`S4` Classes for Distributions—a manual for packages `"distr"`, `"distrEx"`, `"distrMod"`, `"distrSim"`, `"distrTEst"`, `"distrTeach"`, version `2.0`

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

Fraunhofer ITWM
Fraunhofer Platz 1
67663 Kaiserslautern
Germany

e-Mail: `Peter.Ruckdeschel@itwm.fraunhofer.de`

August 4, 2008

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 `2.0`. 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 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

*Fraunhofer ITWM, Kaiserslautern
†Universität Bayreuth
‡Hans-Sachs-Gymnasium Nürnberg
§West-LB, Düsseldorf

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.

Contents

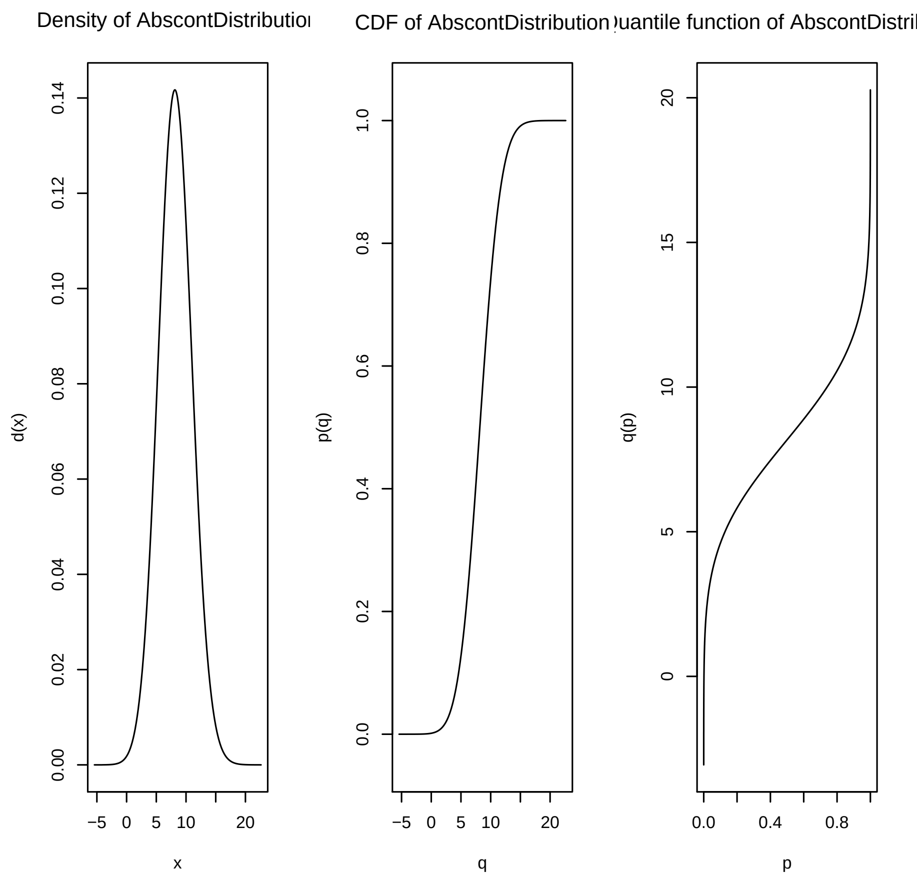
Parts of this document appeared in an earlier and much shorter form in *R-News*, **6**(2) as "S4 Classes for Distributions", c.f. [9], which in its published form refers to package versions 1.6, resp. 0.4-2. This present document takes into account the subsequent revisions and versions.

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 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, withSweave = TRUE)
> p(Z)(0.4)

[1] 0.002415384

> q(Z)(0.3)

[1] 6.70507

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

 [1] 11.89667685  8.75448130  5.55107456  8.97197849 11.40594497  5.36841722
 [7]  9.18979295  9.31295589  6.71074415  4.48770603  8.64463711 11.12739853
[13]  6.78909879  3.04039221  3.90900025  9.86494848  6.65550547  7.88391151
[19] 10.40410547  8.52891067  5.22993677  7.79680298  7.55953206 14.94109051
[25]  6.32370147  6.59769354 15.51724023  7.27695977 10.69500310 10.59820482
[31]  7.82376531  8.91652430  9.90257458 13.42048001 10.98303792 10.99336385
[37]  4.62203484 12.19955756  3.25461646 12.64993944 10.40828730  6.03374446
[43]  2.94513773  9.80422942 12.36492178  3.67964826  9.49448569  8.65414716
[49]  6.07639227 -0.05687983
```



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`, $\mathcal{L}(\mathtt{Z}) = \mathcal{L}(2 * \mathtt{N} + 3 + \mathtt{P})$, 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 vector-valued parameters make no sense; rather one could gather several objects

[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

[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.

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 have considered univariate distributions giving the S4-class UnivariateDistribution, and as typical subclasses, we have introduced 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.
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. From version 1.9 on, there are methods p.l and q.r for the left-continuous variant of the cdf, i.e.; $t \mapsto \text{p.l}(t) = P(X < t))$, and the right-continuous variant of the quantile function, i.e.;

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

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.

[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

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.
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. compoments of slot `MixDistr` may again be of

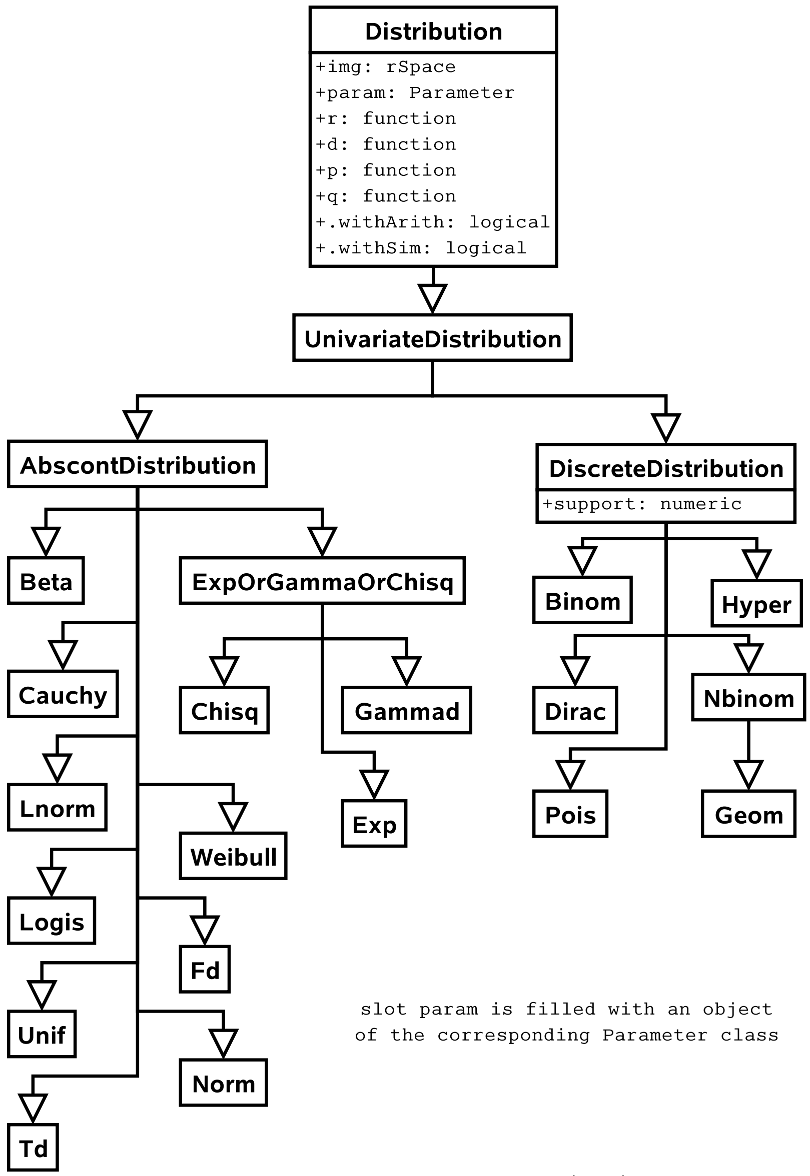


Figure 1: Inheritance relations and slots of the corresponding (sub-)classes for `Distribution` where we do not repeat inherited slots

class `UnivarMixingDistribution`.

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

```
> M2

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`.

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`.

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 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

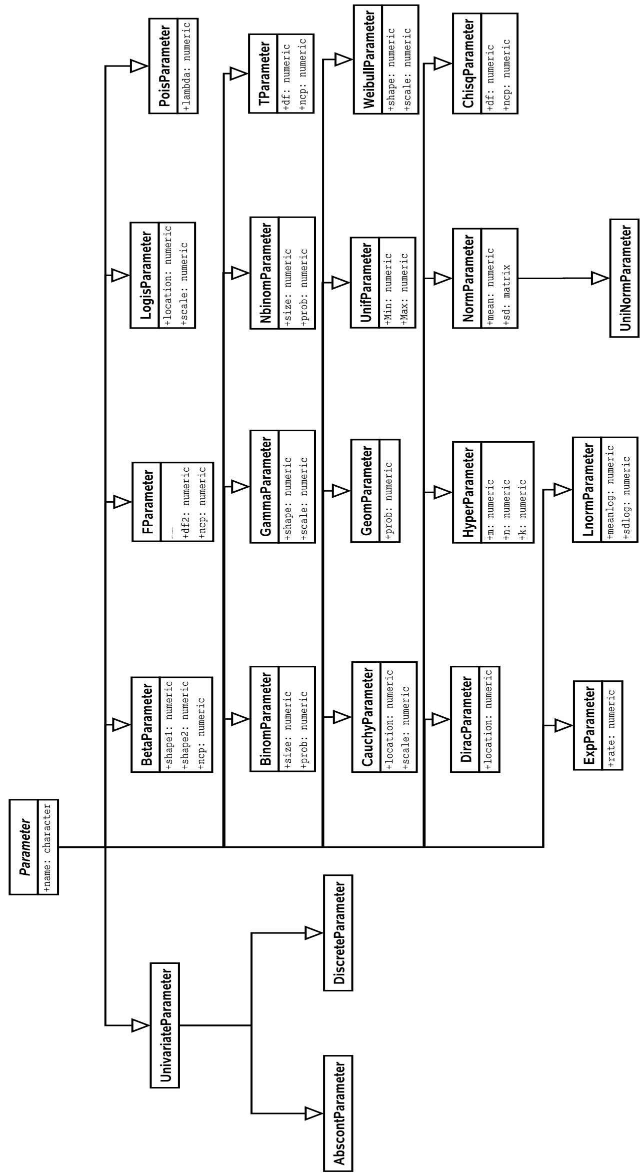


Figure 2: Inheritance relations and slots of the corresponding (sub-)classes for `Parameter`

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.

