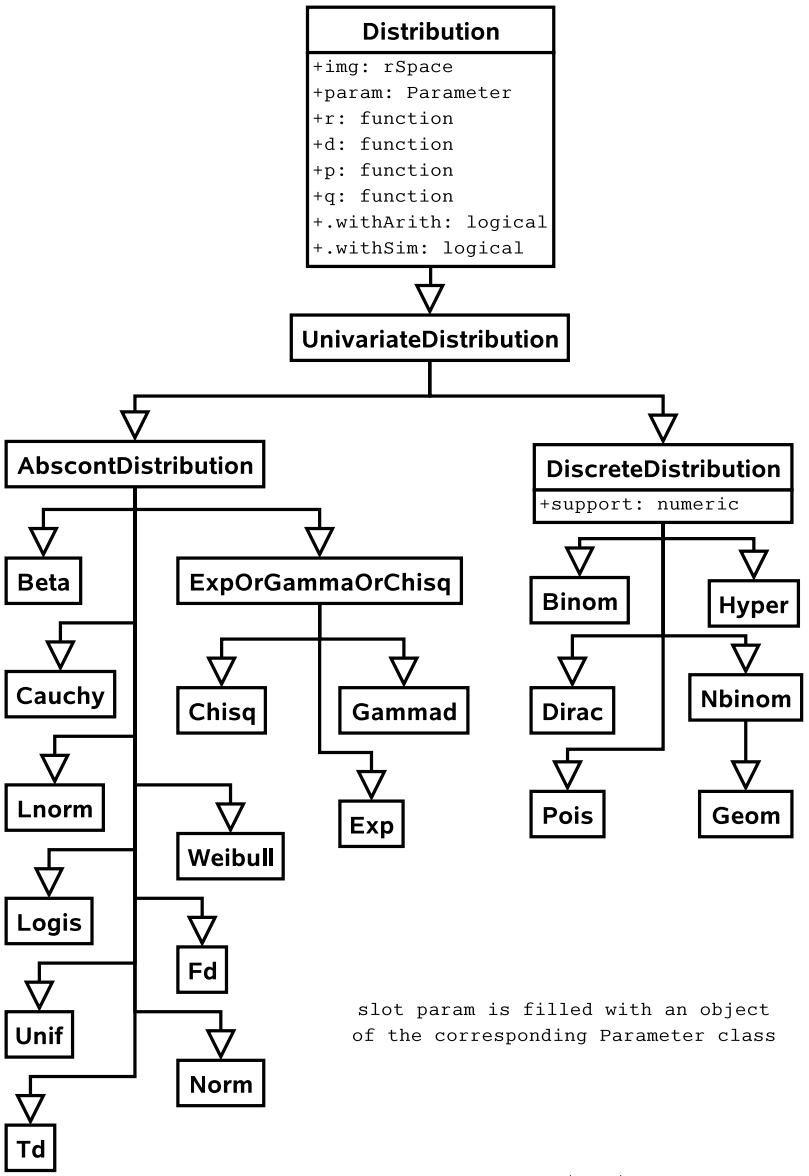


Figure 1: Inheritance relations and slots of the corresponding (sub-)classes for `Distribution` where we do not repeat inherited slots

2.1.3 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

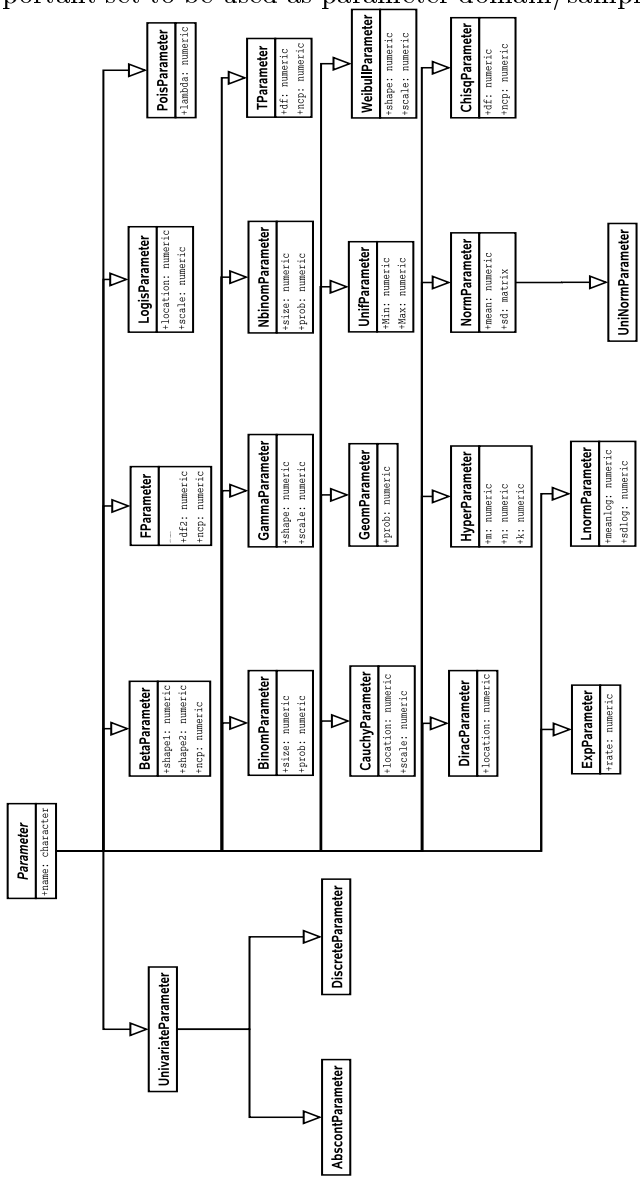


Figure 2: Inheritance relations and slots of the corresponding (sub-)classes for `Parameter`

be an Euclidean space. So `rSpace` and `EuclideanSpace` are also implemented as classes,

the structure of which may be read off in figure 3.

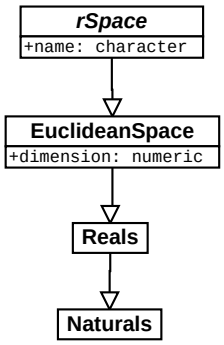


Figure 3: Inheritance relations and slots of the corresponding (sub-)classes for `rSpace`

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 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 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

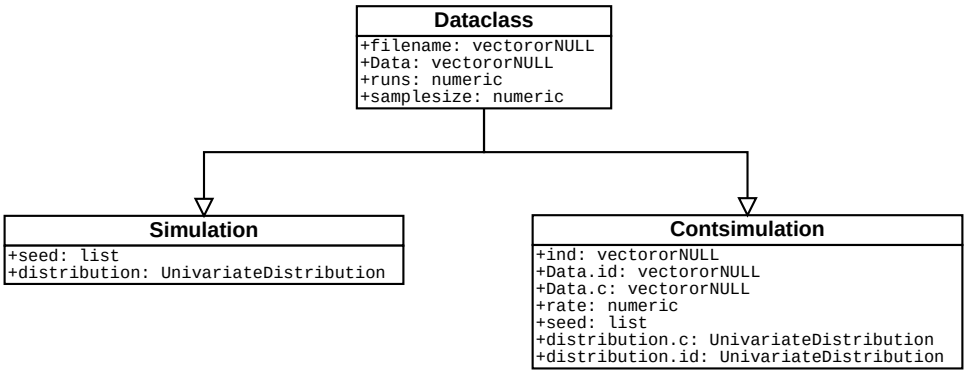


Figure 4: Inheritance relations and slots of the corresponding (sub-)classes for `Dataclass`

`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 3.7.
Schematically, the slots of this class may be read off in figure 5. A corresponding `savedata` 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

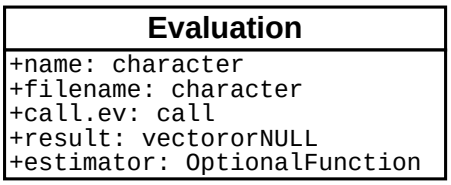


Figure 5: Slots of class `Evaluation`

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

HIER KOMMT DAS BILD

Figure 6: Slots of class `Evaluation`

`EvaluationList` object may be accessed by the accessor method `Data`.

3 Methods

We have made available quite general arithmetical operations to our distribution objects, generating new image distributions automatically.

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

This behaviour will soon be annoying so you may switch it off setting the global option `WarningArith` to `FALSE` (see section 4).

3.1 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.

3.2 The group `math` of unary mathematical operations

Also the group `math` of unary mathematical operations is available for distribution classes; so expressions like `exp(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(sin(Norm()) + 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] 0.3808551

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

[1] 0.3794801

> sin(A1 + A2)

Distribution Object of Class: AbscontDistribution
```

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 4).

3.3 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: By means of the function `RtoDPQ` we first generate $10^{\texttt{RtoDPQ.e}}$ random numbers where `RtoDPQ.e` is a global option of this package and is discussed in section 4. 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. 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.

3.4 Convolution

A convolution method for two independent r.v.'s is implemented by means of explicit calculations for discrete summands, and by means of FFT[5] if one of the summands is absolutely continuous. 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.

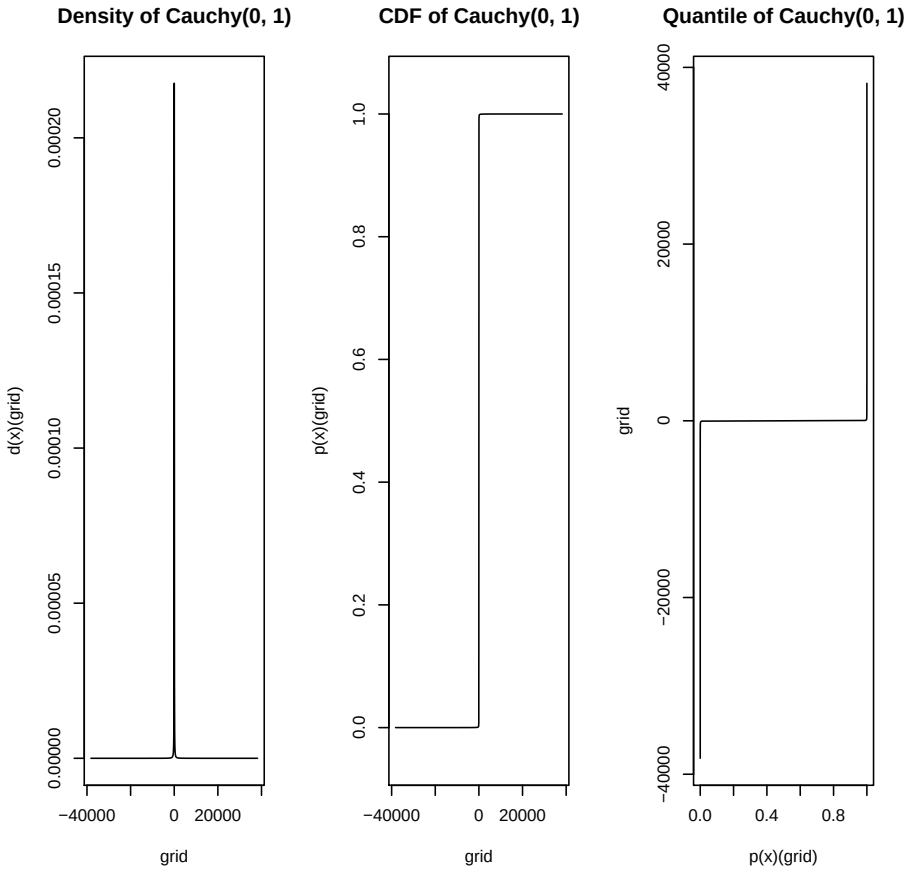
[5] Details to be found in [5]

3.5 Overloaded generic functions

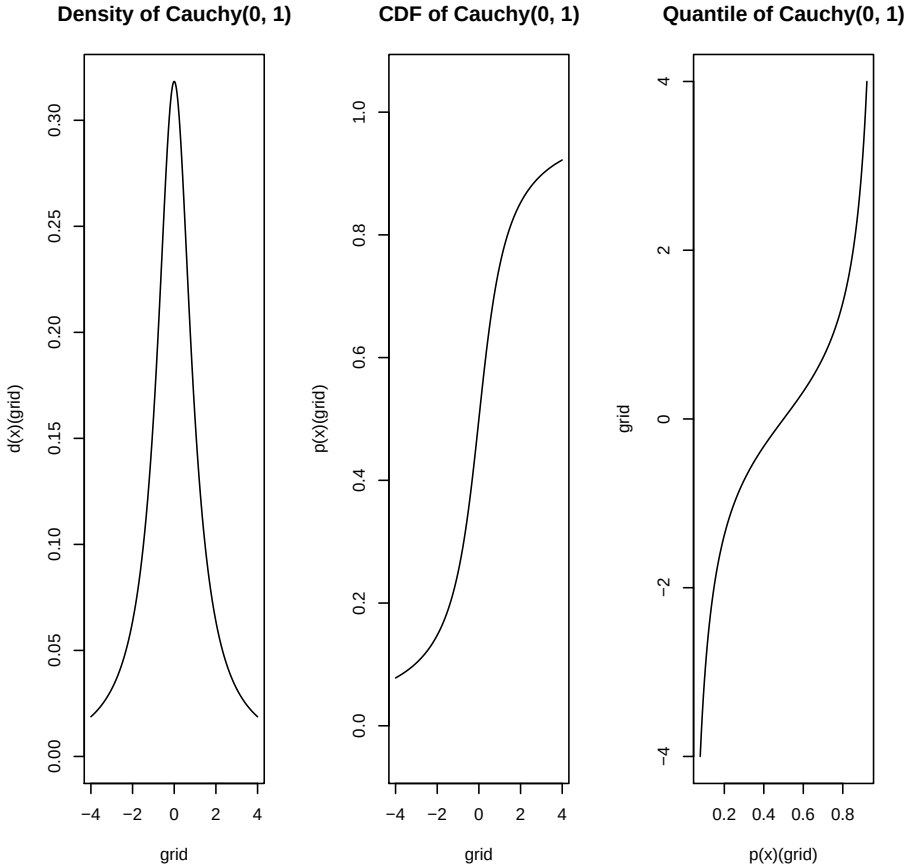
Methods `print`, `plot`, `show` and `summary` have been overloaded for classes `Distribution`, `Dataclass`, `Simulation`, `ContSimulation`, as well as `Evaluation` and `EvaluationList` to produce "pretty" output. `print`, `plot`, `show` and `summary` have additional, optional arguments for plotting subsets 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"`.

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. More important, you may also override the automatically chosen x-region by passing an `xlim` argument:

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



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



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")`.

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.

3.6 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

```
> have.distrSim <- suppressWarnings(require("distrSim"))
> if (have.distrSim) {
+     X <- Simulation()
+     seed(X) <- setRNG()
+     simulate(X)
+ } else {
+     cat("\n functionality not (yet) available; ")
+     cat("you have to install package \"distrSim\" first.\n")
+ }
```

3.7 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 11.8. For data of class Contsimulation, the result takes a slightly different, combining evalations on ideal and real data.

3.8 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`

3.9 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.

3.10 Functionals (in package `"distrEx"`)

3.10.1 Expectation

The most important contribution of package `"distrEx"` is a general expectation operator. In basic statistic courses, the expectation E may come as $\mathrm{E}[X]$, $\mathrm{E}[f(X)]$, $\mathrm{E}[X|Y=y]$, or $\mathrm{E}[f(X)|Y=y]$. Our operator (or in S4-language "generic function") `E` covers all of these situtations (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[6]; 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.

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

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 `Beta` (for noncentrality 0), `Binom`, `Cauchy`, `Chisq`,`Dirac`, `Exp`, `Fd`, `Gammad`, `Geom`, `Hyper`, `Logis`, `Lnorm`, `Nbinom`, `Norm`, `Pois`, `Td`, `Unif`, `Weibull`.

examples

```
> have.distrEx <- suppressWarnings(require("distrEx"))
> if (have.distrEx) {
+     D4 <- LMCondDistribution(theta = 1)
+     D4
+     N <- Norm(mean = 2)
+     E(D4, cond = 1)
+     E(D4, cond = 1, useApply = FALSE)
+     E(as(D4, "UnivariateCondDistribution"), cond = 1)
+     E(as(D4, "UnivariateCondDistribution"), cond = 1, useApply = FALSE)
+     E(D4, function(x) {
+         x^2
+     }, cond = 2)
+     E(D4, function(x) {
+         x^2
+     }, cond = 2, useApply = FALSE)
+     E(N, function(x) {
+         x^2
+     })
+     E(as(N, "UnivariateDistribution"), function(x) {
+         x^2
+     }, useApply = FALSE)
+     E(D4, function(x, cond) {
+         cond * x^2
+     }, cond = 2, withCond = TRUE)
+     E(D4, function(x, cond) {
+         cond * x^2
+     }, cond = 2, withCond = TRUE, useApply = FALSE)
```

```
+     E(N, function(x) {
+         2 * x^2
+     })
+     E(as(N, "UnivariateDistribution"), function(x) {
+         2 * x^2
+     }, useApply = FALSE)
+ } else {
+     cat("\n functionality not (yet) available; ")
+     cat("you have to install package \"distrEx\" first.\n")
+ }