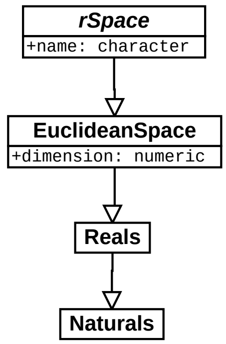


Figure 3: Inheritance relations and slots of the corresponding (sub-)classes for `rSpace`

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

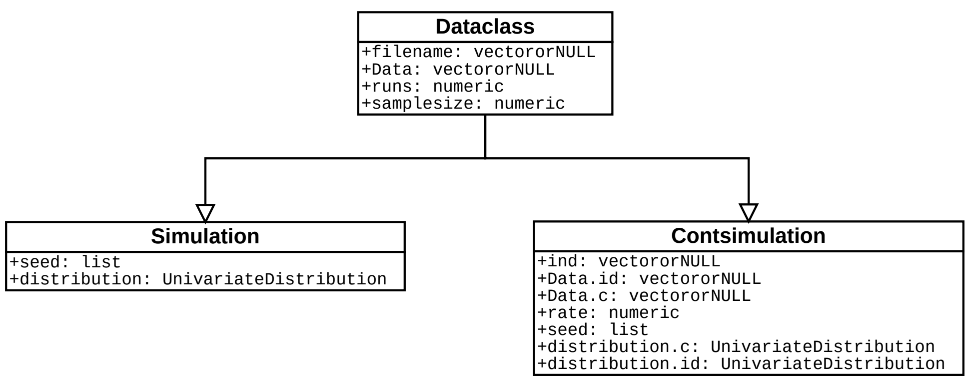


Figure 4: Inheritance relations and slots of the corresponding (sub-)classes for `Dataclass`

`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 3.12. 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

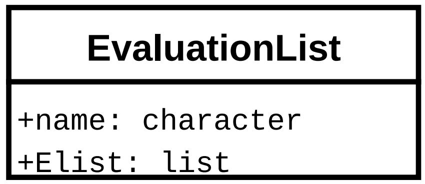


Figure 6: Slots of class `EvaluationList`

`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 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`, `AffLinDiscrete-Distribution`, `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 + 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 and Flattening

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()'.
```

```
    Its absolutely continuous part (with weight 0.773000) is a
 Distribution Object of Class: AbscontDistribution
 This part is accessible with 'acPart()'.

$pos$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()'.

    Its absolutely continuous part (with weight 0.220000) is a
 Distribution Object of Class: AbscontDistribution
 This part is accessible with 'acPart()'.

$neg$w
discreteWeight
     0.6812608


$`0`
$`0`$D
Distribution Object of Class: Dirac
 location: 0

$`0`$w
discreteWeight
     0.1246493
```

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(res$value)'.

    Its absolutely continuous part (with weight 0.620000) is a
 Distribution Object of Class: AbscontDistribution
 This part is accessible with 'acPart(res$value)'.
```

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

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 `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] 0.3848512
```

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

```
[1] 0.3802741
```

```
>   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 5).

As they are frequently needed, from version 1.9 on, math operations `abs()`, `exp()`, and —if an R-version $\geq$ `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 $10^{\texttt{RtoDPQ.e}}$ 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

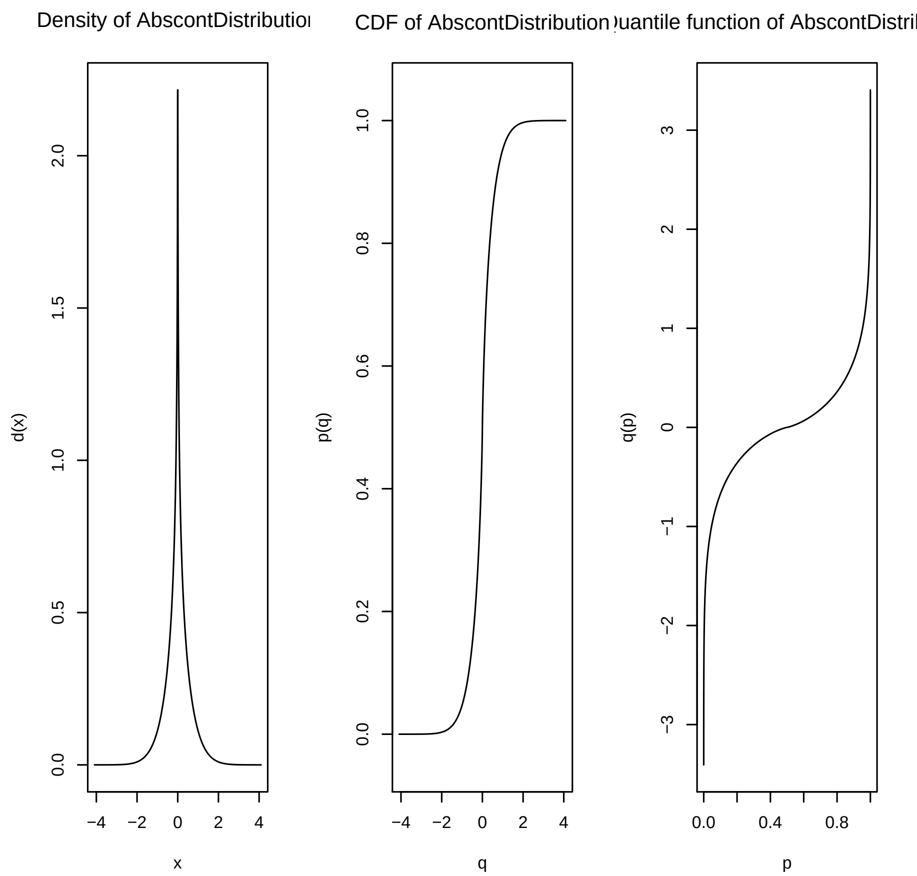
A convolution method for two independent r.v.'s is implemented by means of explicit calculations for discrete summands, and by means of DFT/FFT[7] if one of the summands is absolutely continuous or (from version 1.9 on:) both are lattice distributed with a common lattice as support. This method automatically 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

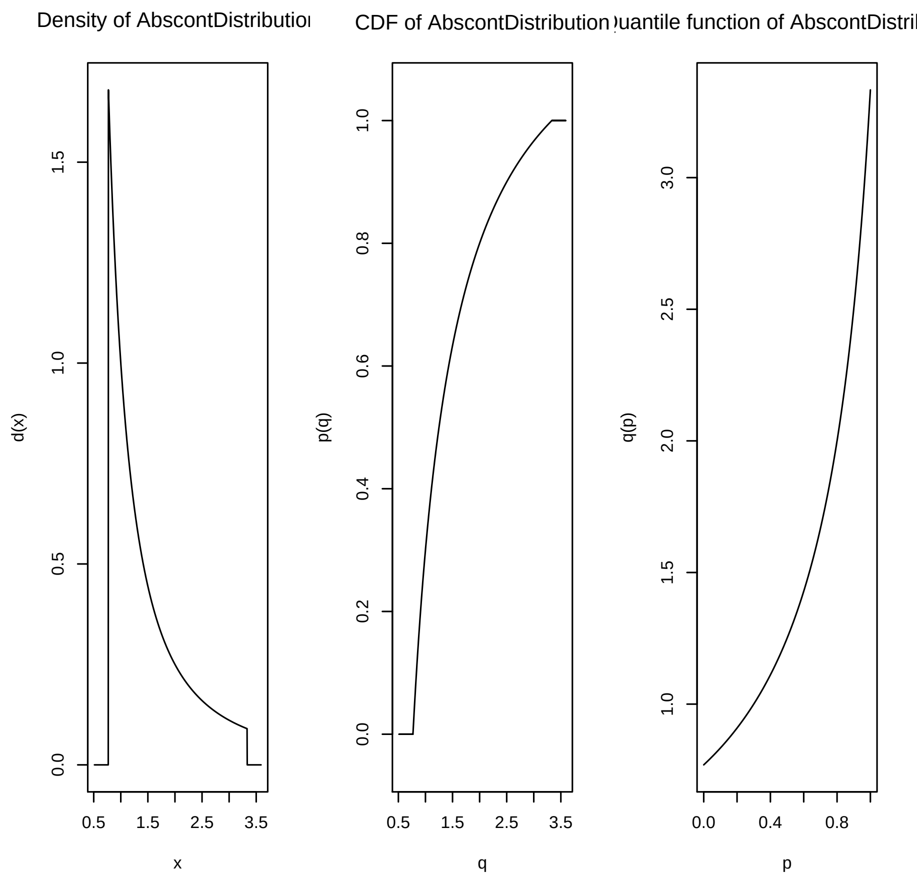
Having implemented a class for Lebesgue decomposed distributions, we have been able to realize further binary operators, in particular we have exact analytical constructions for multiplication, division, exponentiation:

```
>   A1 <- Norm(); A2 <- Unif()
>   A1A2 <- A1*A2
>   plot(A1A2, withSweave = TRUE)
```

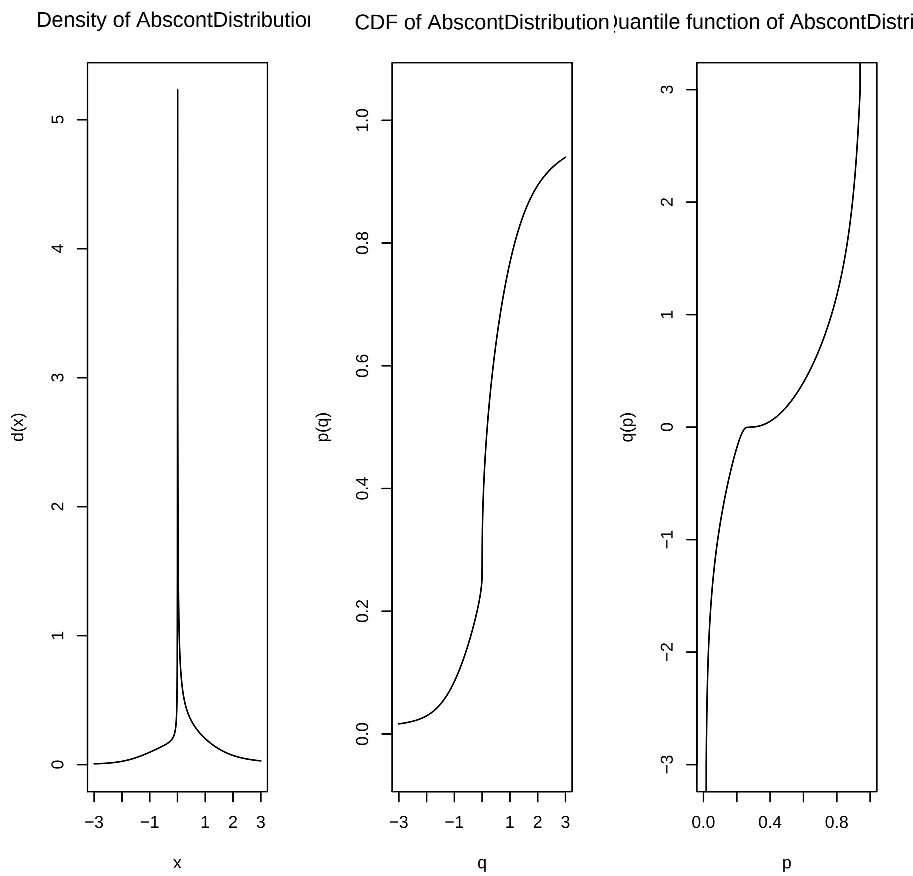
[7]Details to be found in [5]