[1] 10.00136
```

3.10.2 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`.

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

3.10.3 Further functionals

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

- standard deviation: `sd`
- median: `median` (not for function/condition arguments)
- median of absolute deviations: `mad` (not for function/condition arguments)
- interquartile range: `IQR` (not for function/condition arguments)

3.11 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`

3.12 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
- total variation distance
- Hellinger distance
- convex-contamination "distance" (asymmetric!) defined as

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

3.13 Functions for demos (in package "distrEx")

To illustrate the possibilities with packages "distr" and "distrEx" we include two major demos, each with extra code to it

3.13.1 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: illustCLT; 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.

3.13.2 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^ε, 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 Options

4.1 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

- `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`

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.

4.2 Options for `"distrEx"`

For the moment we use 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`. Currently, the following options are available:

- `MCIterations`: number of Monte-Carlo iterations used for crude Monte-Carlo integration.
- `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.
- `GLIntegrateOrder`: The order used for the Gauß-Legendre integration inside of `distrExIntegrate`.
- `ElowerTruncQuantile`: The lower limit of integration used inside of `E` which corresponds to the `ElowerTruncQuantile`-quantile.
- `EupperTruncQuantile`: The upper limit of integration used inside of `E` which corresponds to the `(1-ElowerTruncQuantile)`-quantile.
- `ErelativeTolerance`: The relative tolerance used inside of `E` when calling `distrExIntegrate`.
- `m1dfLowerTruncQuantile`: The lower limit of integration used inside of `m1df` which corresponds to the `m1dfLowerTruncQuantile`-quantile.
- `m1dfRelativeTolerance`: The relative tolerance used inside of `m1df` when calling `distrExIntegrate`.

- `m2dfLowerTruncQuantile`: The lower limit of integration used inside of `m2df` which corresponds to the `m2dfLowerTruncQuantile`-quantile.
- `m2dfRelativeTolerance`: The relative tolerance used inside of `m2df` when calling `distrExIntegrate`.

We are planning to switch to `distroptions`/`getdistrOption`-like commands in the next release of this package.

4.3 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.

4.4 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

5 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>).

6 System/version requirements, license, etc.

6.1 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.

6.2 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.

6.3 Dependencies

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". also available on CRAN.

6.4 License

This software is distributed under the terms of the GNU GENERAL PUBLIC LICENSE Version 2, June 1991, confer

http://www.gnu.org/copyleft/gpl.html

7 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 χ^2 distribution; exact formulae for scale transformations for the Exp-/Gamma-distribution
- 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.
- 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 11.7.

8 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`.

9 Odds and Ends

9.1 What should be done and what we could do —for version >1.8

- application of FFT to any univariate distributions —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 `"*"`
- derivation of a class `LatticeDistribution` from `DiscreteDistribution` to be able to easily apply FFT
- redo the initialize- and the math-method for discrete distributions when only slot `r` is given
- generating function for new distribution classes to ease inclusion of new distributions
- goodness of fit tests for distribution-objects
- use of `\S4method` in documentation
- overloading binary operators of group `Math2` for independent distributions
- defining a subgroup of `Math2` of invertible binary operators
- better use of `concept` in rd-files

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

- multivariate distributions
- conditional distributions
- copula

10 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`.

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.

11 Examples

11.1 12-fold convolution of uniform $(0, 1)$ variables

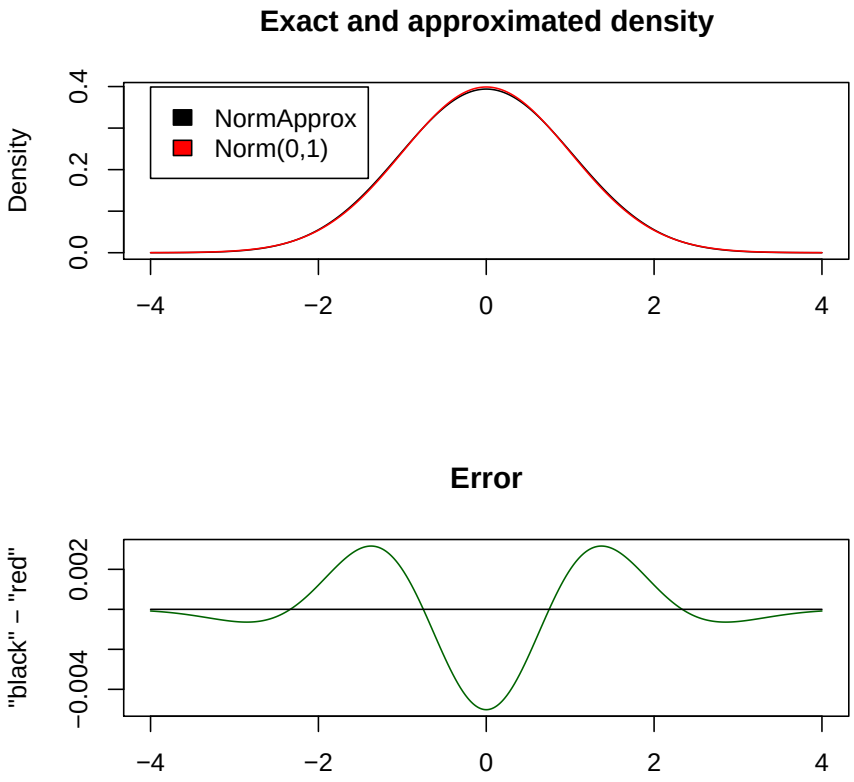
Code also available under

http://www.uni-bayreuth.de/departments/math/org/mathe7/DISTR/NormApprox.R

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)

[1] TRUE

> 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()
```

```
> 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(-4, d(N)(0), 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))
> par(opar)
```



11.2 Comparison of exact convolution to FFT for normal distributions

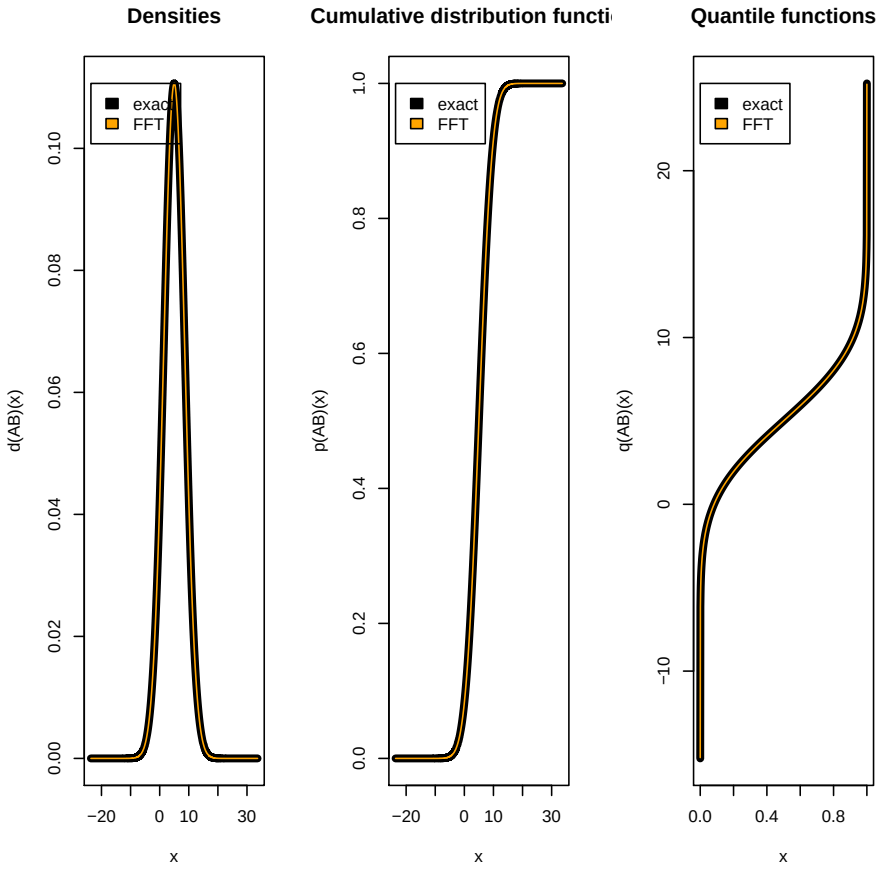
Code also available under

http://www.uni-bayreuth.de/departments/math/org/mathe7/DISTR/ConvolutionNormalDistr.R

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

```
> require(distr)

[1] TRUE

> A <- Norm(mean = 1, sd = 2)
> B <- Norm(mean = 4, sd = 3)
> AB <- A + B
> A1 <- as(A, "AbscontDistribution")
> B1 <- as(B, "AbscontDistribution")
> oldeps <- getdistrOption("TruncQuantile")
> eps <- 1e-08
> distroptions(TruncQuantile = eps)
> AB1 <- A1 + B1
> par(mfrow = c(1, 3))
> low <- q(AB)(1e-15)
> upp <- q(AB)(1 - 1e-15)
> x <- seq(from = low, to = upp, length = 10000)
> plot(x, d(AB)(x), type = "l", lwd = 5)
> lines(x, d(AB1)(x), col = "orange", lwd = 1)
> title("Densities")
> legend(low, d(AB)(5), legend = c("exact", "FFT"), fill = c("black",
+     "orange"))
> plot(x, p(AB)(x), type = "l", lwd = 5)
> lines(x, p(AB1)(x), col = "orange", lwd = 1)
> title("Cumulative distribution functions")
> legend(low, 1, legend = c("exact", "FFT"), fill = c("black",
+     "orange"))
> 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(0, q(AB)(1 - eps), legend = c("exact", "FFT"), fill = c("black",
+     "orange"))
> 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-08,
+     N1 = AB, N2 = AB1)
> cat("Total variation distance of densities:\t")
```

```
Total variation distance of densities:

> print(dv)

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

> z <- r(Unif(Min = low, Max = upp))(1e+05)
> dk <- max(abs(p(AB)(z) - p(AB1)(z)))
> cat("Kolmogorov distance of cdfs:\t", dk, "\n")

Kolmogorov distance of cdfs:         2.028470e-07

> distroptions(TruncQuantile = oldeps)
```



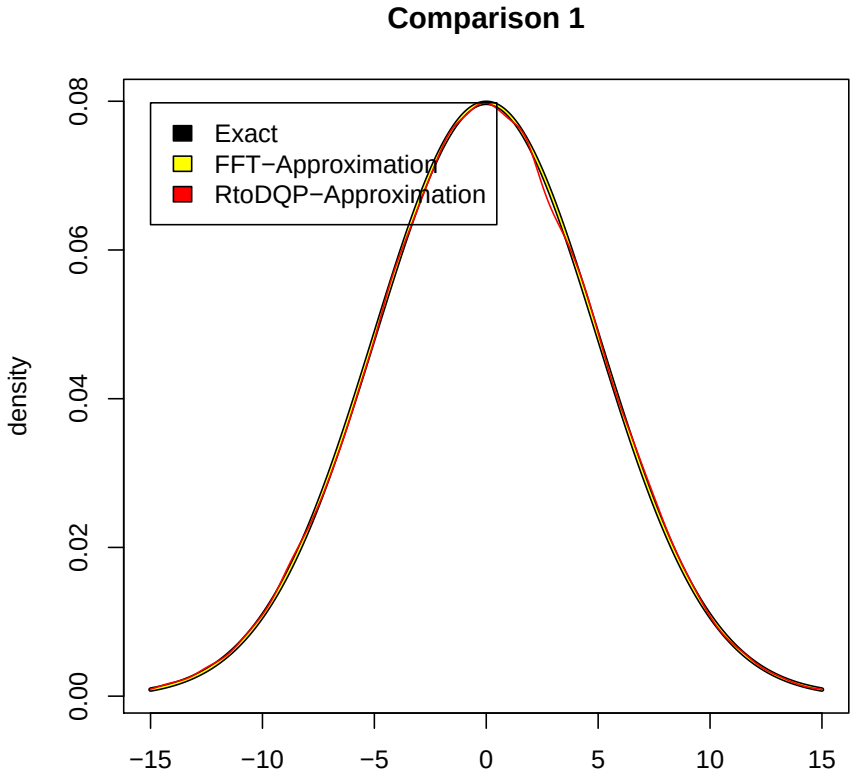
11.3 Comparison of FFT to RtoDPQ

Code also available under

http://www.uni-bayreuth.de/departments/math/org/
/mathe7/DISTR/ComparisonFFTandRtoDPQ.R

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)

[1] TRUE

> N1 <- Norm(0, 3)
> N2 <- Norm(0, 4)
> rnew1 <- function(n) r(N1)(n) + r(N2)(n)
> X <- N1 + N2
> Y <- N1 + as(N2, "AbscontDistribution")
> Z <- new("AbscontDistribution", r = rnew1)
> 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(-15, d(X)(0), legend = c("Exact", "FFT-Approximation",
+     "RtoDQP-Approximation"), fill = c("black", "yellow", "red"))
> B <- Binom(size = 6, prob = 0.5) * 10
> N <- Norm()
> rnew2 <- function(n) r(B)(n) + r(N)(n)
> Y <- B + N
> Z <- new("AbscontDistribution", r = rnew2)
> 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(-5, d(Y)(30), legend = c("Exact", "RtoDQP-Approximation"),
+     fill = c("black", "red"))
```



11.4 Comparison of exact and approximate stationary regressor distribution

Code also available under

http://www.uni-bayreuth.de/departments/math/org/
/mathe7/DISTR/StationaryRegressorDistr.R

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.

```
> require(distr)
```

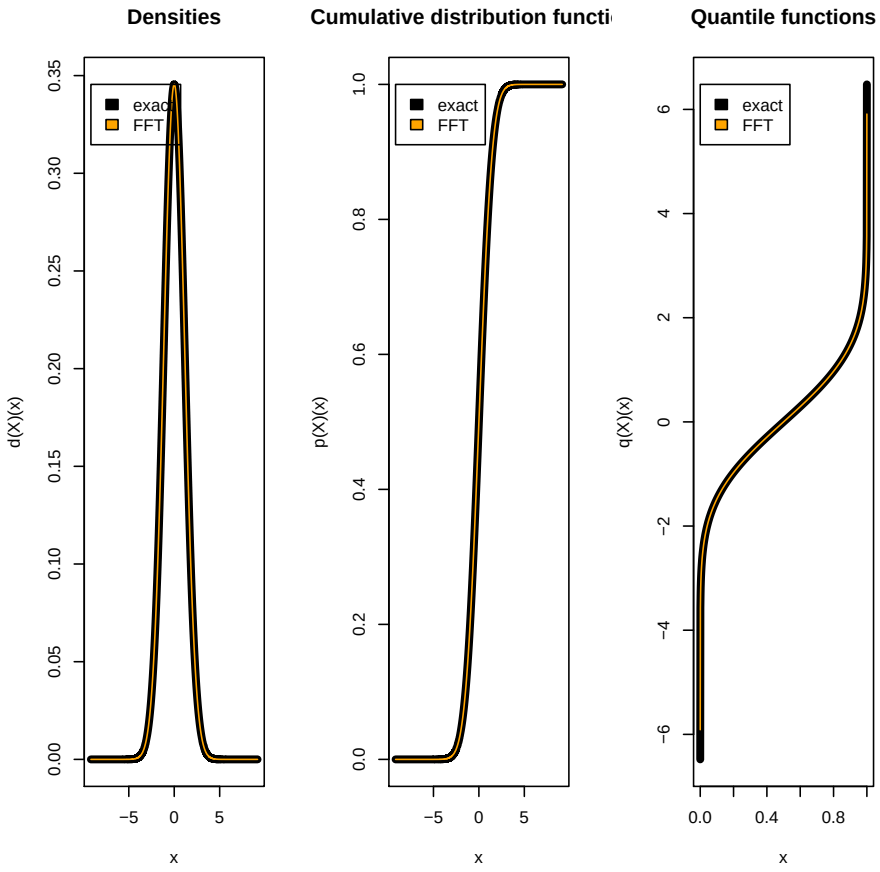
```
[1] TRUE
```

```
> phi <- 0.5
> V <- as(Norm(), "AbscontDistribution")
> oldeps <- getdistrOption("TruncQuantile")
> eps <- 1e-08
> distroptions(TruncQuantile = eps)
> H <- V
> n <- 15
> for (i in 1:n) {
+     Vi <- phi^i * V
+     H <- H + Vi
+ }
> X <- Norm(sd = sqrt(1/(1 - phi^2)))
> par(mfrow = c(1, 3))
> low <- q(X)(1e-15)
> upp <- q(X)(1 - 1e-15)
> x <- seq(from = low, to = upp, length = 10000)
> plot(x, d(X)(x), type = "l", lwd = 5)
> lines(x, d(H)(x), col = "orange", lwd = 1)
> title("Densities")
> legend(low, d(X)(0), legend = c("exact", "FFT"), fill = c("black",
+     "orange"))
> plot(x, p(X)(x), type = "l", lwd = 5)
> lines(x, p(H)(x), col = "orange", lwd = 1)
> title("Cumulative distribution functions")
> legend(low, 1, legend = c("exact", "FFT"), fill = c("black",
+     "orange"))
> 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(0, q(X)(1 - eps), legend = c("exact", "FFT"), fill = c("black",
+     "orange"))
> 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-05,
+     N1 = X, N2 = H)
> cat("Total variation distance of densities:\t")
```

```
Total variation distance of densities:
```

```
> print(dv)
```

```
9.100439e-06 with absolute error < 6.4e-06

> z <- r(Unif(Min = low, Max = upp))(1e+05)
> dk <- max(abs(p(X)(z) - p(H)(z)))
> cat("Kolmogorov distance of cdfs:\t", dk, "\n")

Kolmogorov distance of cdfs:         4.30419e-06

> distroptions(TruncQuantile = oldeps)
```



11.5 Truncation and Huberization/winsorization

Code also available under

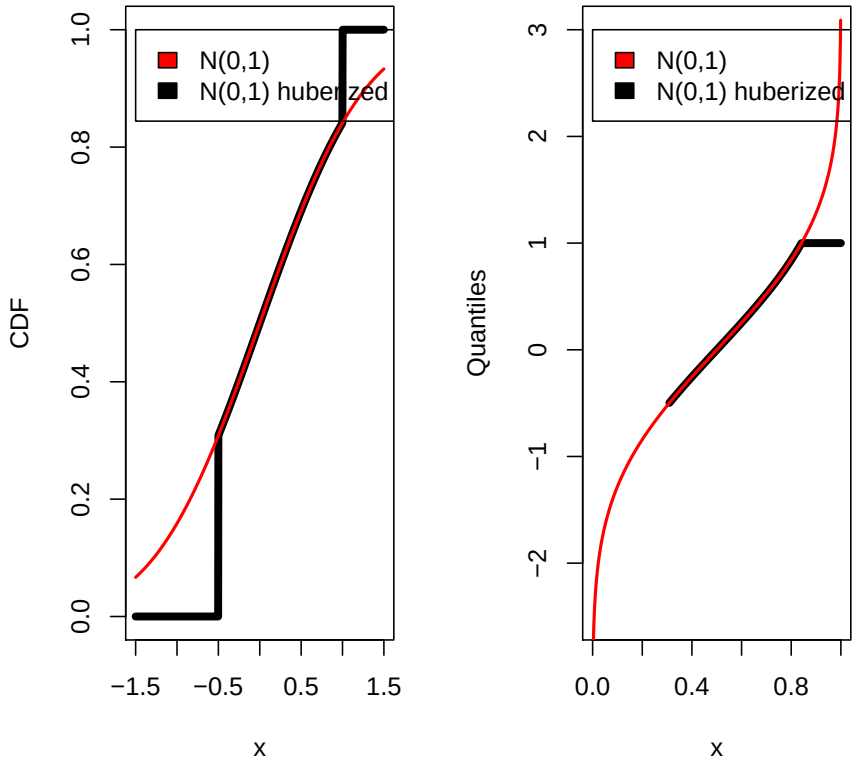
http://www.uni-bayreuth.de/departments/math/org/
/mathe7/DISTR/huberize.R

http://www.uni-bayreuth.de/departments/math/org/
/mathe7/DISTR/truncate.R

The operations of truncation and Huberization play a crucial role in Robust Statistics, but also arise in many other contexts like censoring etc; they may now be formulated quite generally as shown in this example. With the slots `d`, `p` and `q` of class `UnivariateDistribution` being `OptionalFunction` from version `1.4` on, it would be no problem to return a corresponding distribution object now.