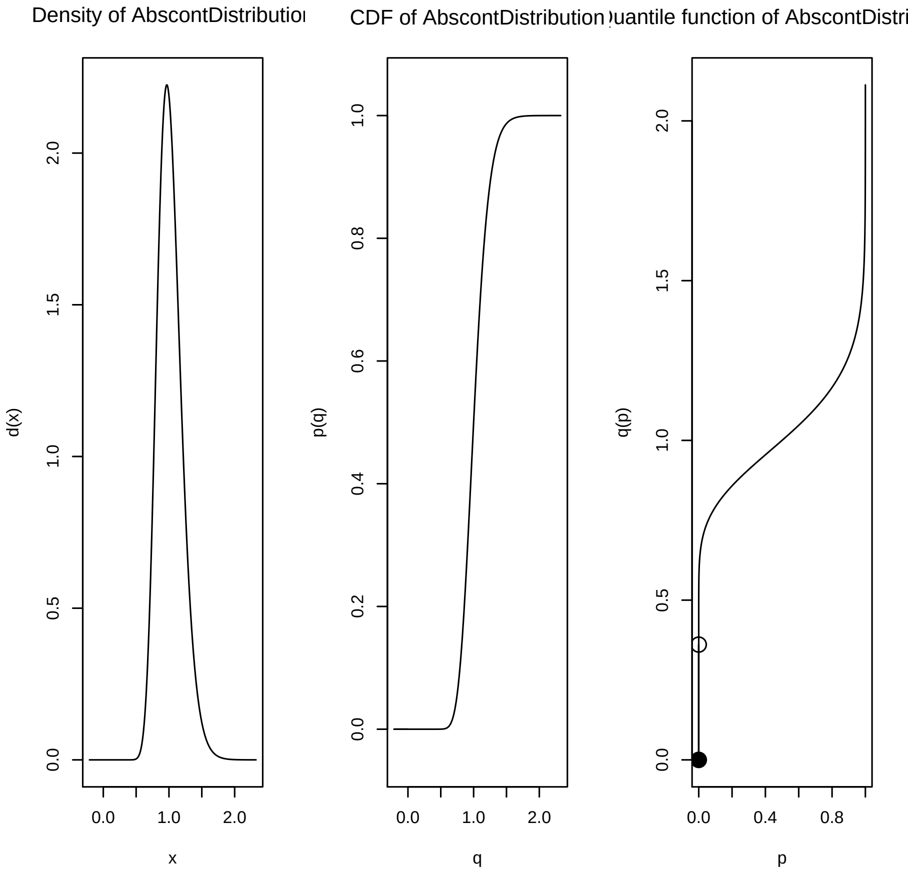

```
>   A12 <- 1/(A2 + .3)
>   plot(A12, withSweave = TRUE)
```



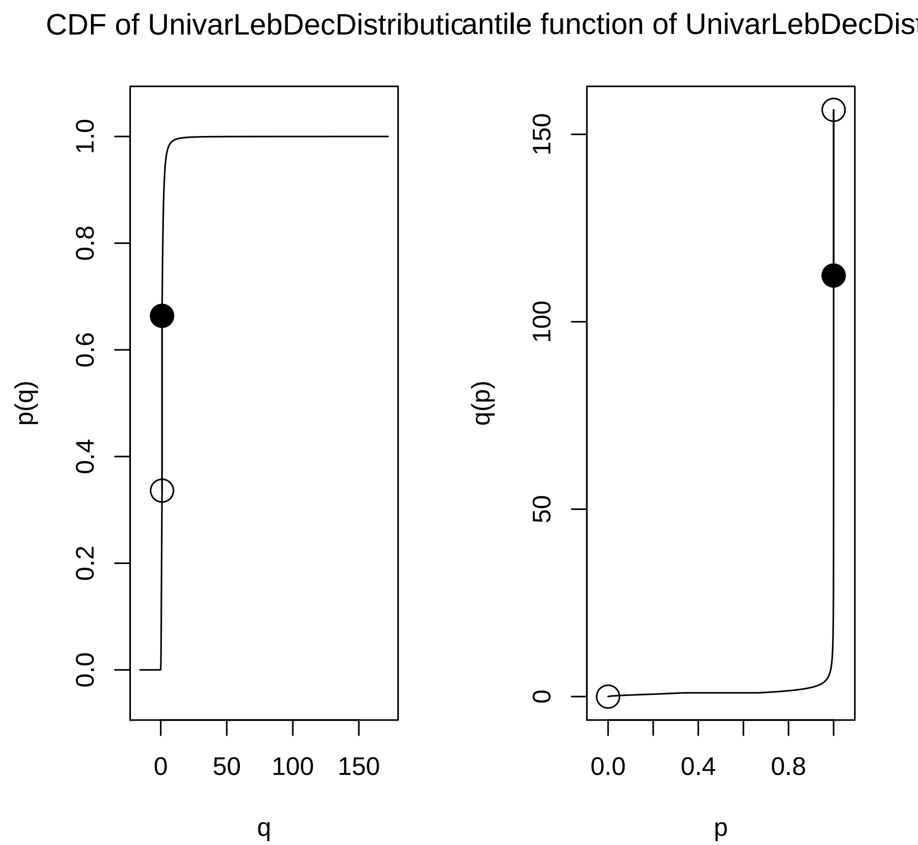

```
>   B <- Binom(5,.2)+1
>   A1B <- A1^B
>   plot(A1B, xlim=c(-3,3), withSweave = TRUE)
```




```
>   plot(1.2^A1, withSweave = TRUE)
```



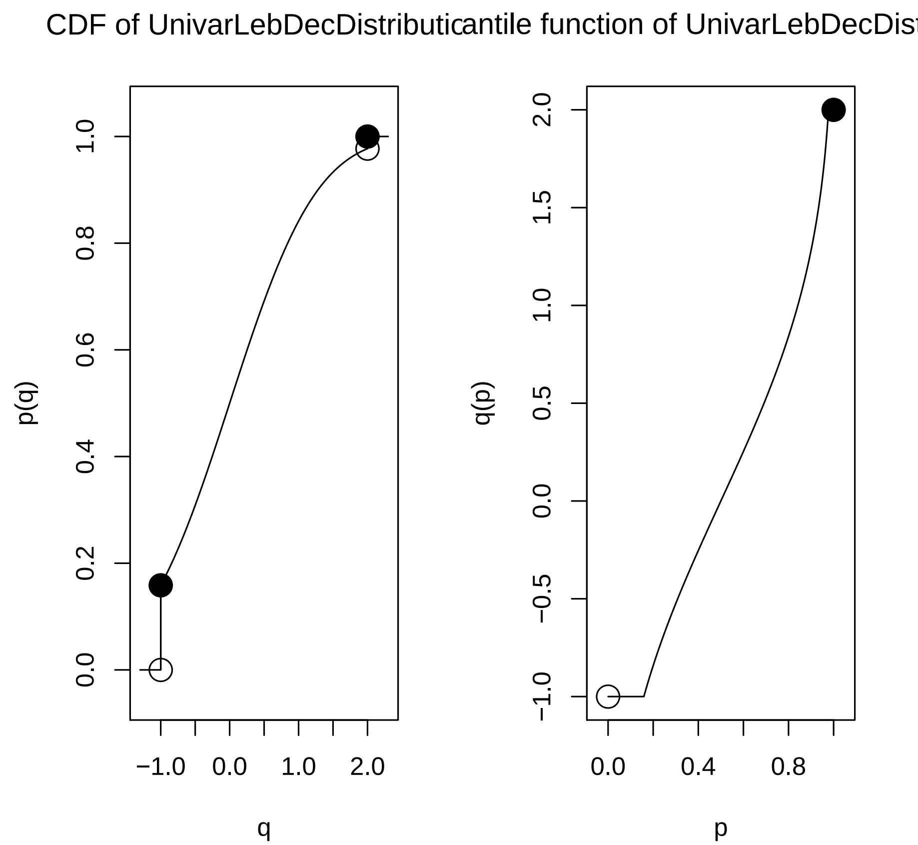

```
>   plot(B^A1, withSweave = TRUE)
```



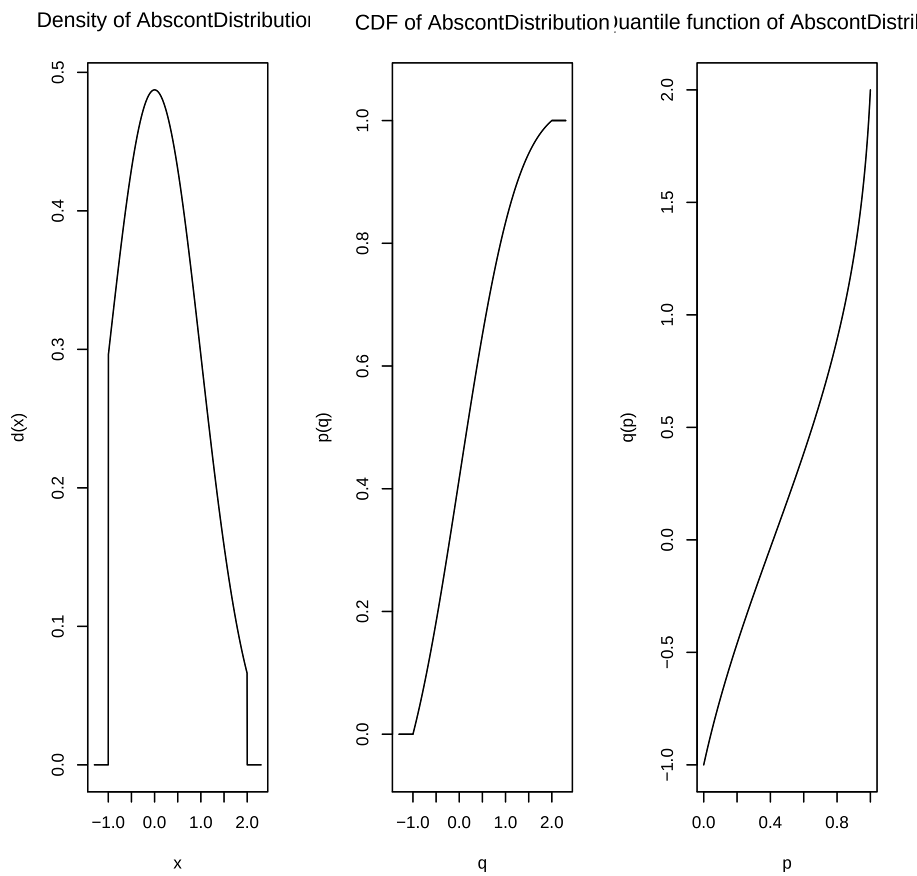
3.8 Truncation, Pairwise Minimum/Maximum, Huberization

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:

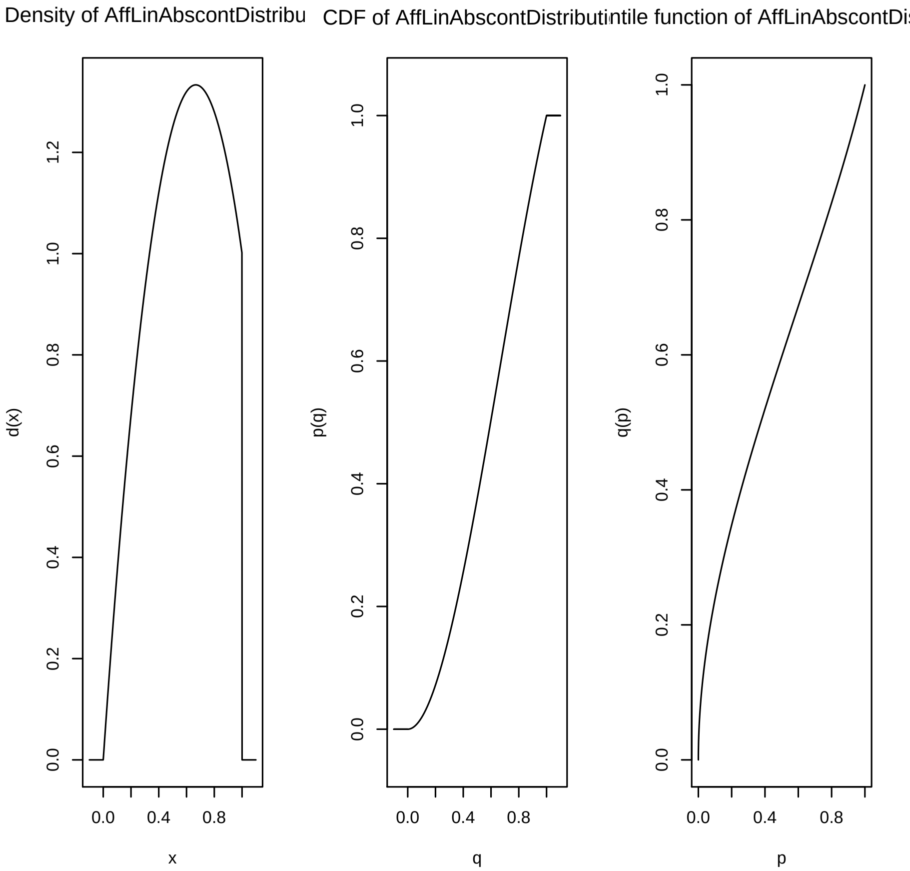
```
> H <- Huberize(Norm(),lower=-1,upper=2)
> plot(H, withSweave = TRUE)
```