```
> require(distr)

[1] TRUE

> if (!isGeneric("Huberize")) setGeneric("Huberize", function(object,
+     lower, upper) standardGeneric("Huberize"))

[1] "Huberize"

> setMethod("Huberize", signature(object = "AbscontDistribution",
+     lower = "numeric", upper = "numeric"), function(object, lower,
+     upper) {
+     rnew = function(n) {
+         rn = r(object)(n)
+         ifelse(rn < lower, lower, ifelse(rn >= upper, upper,
+             rn))
+     }
+     pnew = function(x) ifelse(x < lower, 0, ifelse(x >= upper,
+         1, p(object)(x)))
+     plower = p(object)(lower)
+     pupper = p(object)(upper)
+     qnew = function(x) ifelse(x < plower, ifelse(x < 0, NA, -Inf),
+         ifelse(x >= pupper, ifelse(x > 1, NA, upper), q(object)(x)))
+     new("UnivariateDistribution", r = rnew, p = pnew, q = qnew,
+         d = NULL)
+ })

[1] "Huberize"

> N = Norm()
> HN = Huberize(N, -0.5, 1)
> r(HN)(10)

 [1]  1.0000000 -0.5000000  0.2154488  1.0000000  0.6802175 -0.5000000
 [7] -0.5000000  1.0000000  0.2453647  0.5290800
```

```
> oldpar = par()
> par(mfrow = c(1, 2))
> x = seq(-1.5, 1.5, length = 1000)
> plot(x, p(HN)(x), type = "l", lwd = 5, ylab = "CDF")
> lines(x, p(N)(x), lwd = 2, col = "red")
> legend(-1.5, 1, legend = c("N(0,1)", "N(0,1) huberized"), fill = c("red",
+     "black"))
> x = seq(0, 1, length = 1000)
> plot(x, q(HN)(x), type = "l", lwd = 5, ylab = "Quantiles", ylim = c(-2.5,
+     3))
> lines(x, q(N)(x), lwd = 2, col = "red")
> legend(0, 3, legend = c("N(0,1)", "N(0,1) huberized"), fill = c("red",
+     "black"))
> par(oldpar)
```



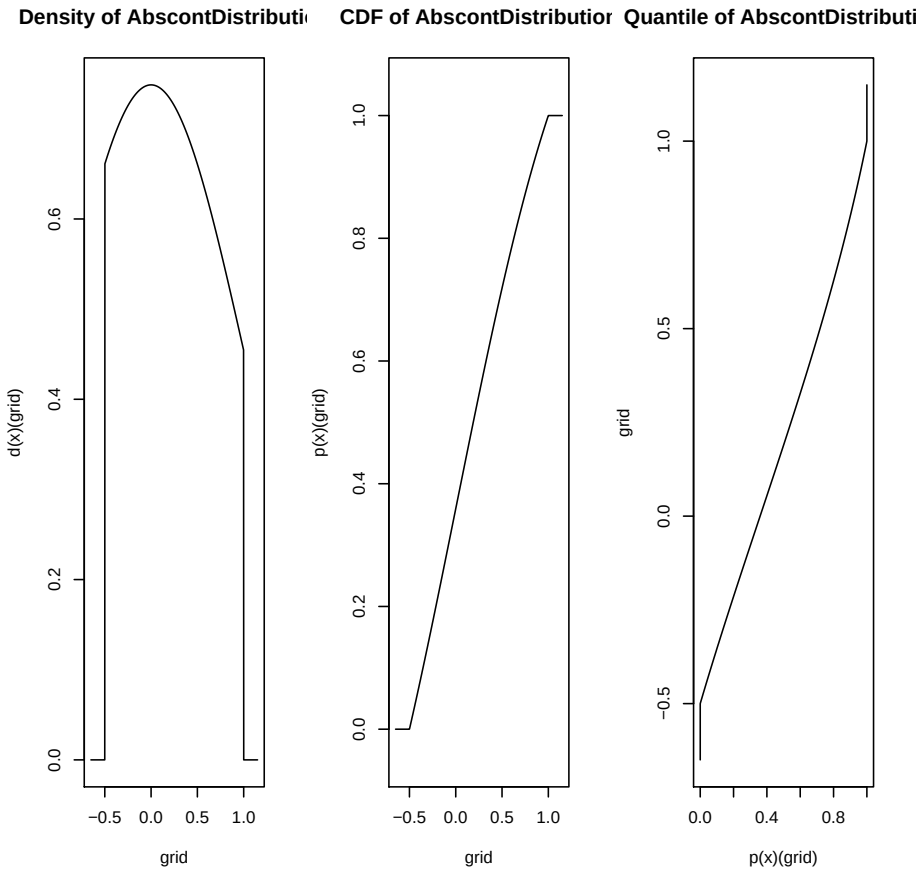
```
> require(distr)

[1] TRUE

> if (!isGeneric("Truncate")) setGeneric("Truncate", function(object,
+     lower, upper) standardGeneric("Truncate"))

[1] "Truncate"

> setMethod("Truncate", signature(object = "AbscontDistribution",
+     lower = "numeric", upper = "numeric"), function(object, lower,
+     upper) {
+     rnew = function(n) {
+         rn = r(object)(n)
+         while (TRUE) {
+             rn[rn < lower] = NA
+             rn[rn > upper] = NA
+             index = is.na(rn)
+             if (!(any(index)))
+                 break
+             rn[index] = r(object)(sum(index))
+         }
+         rn
+     }
+     plower = p(object)(lower)
+     pupper = p(object)(upper)
+     pnew = function(x) ifelse(x < lower, 0, ifelse(x >= upper,
+         1, (p(object)(x) - plower)/(pupper - plower)))
+     lostmass = plower + 1 - pupper
+     dnew = function(x) ifelse(x < lower, 0, ifelse(x >= upper,
+         0, d(object)(x)/(1 - lostmass)))
+     qfun1 <- function(x) {
+         if (x == 0)
+             return(lower)
+         if (x == 1)
+             return(upper)
+         fun <- function(t) pnew(t) - x
+         uniroot(fun, interval = c(lower, upper))$root
+     }
+     qfun2 <- function(x) sapply(x, qfun1)
+     return(new("AbscontDistribution", r = rnew, d = dnew, p = pnew,
+         q = qfun2))
+ })
```

```
[1] "Truncate"
```

```
> N = Norm()
> Z = Truncate(N, -0.5, 1)
> plot(Z)
> r(Z)(10)
```

```
 [1] -0.26004404 -0.20930077  0.42901044  0.20736589 -0.32304838  0.02591731
 [7] -0.16227450  0.17318232  0.58873571  0.78461899
```

```
> oldpar = par()
> par(mfrow = c(1, 2))
> x = seq(-1.5, 1.5, length = 1000)
> plot(x, p(Z)(x), type = "l", lwd = 5, xlab = "", ylab = "CDF")
> lines(x, p(N)(x), lwd = 2, col = "red")
> legend(-1.5, 1, legend = c("N(0,1)", "N(0,1) truncated"), fill = c("red",
+     "black"))
> x = seq(-1.5, 1.5, length = 1000)
> plot(x, d(Z)(x), type = "l", lwd = 5, xlab = "", ylab = "density")
> lines(x, d(N)(x), lwd = 2, col = "red")
> legend(-1.5, 0.7, legend = c("N(0,1)", "N(0,1) truncated"), fill = c("red",
+     "black"))
> par(oldpar)
```



11.6 Distribution of minimum and maximum of two independent random variables

Code also available under

http://www.uni-bayreuth.de/departments/math/org/
/mathe7/DISTR/minandmax.R

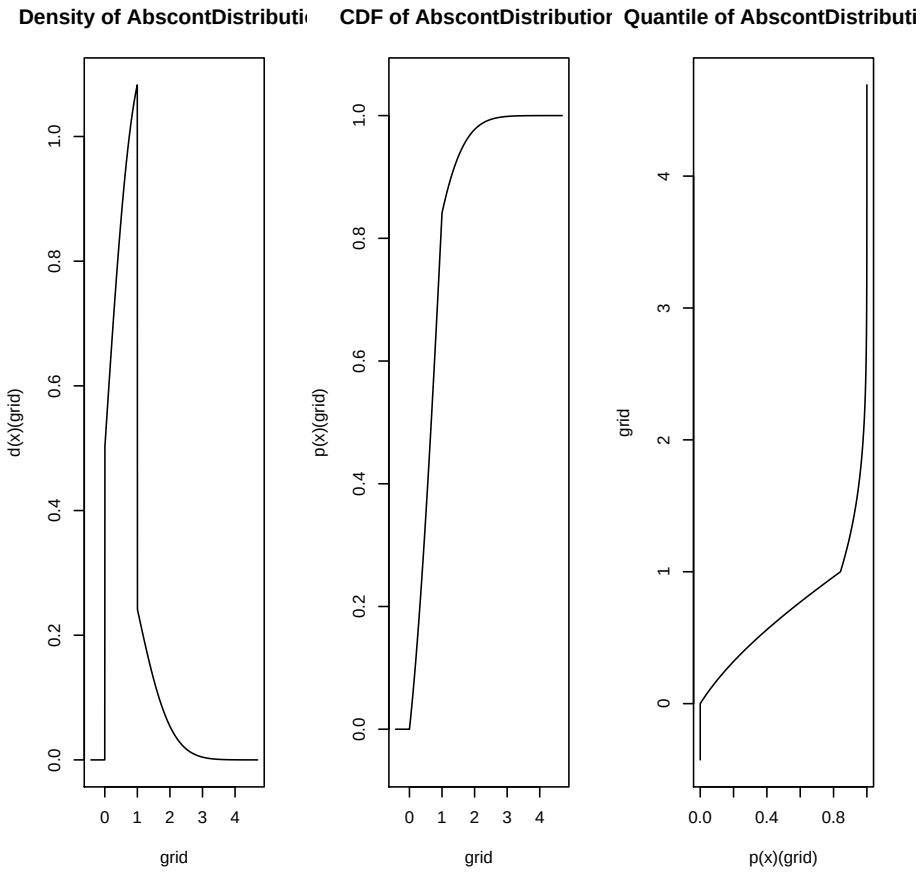
As in the preceding example, we illustrate the use of package "distr" by making available widely necessary operations: Minimum and maximum of two independent random variables.

```
> require(distr)

[1] TRUE

> if (!isGeneric("Minimum")) setGeneric("Minimum", function(e1,
+     e2) standardGeneric("Minimum"))
```

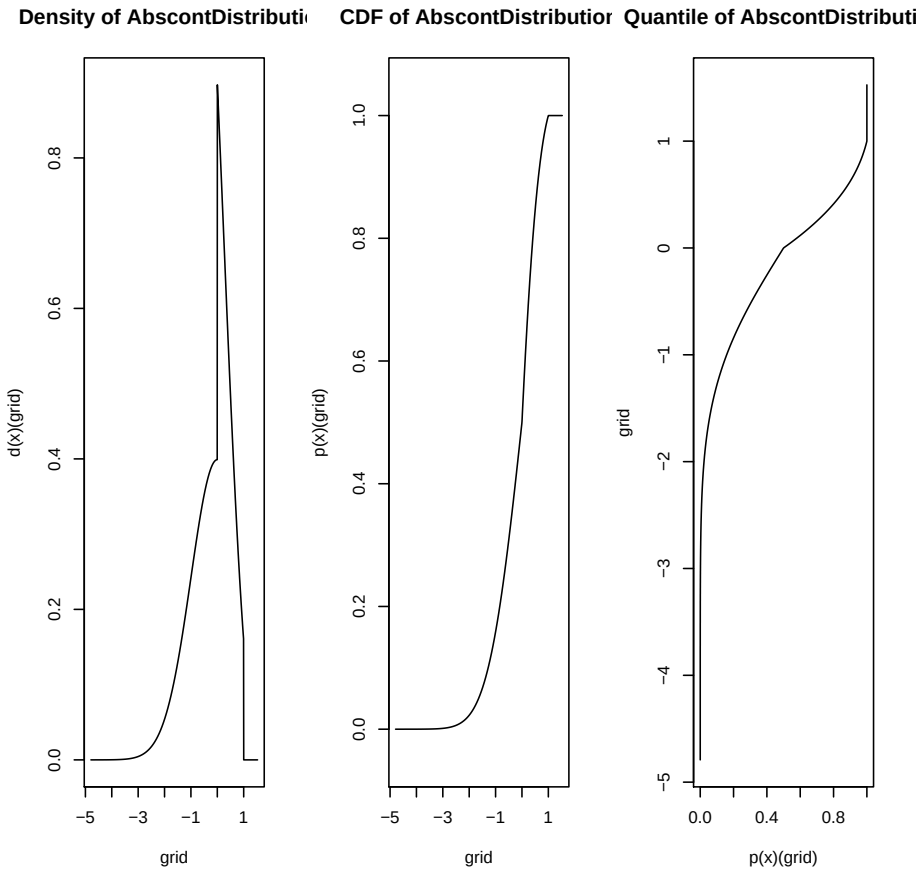
```
[1] "Minimum"

> setMethod("Minimum", signature(e1 = "AbscontDistribution", e2 = "AbscontDistribution"),
+     function(e1, e2) {
+         rnew <- function(n) {
+             rn1 <- r(e1)(n)
+             rn2 <- r(e2)(n)
+             ifelse(rn1 < rn2, rn1, rn2)
+         }
+         pnew <- function(x) {
+             p1 <- p(e1)(x)
+             p2 <- p(e2)(x)
+             p1 + p2 - p1 * p2
+         }
+         dnew <- function(x) {
+             d1 <- d(e1)(x)
+             d2 <- d(e2)(x)
+             p1 <- p(e1)(x)
+             p2 <- p(e2)(x)
+             d1 + d2 - d1 * p2 - p1 * d2
+         }
+         lower1 <- q(e1)(0)
+         lower2 <- q(e2)(0)
+         upper1 <- q(e1)(1)
+         upper2 <- q(e2)(1)
+         lower <- min(lower1, lower2)
+         upper <- min(upper1, upper2)
+         maxquantile = min(q(e1)(1 - 1e-06), q(e2)(1 - 1e-06))
+         minquantile = min(q(e1)(1e-06), q(e2)(1e-06))
+         qfun1 <- function(x) {
+             if (x == 0)
+                 return(lower)
+             if (x == 1)
+                 return(upper)
+             fun <- function(t) pnew(t) - x
+             uniroot(fun, interval = c(maxquantile, minquantile))$root
+         }
+         qfun2 <- function(x) sapply(x, qfun1)
+         return(new("AbscontDistribution", r = rnew, d = dnew,
+             p = pnew, q = qfun2))
+     })
```

```
[1] "Minimum"

> if (!isGeneric("Maximum")) setGeneric("Maximum", function(e1,
+     e2) standardGeneric("Maximum"))

[1] "Maximum"

> setMethod("Maximum", signature(e1 = "AbscontDistribution", e2 = "AbscontDistribution"),
+     function(e1, e2) {
+         rnew <- function(n) {
+             rn1 <- r(e1)(n)
+             rn2 <- r(e2)(n)
+             ifelse(rn1 > rn2, rn1, rn2)
+         }
+         pnew <- function(x) {
+             p1 <- p(e1)(x)
+             p2 <- p(e2)(x)
+             p1 * p2
+         }
+         dnew <- function(x) {
+             d1 <- d(e1)(x)
+             d2 <- d(e2)(x)
+             p1 <- p(e1)(x)
+             p2 <- p(e2)(x)
+             d1 * p2 + p1 * d2
+         }
+         lower1 <- q(e1)(0)
+         lower2 <- q(e2)(0)
+         upper1 <- q(e1)(1)
+         upper2 <- q(e2)(1)
+         lower <- max(lower1, lower2)
+         upper <- max(upper1, upper2)
+         maxquantile = max(q(e1)(1 - 1e-06), q(e2)(1 - 1e-06))
+         minquantile = max(q(e1)(1e-06), q(e2)(1e-06))
+         qfun1 <- function(x) {
+             if (x == 0)
+                 return(lower)
+             if (x == 1)
+                 return(upper)
+             fun <- function(t) pnew(t) - x
+             uniroot(fun, interval = c(maxquantile, minquantile))$root
+         }
```

```
+         qfun2 <- function(x) sapply(x, qfun1)
+         return(new("AbscontDistribution", r = rnew, d = dnew,
+             p = pnew, q = qfun2))
+     })

[1] "Maximum"

> N <- Norm(mean = 0, sd = 1)
> U <- Unif(Min = 0, Max = 1)
> Y <- Maximum(N, U)
> plot(Y)
```




```
> Z <- Minimum(N, U)
> plot(Z)
```

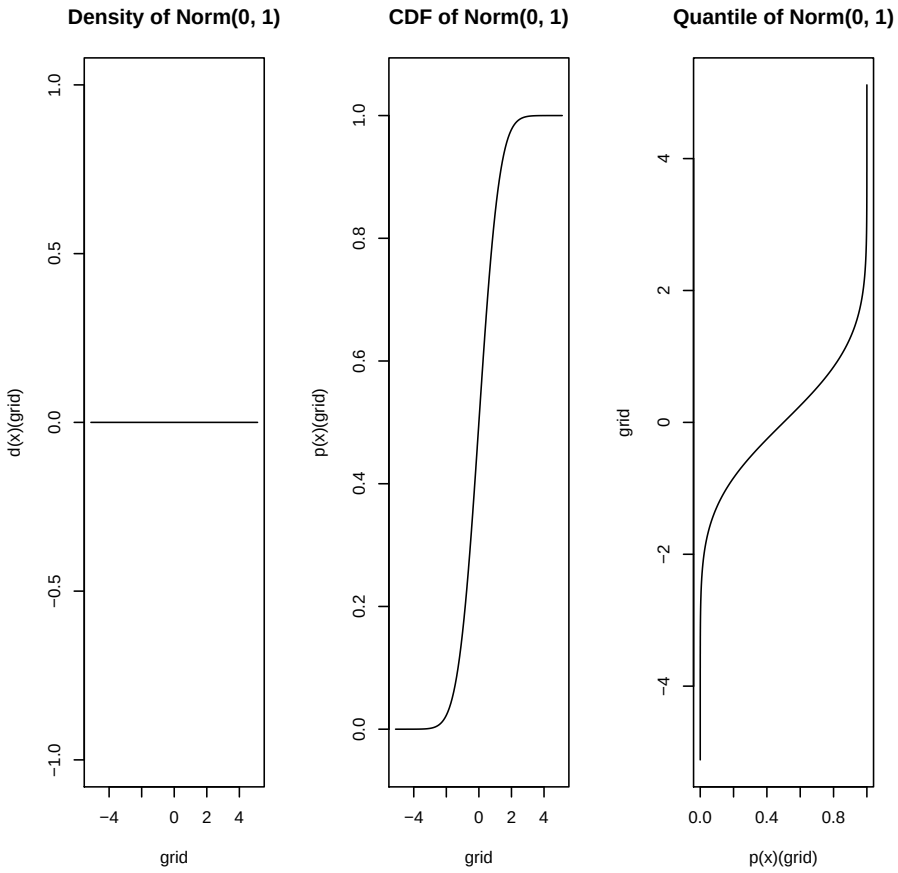


11.7 Instructive destructive example

Code also available under

http://www.uni-bayreuth.de/departments/math/org/
/mathe7/DISTR/destructive.R