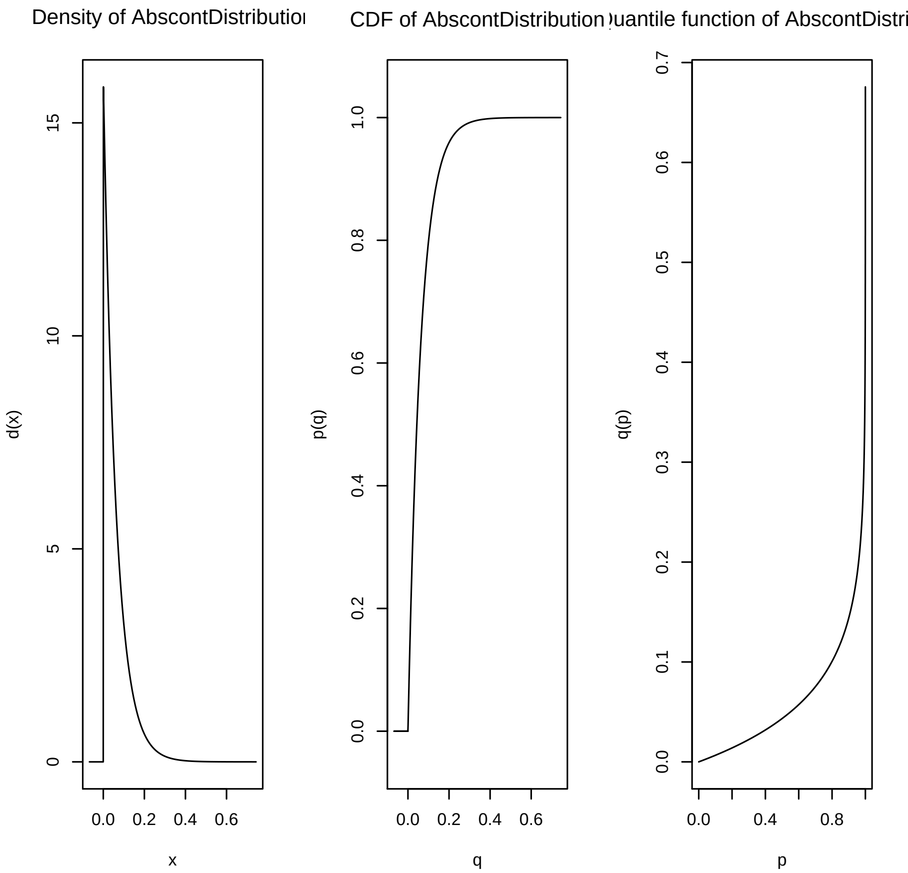

```
> T <- Truncate(Norm(),lower=-1,upper=2)
> plot(T, withSweave = TRUE)
```



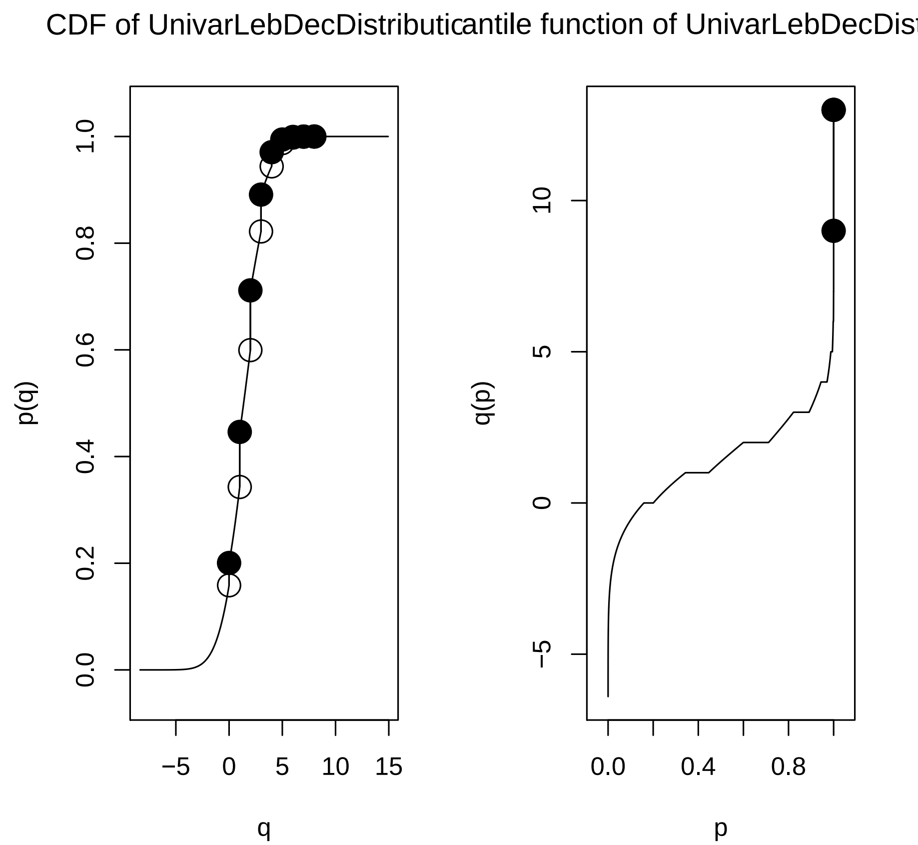

```
> M1 <- Maximum(Unif(0,1), Minimum(Unif(0,1), Unif(0,1)))
> plot(M1, withSweave = TRUE)
```




```
> M2 <- Minimum(Exp(4),4)
> plot(M2, withSweave = TRUE)
```




```
> M3 <- Minimum(Norm(2,2), Pois(3))
> plot(M3, withSweave = TRUE)
```

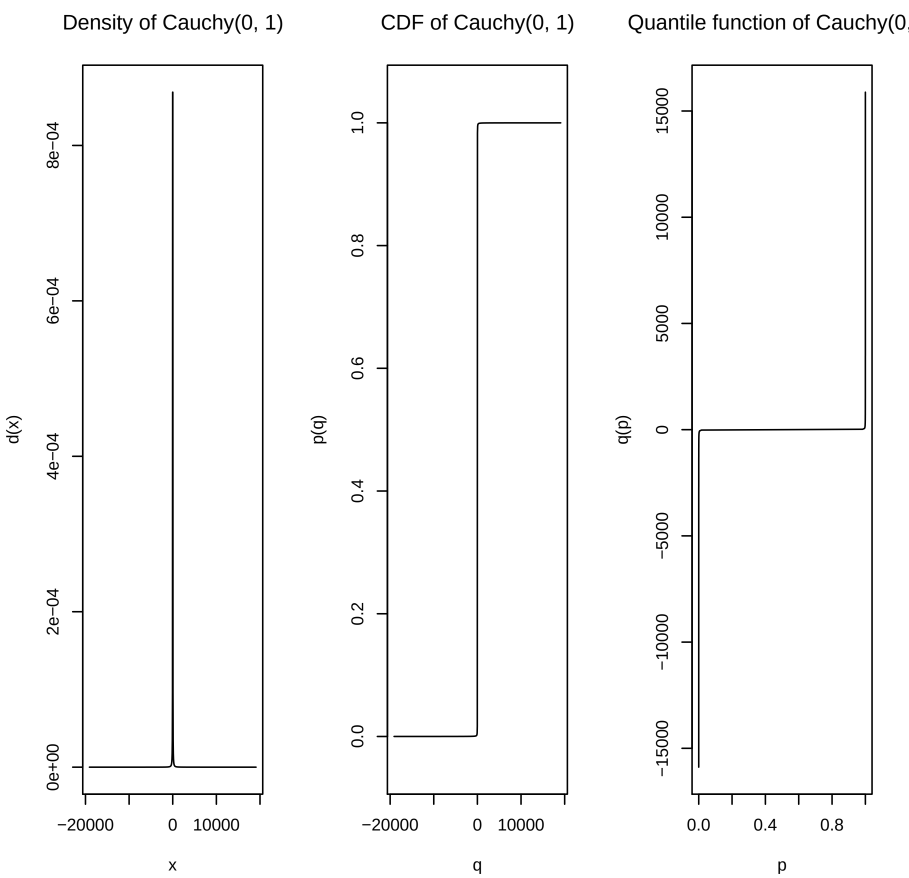


3.9 Overloaded generic functions

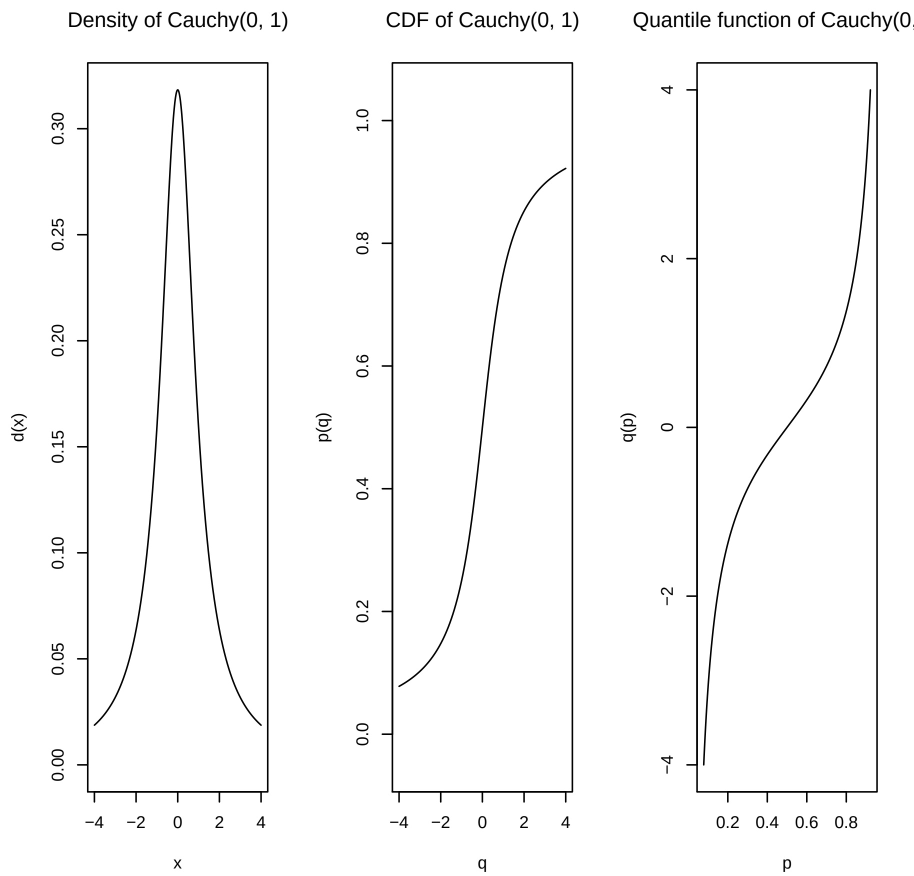
Methods `print`, `plot`, `show` and `summary` have been overloaded for classes `Distribution`, `Dataclass`, `Simulation`, `ContSimulation`, as well as `Evaluation` and `EvaluationList` to produce "pretty" output. More specifically there are also particular `show` methods for classes `UnivarDistrList`, `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"`.