```
> require(distr)

[1] TRUE

> N <- Norm()
> B <- Binom()
> N@d <- B@d
> plot(N)
```



11.8 A simulation example

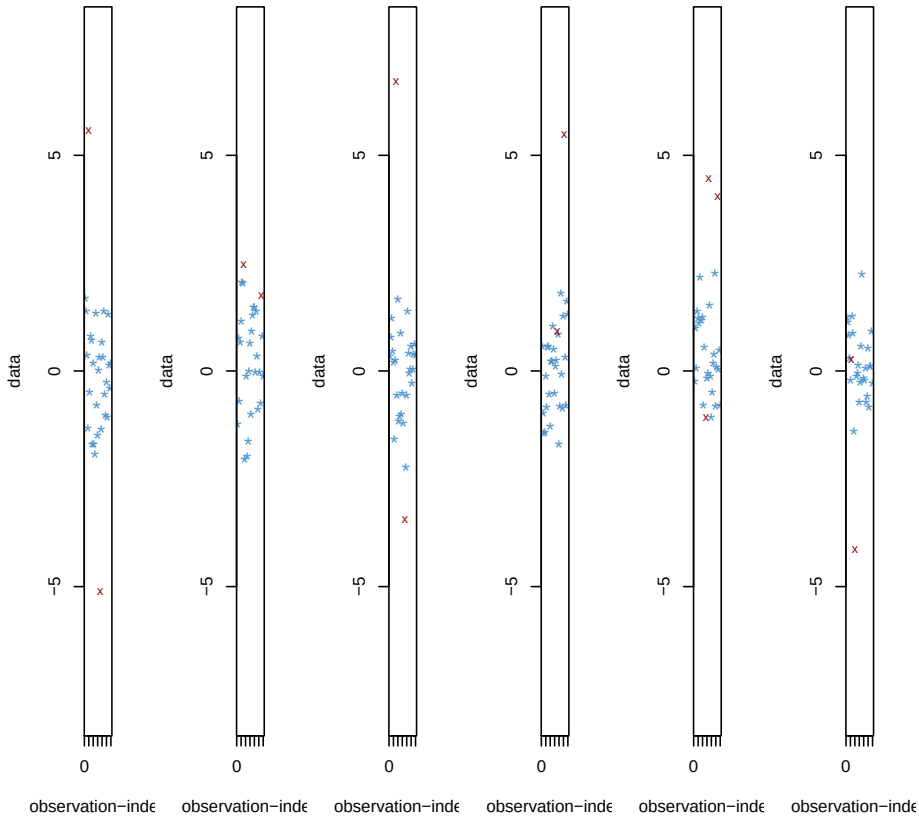
needs packages `"distrSim"/"distrTEst"`

Code also available under

http://www.uni-bayreuth.de/departments/math/org/
/mathe7/DISTR/SimulateandEstimate.R

```
> have.distrTEst <- suppressWarnings(require(distrTEst))
> if (have.distrTEst) {
+     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)
```

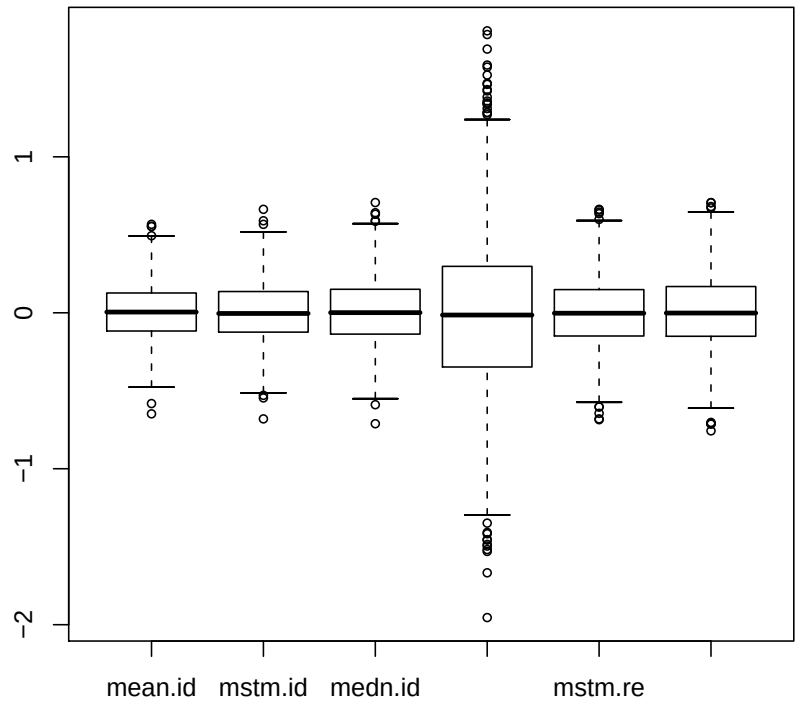
```
+     simulate(sim)
+     simulate(contsim)
+     print(sim)
+     summary(contsim)
+     plot(contsim)
+ } else {
+     cat("\n functionality not (yet) available; ")
+     cat("you have to install package \"distrTEst\" first.\n")
+ }

filename of Simulation: sim_01
seed of Simulation: Mersenne-Twister
 seed of Simulation: Inversion
 seed of Simulation: c(312, 992586505, -1091336977, -1873268990, 1830591758, 568584456, -17
number of runs: 1000
dimension of the observations: 1
size of sample: 30
object was generated by version: 1.8
Distribution:
Distribution Object of Class: Norm
mean :  0
sd :  1
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
```




```
> have.distrTEst <- suppressWarnings(require("distrTEst"))
> if (have.distrTEst) {
+     psim <- function(theta, y, m0) {
+         mean(pmin(pmax(-m0, y - theta), m0))
+     }
+     mestimator <- function(x, m = 0.7) {
+         uniroot(psim, low = -20, up = 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)
+     print(elist)
+     summary(elist)
+     plot(elist, cex = 0.7)
+ } else {
+     cat("\n functionality not (yet) available; ")
+     cat("you have to install package \"distrTEst\" first.\n")
+ }

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  -0.13697  0.06051  0.00985 -0.04599  0.43008 ...
 $ mean.re: num  -0.064  0.991 -0.445  0.237  1.423 ...
---------------------------------
An Evaluation Object
estimator: mestimator
Result: 'data.frame':        1000 obs. of  2 variables:
 $ mstm.id: num  -0.0720  0.0621  0.0818 -0.0753  0.3795 ...
 $ mstm.re: num  -0.0483  0.5620  0.0514  0.1390  0.5566 ...
---------------------------------
An Evaluation Object
estimator: median
Result: 'data.frame':        1000 obs. of  2 variables:
 $ medn.id: num  0.10299 0.00262 0.15605 0.05464 0.30376 ...
 $ medn.re: num  0.103 0.681 0.156 0.260 0.465 ...
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.   :-0.647788   Min.   :-1.95427
```

```
 1st Qu.:-0.116459   1st Qu.:-0.34752
 Median : 0.004664   Median :-0.01436
 Mean   : 0.005352   Mean   :-0.01149
 3rd Qu.: 0.126965   3rd Qu.: 0.29756
 Max.   : 0.567246   Max.   : 1.80795
----------------------------------
name of Evaluation: object
estimator: mestimator
Result:
    mstm.id             mstm.re
 Min.   :-0.680226   Min.   :-0.685379
 1st Qu.:-0.123801   1st Qu.:-0.148639
 Median :-0.004340   Median :-0.002694
 Mean   : 0.004876   Mean   : 0.002649
 3rd Qu.: 0.136163   3rd Qu.: 0.148642
 Max.   : 0.663399   Max.   : 0.663399
----------------------------------
name of Evaluation: object
estimator: median
Result:
    medn.id             medn.re
 Min.   :-0.711295   Min.   :-0.757078
 1st Qu.:-0.136414   1st Qu.:-0.150264
 Median : 0.000839   Median :-0.001653
 Mean   : 0.008166   Mean   : 0.005374
 3rd Qu.: 0.150733   3rd Qu.: 0.167842
 Max.   : 0.706914   Max.   : 0.706914
```



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  -0.0720  0.0621  0.0818 -0.0753  0.3795 ...
 $ mstm.re: num  -0.0483  0.5620  0.0514  0.1390  0.5566 ...
```