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(),withSweave = TRUE)
```




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



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

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. plot() can also cope with log-arguments. 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 10 plots:

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.

3.10 liesInSupport

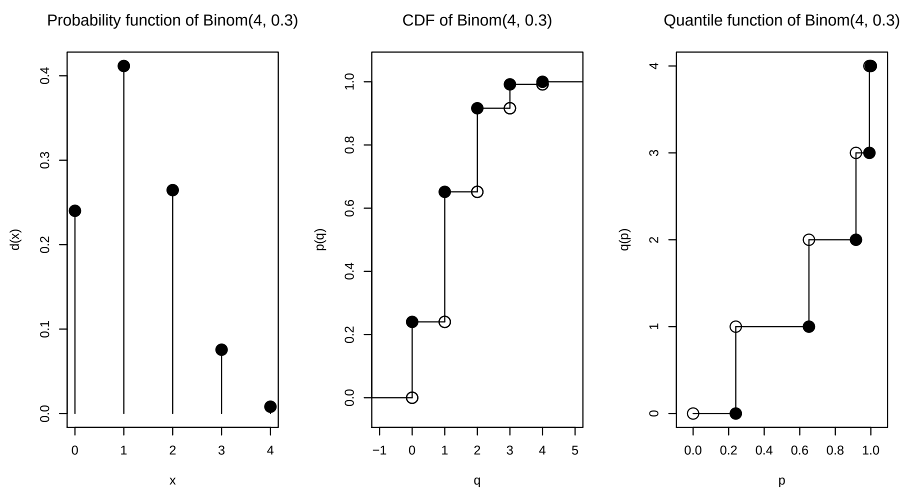
For all discrete distribution classes, we have methods liesInSupport to check whether a given vector/ a matrix of points lies in the support of the distribution.

3.11 Simulation (in package "distrSim")

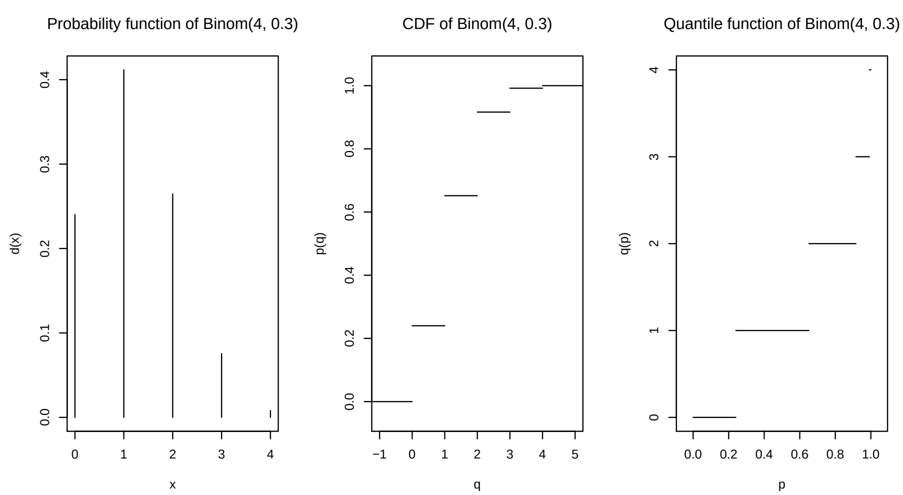
From version 1.6 on, simulation is available in package "distrSim".

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

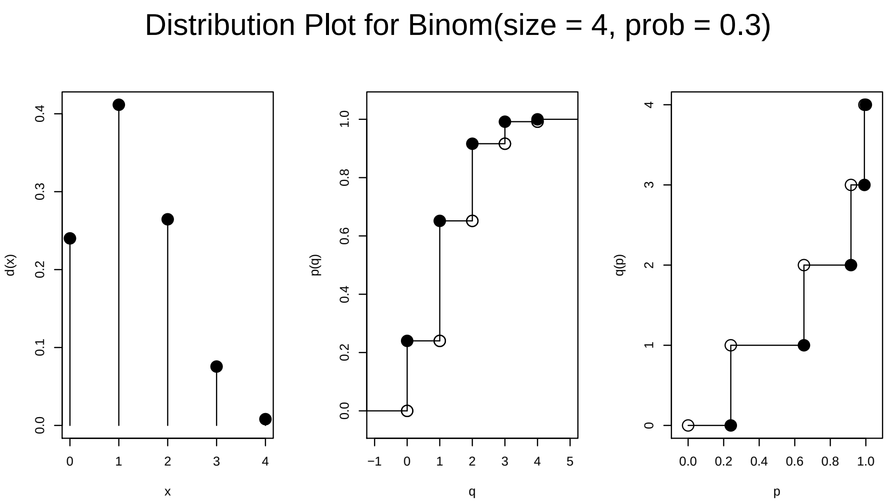
```
> plot(Binom(size = 4, prob = 0.3), withSweave = TRUE)
```



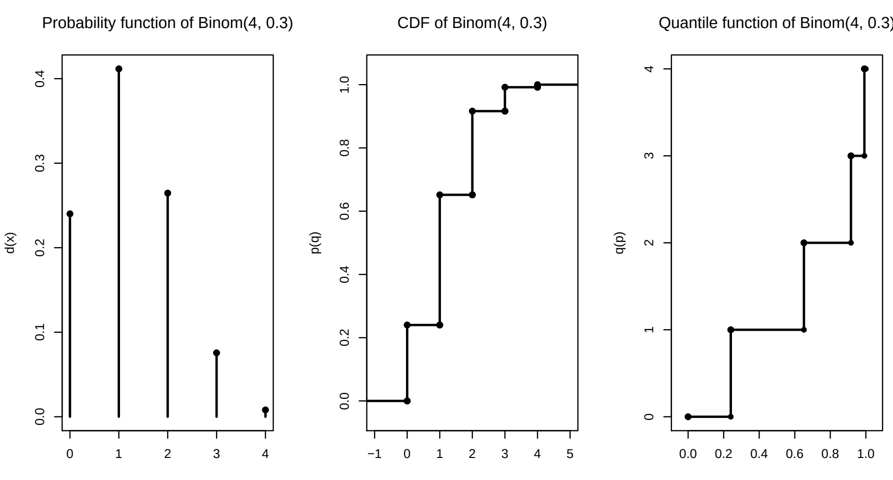

```
> plot(Binom(size = 4, prob = 0.3), do.points = FALSE, verticals = FALSE,
+       withSweave = TRUE)
```



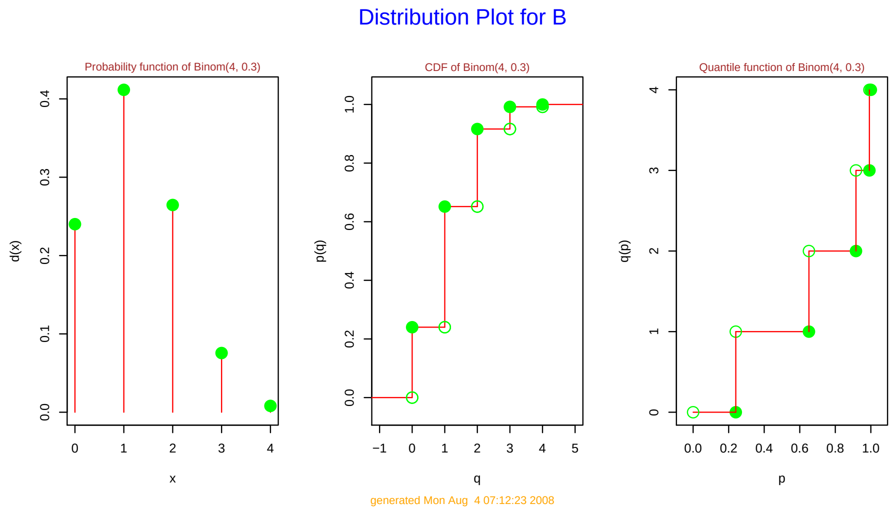
```
> plot(Binom(size = 4, prob = 0.3), main = TRUE, inner = FALSE, cex.main = 1.6,
+       tmar = 6, withSweave = TRUE)
```



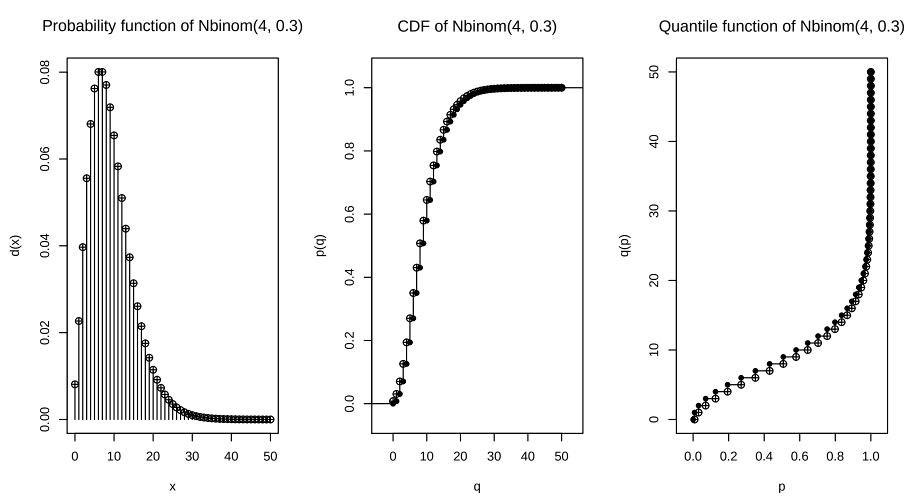

```
> plot(Binom(size = 4, prob = 0.3), cex.points = 1.2, pch = 20, lwd = 2,
+       withSweave = TRUE)
```



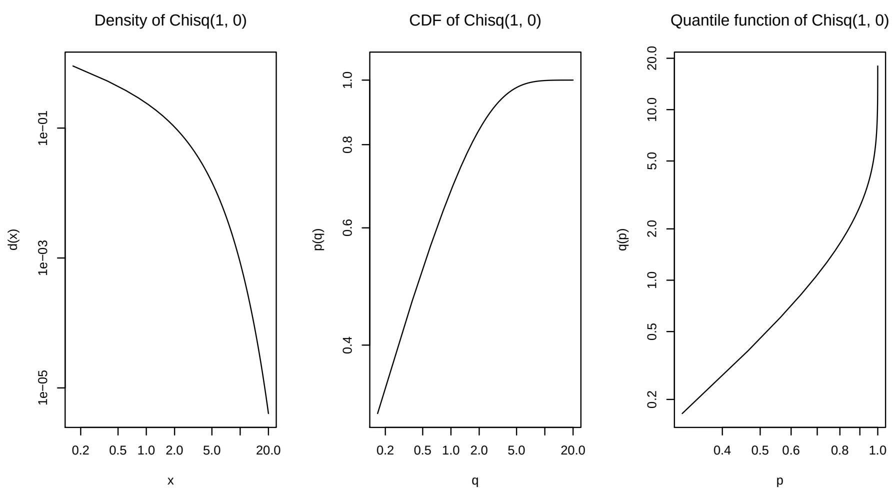
```
> B <- Binom(size = 4, prob = 0.3)
> plot(B, col="red", col.points = "green", main = TRUE, col.main="blue",
+       col.sub = "orange", sub = TRUE, cex.sub = 0.6, col.inner = "brown",
+       withSweave = TRUE)
```



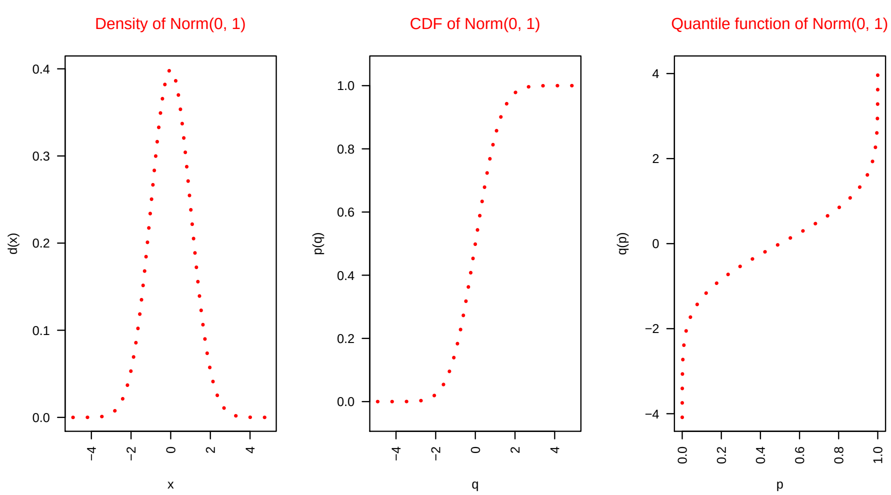

```
> plot(Nbinom(size = 4,prob = 0.3), cex.points = 1.2, pch.u = 20, pch.a = 10,
+       withSweave = TRUE)
```



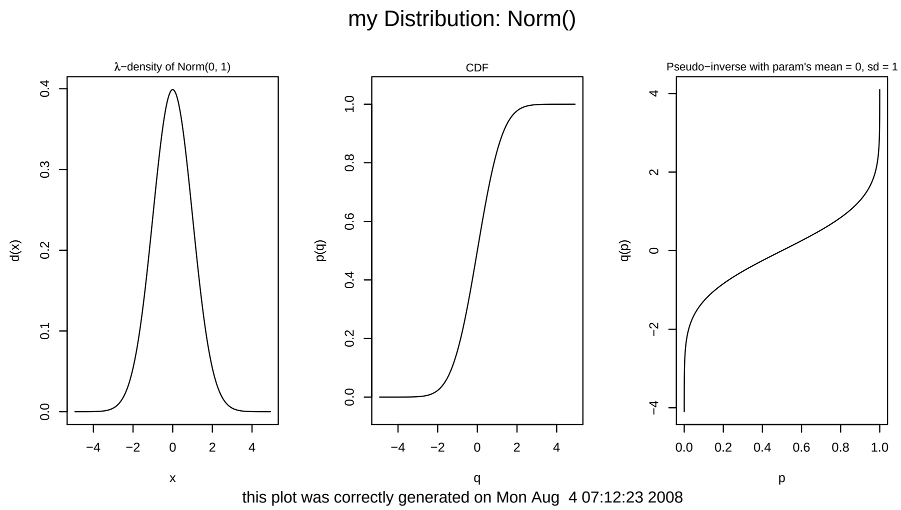
```
> plot(Chisq(), log = "xy", ngrid = 100, withSweave = TRUE)
```



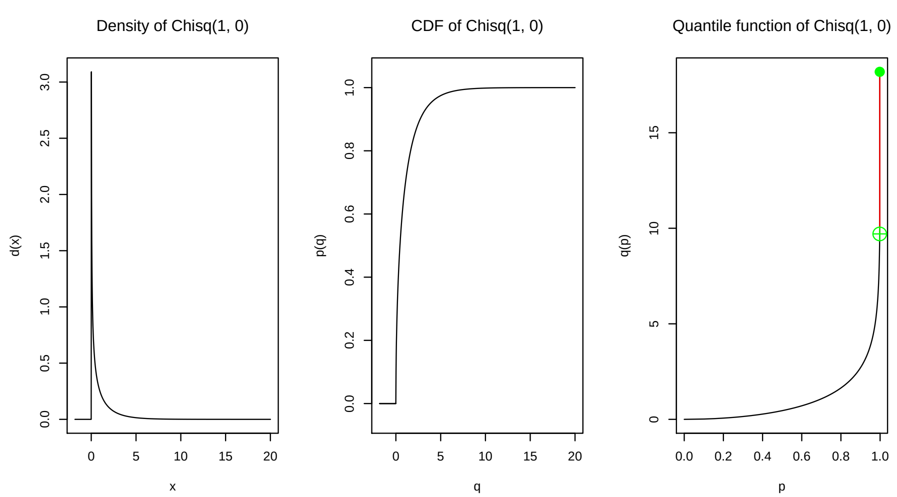

```
> plot(Norm(), lwd=3, col = "red", ngrid = 200, lty = 3, las = 2,
+       withSweave = TRUE)
```



```
> plot(Norm(), 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, withSweave = TRUE)
```




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



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)
+     print(Data(X)[1:10])
+    } else {
+     cat("\n functionality not (yet) available; ")
+     cat("you have to install package \"distrSim\" first.\n")
+     }
```

```
 [1]  1.59186275 -0.09758916 -0.77226047  1.60792363  1.53803205  1.66854417
 [7] -0.34325930  0.08114336  0.92093133  0.33362362
```

3.12 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 12.8. For data of class `Contsimulation`, the result takes a slightly different, combining evalations on ideal and real data.

3.13 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"

3.14 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.

3.15 Functionals (in package "distrEx")

3.15.1 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[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.

[8] 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  # corresponds to Norm(cond, 1)
+      N <- Norm(mean = 2)
+
+      print(E(D4, cond = 1))
+      print(E(D4, cond = 1, useApply = FALSE))
+      print(E(as(D4, "UnivariateCondDistribution"), cond = 1))
+      print(E(D4, function(x){x^2}, cond = 2))
+      print(E(D4, function(x){x^2}, cond = 2, useApply = FALSE))
+      print(E(N, function(x){x^2}))
+      print(E(as(N, "UnivariateDistribution"), function(x){x^2},
+        useApply = FALSE)) # crude Monte-Carlo
+      print(E(D4, function(x, cond){cond*x^2}, cond = 2,
+        withCond = TRUE))
+      print(E(D4, function(x, cond){cond*x^2}, cond = 2,
+        withCond = TRUE, useApply = FALSE))
+      print(E(N, function(x){2*x^2}))
+      print(E(as(N, "UnivariateDistribution"), function(x){2*x^2},
+        useApply = FALSE)) # crude Monte-Carlo
+      Y <- 5 * Binom(4, .25) - 3
+      print(Y); print(E(Y))
+     } else {
+     cat("\n functionality not (yet) available; ")
+     cat("you have to install package \"distrEx\" first.\n")
+     }
```