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.   :-0.647788   Min.   :-1.95427
 1st Qu.:-0.116459   1st Qu.:-0.34752
 Median : 0.004664   Median :-0.01436
 Mean   : 0.005352   Mean   :-0.01149
 3rd Qu.: 0.126965   3rd Qu.: 0.29756
 Max.   : 0.567246   Max.   : 1.80795
----------------------------------
name of Evaluation: object
estimator: mestimator
Result:
    mstm.id             mstm.re
 Min.   :-0.680226   Min.   :-0.685379
 1st Qu.:-0.123801   1st Qu.:-0.148639
 Median :-0.004340   Median :-0.002694
 Mean   : 0.004876   Mean   : 0.002649
 3rd Qu.: 0.136163   3rd Qu.: 0.148642
 Max.   : 0.663399   Max.   : 0.663399
----------------------------------
name of Evaluation: object
estimator: median
Result:
    medn.id             medn.re
 Min.   :-0.711295   Min.   :-0.757078
 1st Qu.:-0.136414   1st Qu.:-0.150264
 Median : 0.000839   Median :-0.001653
 Mean   : 0.008166   Mean   : 0.005374
 3rd Qu.: 0.150733   3rd Qu.: 0.167842
 Max.   : 0.706914   Max.   : 0.706914
```

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...

11.9 Expectation of a given function under a given distribution

Code also available under

http://www.uni-bayreuth.de/departments/math/org/
/mathe7/DISTR/Expectation.R

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. As in examples 11.5 and 11.6, we illustrate the use of package "distr" by implementing a general evaluation of expectation and variance under a given distribution.

```
> have.distrEx <- suppressWarnings(require("distrEx"))
> if (have.distrEx) {
+     id <- function(x) x
+     sq <- function(x) x^2
```

```
+     B <- Binom(6, 0.5)
+     E(B, id)
+     E(B, sq) - E(B, id)^2
+     N <- Norm(1, 1)
+     E(N, id)
+     E(N, sq) - E(N, id)^2
+ } else {
+     cat("\n functionality not (yet) available; ")
+     cat("you have to install package \"distrEx\" first.\n")
+ }

[1] 1
```

11.10 n-fold convolution of absolutely continuous distributions

Code also available under

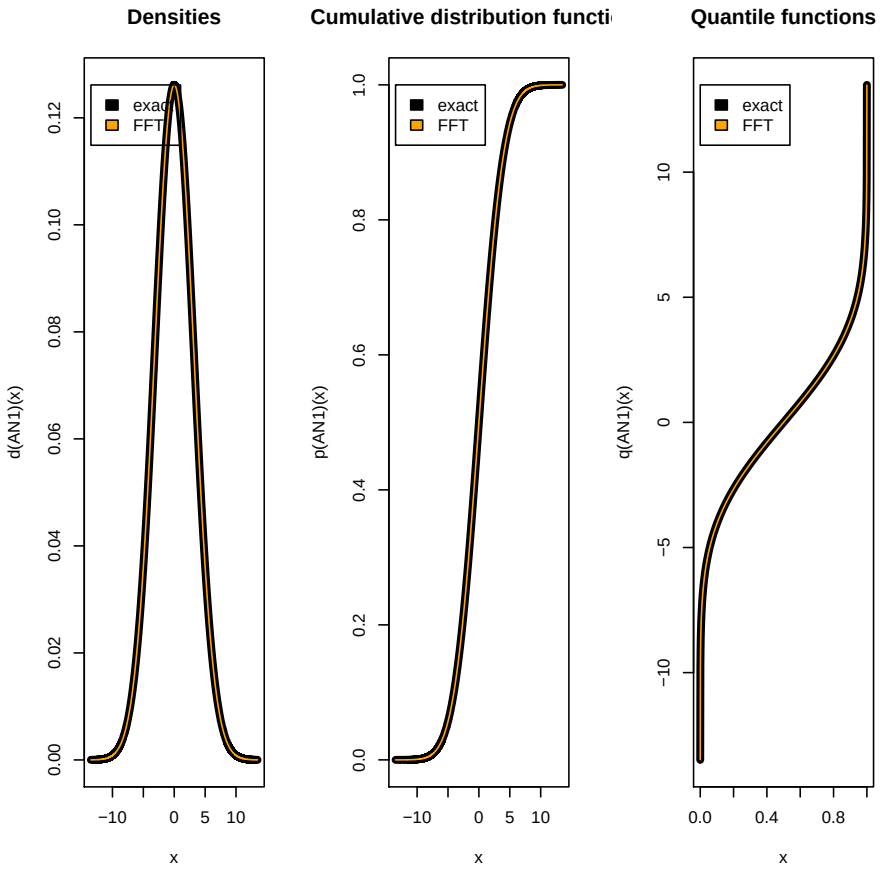
http://www.uni-bayreuth.de/departments/math/org/
/mathe7/DISTR/nFoldConvolution.R

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.

```
> require(distr)

[1] TRUE

> if (!isGeneric("convpow")) setGeneric("convpow", function(D1,
+     N) standardGeneric("convpow"))

[1] "convpow"

> setMethod("convpow", signature(D1 = "AbscontDistribution", N = "numeric"),
+     function(D1, N) {
+         if ((N < 1) || (!identical(floor(N), N)))
+             stop("N has to be a natural greater than 0")
+         m <- getdistrOption("DefaultNrFFTGridPointsExponent")
+         lower <- ifelse((q(D1)(0) > -Inf), q(D1)(0), q(D1)(getdistrOption("TruncQuantile"
+         upper <- ifelse((q(D1)(1) < Inf), q(D1)(1), q(D1)(1 -
+             getdistrOption("TruncQuantile")))
+         M <- 2^m
+         h <- (upper - lower)/M
+         if (h > 0.01)
```

```
+             warning(paste("Grid for approxfun too wide, ", "increase DefaultNrFFTGridPoin
+                 sep = ""))
+         x <- seq(from = lower, to = upper, by = h)
+         p1 <- p(D1)(x)
+         p1 <- p1[2:(M + 1)] - p1[1:M]
+         pn <- c(p1, numeric((N - 1) * M))
+         fftpn <- fft(pn)
+         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)
+         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)
+         standardizer <- sum(dn[2:i.max]) + (dn[1] + dn[i.max +
+             1])/2
+         dnfun2 <- function(x) dnfun1(x)/standardizer
+         pnfun1 <- approxfun(x = x + 0.5 * h, y = pn, yleft = 0,
+             yright = pn[i.max + 1])
+         pnfun2 <- function(x) pnfun1(x)/pn[i.max + 1]
+         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))
+     })

[1] "convpow"

> A <- Norm(mean = 0, sd = 1)
> N <- 10
> AN <- convpow(A, N)
> AN1 <- Norm(mean = 0, sd = sqrt(N))
> eps <- getdistrOption("TruncQuantile")
> par(mfrow = c(1, 3))
> low <- q(AN1)(eps)
> upp <- q(AN1)(1 - eps)
> x <- seq(from = low, to = upp, length = 10000)
> plot(x, d(AN1)(x), type = "l", lwd = 5)
> lines(x, d(AN)(x), col = "orange", lwd = 1)
> title("Densities")
> legend(low, d(AN)(0), legend = c("exact", "FFT"), fill = c("black",
+     "orange"))
> plot(x, p(AN1)(x), type = "l", lwd = 5)
> lines(x, p(AN)(x), col = "orange", lwd = 1)
> title("Cumulative distribution functions")
> legend(low, 1, legend = c("exact", "FFT"), fill = c("black",
+     "orange"))
> 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(0, q(AN1)(1 - eps), legend = c("exact", "FFT"), fill = c("black",
+     "orange"))
```



References

[1] Bengtsson H. 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/ 6

[2] Chambers J.M. *Programming with data. A guide to the S language*. Springer. http://cm.bell-labs.com/stat/Sbook/index.html 6

[3] Gentleman R. *Object Orientated Programming. Slides of a Short Course held in Auckland*. http://www.stat.auckland.ac.nz/S-Workshop/Gentleman/Methods.pdf 27

[4] Kohl M. *Numerical Contributions to the Asymptotic Theory of Robustness*. Dissertation, Universität Bayreuth. See also http://stamats.de/ThesisMKohl.pdf 2

[5] Kohl M., Ruckdeschel P. and Stabla T. General Purpose Convolution Algorithm for Distributions in S4-Classes by means of FFT. unpublished manual 6, 15, 27

[6] Press W.H., Teukolsky S.A., Vetterling W.T. and Flannery B.P. *Numerical recipes in C. The art of scientific computing.* Cambridge Univ. Press, 2. Aufl. 19

[7] Rice J.A. *Mathematical statistics and data analysis.* The Wadsworth & Brooks/Cole Statistics/Probability Series. Wadsworth & Brooks/Cole Advanced Books & Software, Pacific Grove, California. 29

[8] Ruckdeschel P., Kohl M., Stabla T., and Camphausen F. S4 Classes for Distributions. *R-News*, **6**(2): 10–13. http://CRAN.R-project.org/doc/Rnews/Rnews_2006-2.pdf 3