```
[1] 0.9999998
[1] 0.9999998
[1] 0.997028
[1] 4.999993
[1] 4.999993
[1] 4.999993
[1] 4.999182
[1] 9.999987
[1] 9.999987
[1] 9.999987
[1] 10.00531
Distribution Object of Class: AffLinLatticeDistribution
[1] 2
```

3.15.2 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`. For an object `Y` of a subclass of class union `AffLinDistribution`, we determine the variance as `Y@a^2 * 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 `Beta` (for noncentrality 0), `Binom`, `Cauchy`, `Chisq`,`Dirac`, `Exp`, `Fd`, `Gammad`, `Geom`, `Hyper`, `Logis`, `Lnorm`, `Nbinom`, `Norm`, `Pois`, `Unif`, `Td`, `Weibull`.

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

3.16 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`

3.17 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
- Cramér von Mises distance
- convex-contamination "distance" (asymmetric!) defined as

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

3.18 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.

3.18.1 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".

3.18.2 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"`.

3.18.3 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^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 New 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 7.

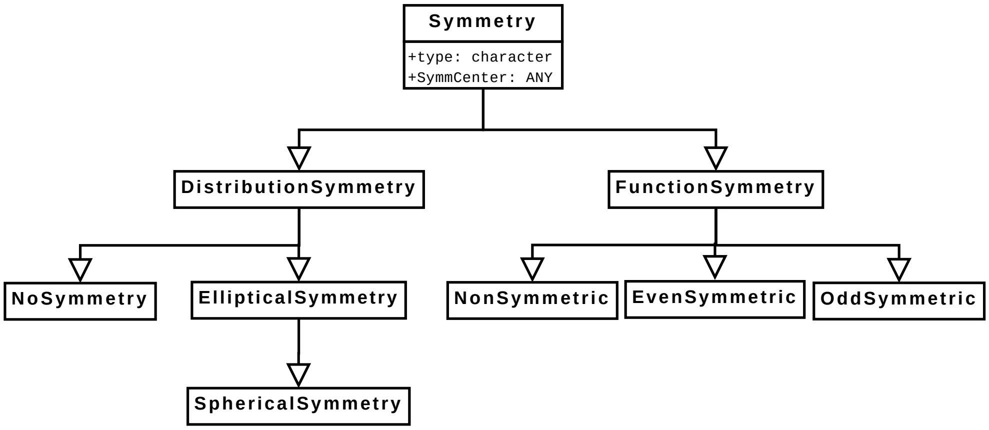


Figure 7: Inheritance relations and slots of the corresponding (sub-)classes for `Symmetry` where we do not repeat inherited slots

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 8. But it would also be possible to derive subclasses for other (e.g., semi-parametric) 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].

4.2.1 `ParamFamily` and friends

4.2.2 Class `ParamFamParameter`

In particular, to cope with partitioned parameters (with main and nuisance part) and to allow for smooth transformations of the parameter, we introduce a sub-class `ParamFamParameter` to our class `Parameter` from subsection **??**. It has slots **main**, **nuisance**, and **trafo**.

Implementaion of partial influence curves (pICs) is done via slot **trafo**. pICs arise if we are only interested is some possibly lower dimensional, smooth (not necessarily linear or even coordinate-wise) aspect/transformation τ of the whole parameter θ.

The nuisance part has its own slot. It arises if there is some unknown parameter, the value of which is of secondary interest compared to the main aspect of the parameter, but knowledge of which is necessary for estimation or testing (e.g. an unknown scale parameter for confidence intervals about an interesting location parameter).

To get a coherent implementation of the two concepts, nuisance and transformation τ, we make the following convention:

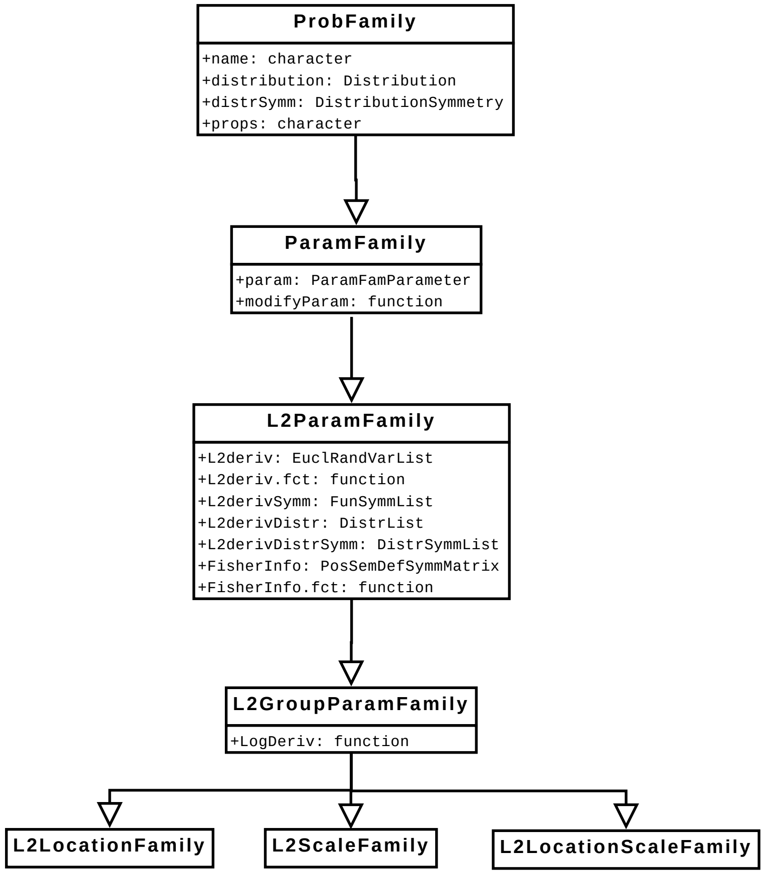


Figure 8: Inheritance relations and slots of the corresponding (sub-)classes for `ProbFamily` where we do not repeat inherited slots

The full parameter θ is first split up coordinate-wise into a main parameter θ' and a nuisance parameter θ''. Without loss of generality, we restrict the transformation τ only to act on the main parameter θ'; this is no real restriction, because if we want to transform the whole parameter, we only need to assume that nuisance parameter θ'' be of length 0.

To the implementation: As we have given the same name `trafo` to several different objects, we have to be a little careful about what type of `trafo` we are talking about; we will try to be clear about this.

Slot `trafo` of an object of class `ParamFamParameter` can either contain a (constant) matrix D_θ or a function $\tau\colon \Theta \to \tilde{\Theta}, \qquad \theta \mapsto \tau(\theta)$ mapping the main parameter θ' to some range $\tilde{\Theta}$.

If *slot value* `trafo` is a function, besides the function value $\tau(\theta)$, it will also return the corresponding derivative matrix $D_\theta = \frac{\partial}{\partial\theta}\tau(\theta)$. 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, we interpet $\tau(\theta)$ as the linear mapping $\tau(\theta) = D\,\theta$.

For several classes, trafo also denotes the accessor function to slots containing information about the transformation. According to the signature, this *method* trafo will return different return value types. For signature

Estimate,missing: it will return a list with entries fct, the function τ, and mat, the matrix $\frac{\partial}{\partial\theta}\tau(\theta)$. Function τ then returns the list list(fval, mat) mentioned above

ParamFamParameter,missing: it will just return the corresponding matrix.

ParamFamily,missing: being just a wrapper to signature ParamFamParameter,missing, it will do the same.

ParamFamily,ParamFamParameter: it will return the same as signature Estimate,missing.

4.3 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 11. The bias and norm classes given in Figure 9 and Figure 10, respectively, occur as slots of the risk classes.

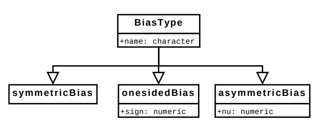


Figure 9: Inheritance relations and slots of the corresponding (sub-)classes for BiasType where we do not repeat inherited slots

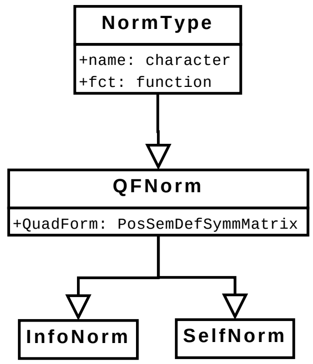


Figure 10: Inheritance relations and slots of the corresponding (sub-)classes for `NormType` where we do not repeat inherited slots

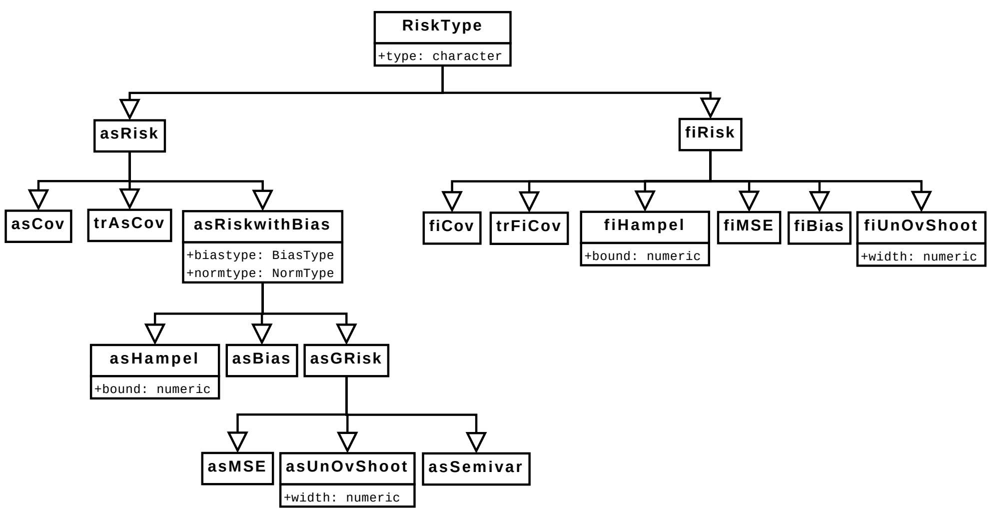


Figure 11: Inheritance relations and slots of the corresponding (sub-)classes for `RiskType` where we do not repeat inherited slots

4.4 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.

```
> have.distrMod <- suppressWarnings(require("distrMod"))
> if (have.distrMod){
+     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 âĂIJEstimateâĂİ
generated by call
  MCEstimator(x = x, ParamFamily = G, criterion = negLoglikelihood)
samplesize:   50
estimate:
    scale     shape
0.4969899 3.2097398
Criterium:
negLoglikelihood
        59.53214
```

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.

```
> if (have.distrMod){
+     print(MLEstimator(x = x, ParamFamily = G))
+     MDEstimator(x = x, ParamFamily = G, distance = KolmogorovDist)
+ }
```

```
Evaluations of Maximum likelihood estimate:
-------------------------------------------
An object of class âĂIJEstimateâĂİ
generated by call
  MLEstimator(x = x, ParamFamily = G)
samplesize:   50
estimate:
     scale       shape
  0.4969899   3.2097398
 (0.1024933) (0.6115292)
asymptotic (co)variance:
           scale     shape
scale  0.5252438 -2.895224
shape -2.8952238 18.698397
Criterium:
negative log-likelihood
               59.53214
Evaluations of Minimum Kolmogorov distance estimate:
----------------------------------------------------
An object of class âĂIJEstimateâĂİ
generated by call
  MDEstimator(x = x, ParamFamily = G, distance = KolmogorovDist)
samplesize:   50
estimate:
    scale     shape
0.4026151 3.8822864
Criterium:
Kolmogorov distance
         0.06988294
```

The results of these computations are objects of the S4-class `MCEstimate` which inherits from the S4-class `Estimate`. The definitions are given in Figure 12. As an example of how a transformation τ may enter, consider the following reparametrization in our example: If $\theta =$ (`shape`, `scale`), let $\tau(\theta) = (\theta_2^{-1}, \theta_1) =$ (`rate`,`shape`), the parametrization as in B. Ripley's `fit.distr` (confer [8]) from package `"MASS"`. This gives the following transformation

```
> if (have.distrMod){
+ mtrafo <- function(x){
+      nms0 <- names(c(main(param(G)),nuisance(param(G))))
+      nms <- c("shape","rate")
+      fval0 <- c(x[2], 1/x[1])
+      names(fval0) <- nms
```

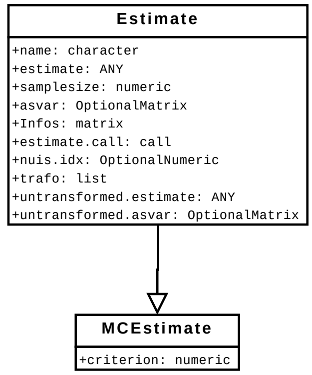


Figure 12: Inheritance relations and slots of the corresponding (sub-)classes for `Estimate` where we do not repeat inherited slots

```
+       mat0 <- matrix( c(0, -1/x[1]^2, 1, 0), nrow = 2, ncol = 2,
+                       dimnames = list(nms,nms0))
+       list(fval = fval0, mat = mat0)}
+ G2 <- G
+ trafo(G2) <- mtrafo
+ (res.Mod <- MLEstimator(x = x, ParamFamily = G2))
+ }
```

```
Evaluations of Maximum likelihood estimate:
-------------------------------------------
An object of class âĂIJEstimateâĂİ
generated by call
  MLEstimator(x = x, ParamFamily = G2)
samplesize:   50
estimate:
     shape       rate
  3.2097398   2.0121132
 (0.6115292) (0.4149543)
asymptotic (co)variance:
         shape      rate
shape 18.69840 11.721602
rate  11.72160  8.609356
untransformed estimate:
     scale       shape
  0.4969899   3.2097398
 (0.1024933) (0.6115292)
```

```
asymptotic (co)variance of untransformed estimate:
           scale     shape
scale  0.5252438 -2.895224
shape -2.8952238 18.698397
Transformation of main parameter:
function(x){
     nms0 <- names(c(main(param(G)),nuisance(param(G))))
     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)}
Trafo / derivative matrix:
        scale shape
shape  0.0000     1
rate  -4.0486     0
Criterium:
negative log-likelihood
               59.53214
```

For comparison, consider

```
> if (have.distrMod){
+ require(MASS)
+ (res.MASS <- fitdistr(x, "gamma"))
+ }
```

```
     shape       rate
  3.2099685   2.0122386
 (0.6115739) (0.4149786)
```

The detailedness of the output of these result classes by `print` and `show` can be controlled via the global option `show.details` (see subsection 5.3). In case of `print`, this option (as well as option `digits` from package `base`) may also be set temporarily by transferring a corresponding `show.details`- (resp. `digits`-) argument in the call to `print`. As an example, consider

```
> if (have.distrMod){
+ print(res.Mod, digits = 3, show.details = "minimal")
+ print(res.MASS, digits = 3)
+ }
```

```
Evaluations of Maximum likelihood estimate:
-------------------------------------------
   shape    rate  
  3.210    2.012  
 (0.612)  (0.415) 
   shape    rate  
  3.210    2.012  
 (0.612)  (0.415) 
```

The actual behavior of `print` and `show` for various option settings is described in detail in the corresponding `<classname>-class`-help file, e.g. by inspecting `?Estimate-Class`.

4.5 Confidence Intervals

For a unified interface to confidence intervals, we have overloaded method `confint` from package `stats`. It now dispatches according to arguments `object` and `method`, where upto now we only use the first one and have only implemented methods for the second being `missing`. Later on, for one type of estimator in the first argument, several methods can be defined like asymptotic, bootstrap-based, simulation-based, Cornish-Fisher-based and so on. For the time being, we have implemented methods for asymptotic CLT-based confidence intervals for objects of class `Estimate`. The default method (signature `object=ANY,method=missing`) just uses `stats::confint`. Our method for class `Estimate` uses the asymptotic (co)variance (if present in slot `asvar`) of the estimator, and then builds a symmetric CLT (i.e. Gaussian) confidence interval about the estimator value. In our example this gives

```
> (ci.Mod <- confint(res.Mod))

A[n] asymptotic (CLT-based) confidence interval:
         2.5 %   97.5 %
shape 2.011165 4.408315
rate  1.198818 2.825409
Type of estimator: Maximum likelihood estimate:
samplesize:   50
Call by which estimate was produced:
MLEstimator(x = x, ParamFamily = G2)
Transformation of main parameter by which estimate was produced:
function(x){
     nms0 <- names(c(main(param(G)),nuisance(param(G))))
     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)}
Trafo / derivative matrix at which estimate was produced:
        scale shape
shape  0.0000     1
rate  -4.0486     0
```

For comparison, we add a convenient (albeit wrong) S3-method for `vcov` —wrong as in general the covariance matrix of an estimator will not be diagonal; but for our coordinate-wise confidence intervals this does not matter, as only the diagonal entries determine the confidence bounds.

```
> if (have.distrMod){
+ vcov.fitdistr <- function(object, ...){
+      v<-diag(object$sd^2)
+      rownames(v) <- colnames(v) <- names(object$estimate)
+      v}
+ (ci.MASS <- confint(res.MASS))
+ }

         2.5 %   97.5 %
shape 2.011306 4.408631
rate  1.198896 2.825582
```

or again, shortly:

```
> if (have.distrMod){
+ print(ci.Mod, digits = 3, show.details = "minimal")
+ print(ci.MASS, digits = 3)
+ }

A[n] asymptotic (CLT-based) confidence interval:
      2.5 % 97.5 %
shape  2.01   4.41
rate   1.20   2.83
      2.5 % 97.5 %
shape  2.01   4.41
rate   1.20   2.83
```

`confint` will return objects of (S4-)class `Confint`, so that the user can easily manipulate the output within R. The class structure is depicted in Figure 13.

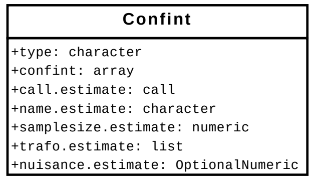


Figure 13: Inheritance relations and slots of the corresponding (sub-)classes for `Confint` where we do not repeat inherited slots

4.6 Masking

To ensure positive (semi)-definiteness of matrices like covariances, we have introduced classes `PosSemDefSymmMatrix` and `PosDefSymmMatrix`; for these the square root can be defined. More specifically we have overloaded function `sqrt` for these matrix classes; to do so, we had to mask function `sqrt` from package `"base"`. Similarly, in order to only use `ginv` when really needed, we have overloaded function `solve` (also from package `"base"`). For singular matrices, we now use `ginv`, respetively an `eigen` decomposition based procedure for classes `PosSemDefSymmMatrix` and `PosDesSymmMatrix`. Both `sqrt` and `solve` retain their semantics and can be used without change as before in the cases where this has already been possible. For our confidence intervals, we had to produce a "new" signature for function `confint`. This is done by a wrapper masking the original `confint` function from package `"stats"`, but which then defaults to `stats::confint`, so calls to the "old" `confint` method should not be affected. The user can also inspect this intentional maskings by `distrModMASK()`.

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`, $10^{\mathrm{RtoDPQ.e}}$ 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 ε = 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; in convolving discrete distributions, shall support points with distance smaller than DistrResolution be collapsed; default value: 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.

- 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`.

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` controls the detailedness of output by methods `print` and `show`. Allowed values are `"maximal"`, `"medium"`, and `"minimal"`; defaults to ”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 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>)`.

7 System/version requirements, license, etc.

7.1 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.

7.2 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.

7.3 Dependencies

In package `"distr"`, from version 2.0, we make use of `D1ss` from Martin Mächler's package `"sfsmisc"`. In package `"distrEx"`, we need Alec Stephenson's package `"evd"` for the extreme value distributions implemented therein. 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`.

7.4 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`

8 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 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 12.7.

9 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`.

10 Odds and Ends

10.1 What should be done and what we could do —for version >`2.0`

- 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

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

- multivariate distributions
- conditional distributions
- copula

11 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.

12 Examples

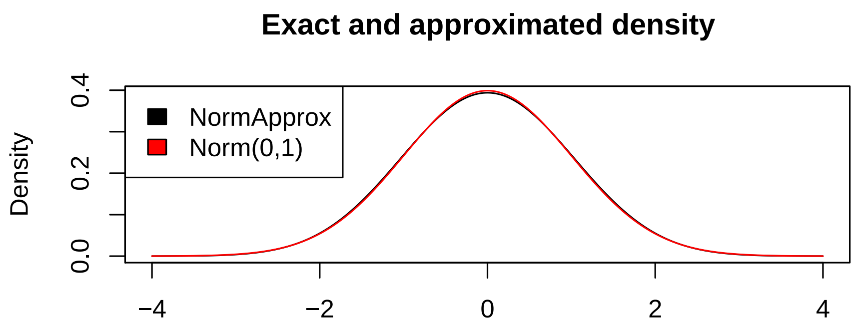
12.1 12-fold convolution of uniform $(0,1)$ variables

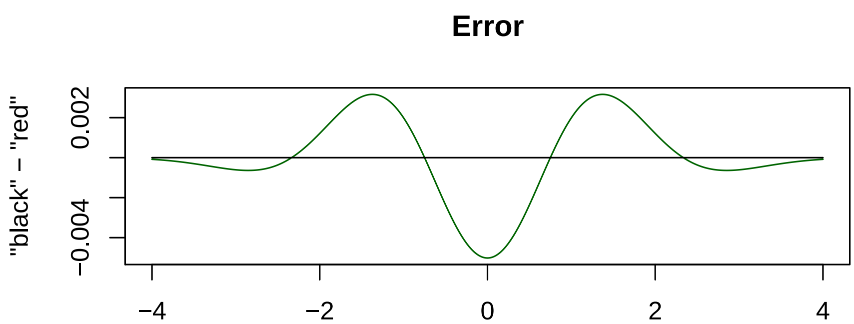
Code also available under

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)
> 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("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))
> par(opar)
```





12.2 Comparison of exact convolution to FFT for normal distributions

Code also available under

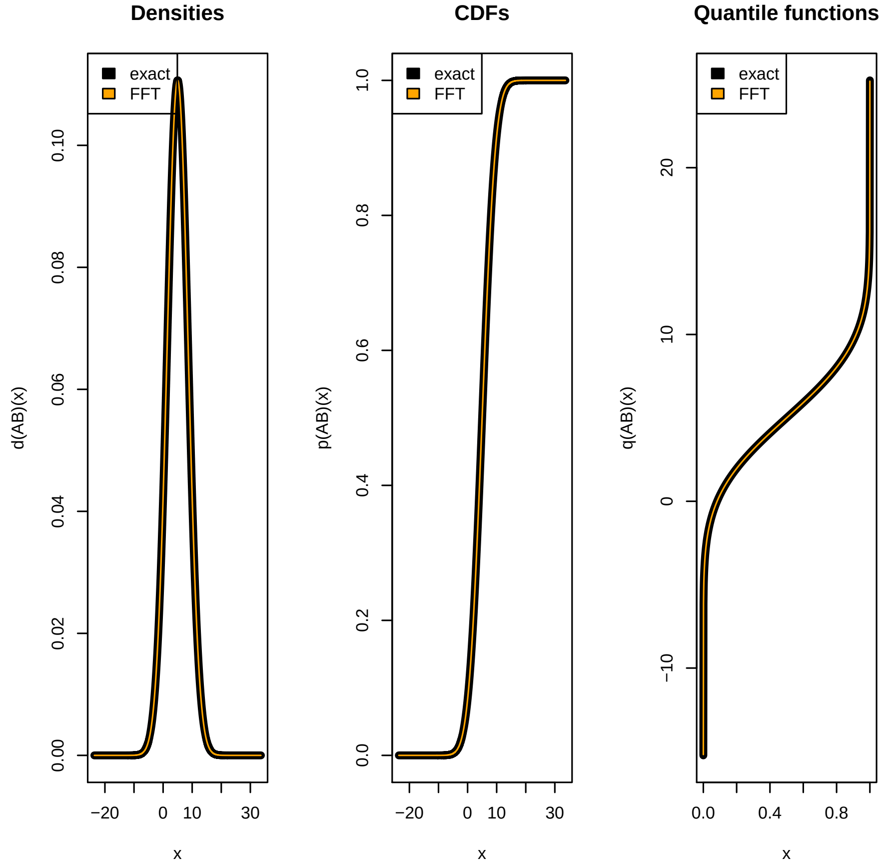
> http://www.uni-bayreuth.de/departments/math/org/
> /mathe7/DISTR/ConvolutionNormalDistr.R

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)
> ## 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"))
> ## 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"))
> ## 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:         7.269353e-07

> # 2.03e-07
>
> ### meanwhile realized in package "distrEx"
> ### as KolmogorovDist(N1,N2)
>
> ## old distroptions
> distroptions("TruncQuantile" = oldeps)
```



12.3 Comparison of FFT to `RtoDPQ`

Code also available under

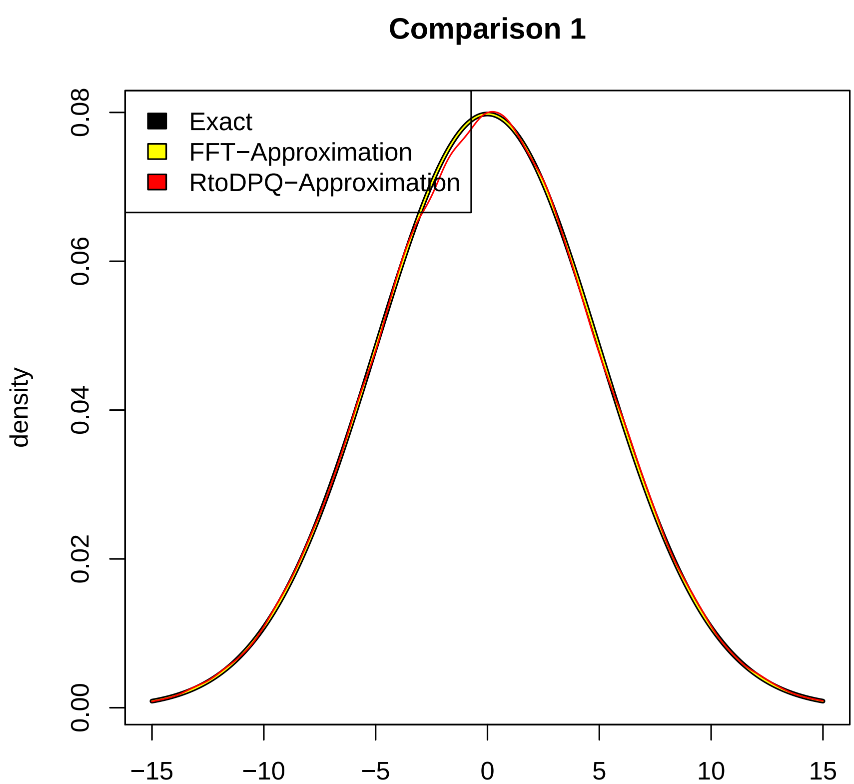
http://www.uni-bayreuth.de/departments/math/org/
/mathe7/DISTR/ComparisonFFTandRtoDPQ.R

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)
> ################################
> ## 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 2 - "Exact" Formula and RtoDPQ
> ############################################
>
> B <- Binom(size = 6, prob = 0.5) * 10
> N <- Norm()
> rnew2 <- function(n) r(B)(n) + r(N)(n)
> 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"))
```



12.4 Comparison of exact and approximate stationary regressor distribution

Code also available under

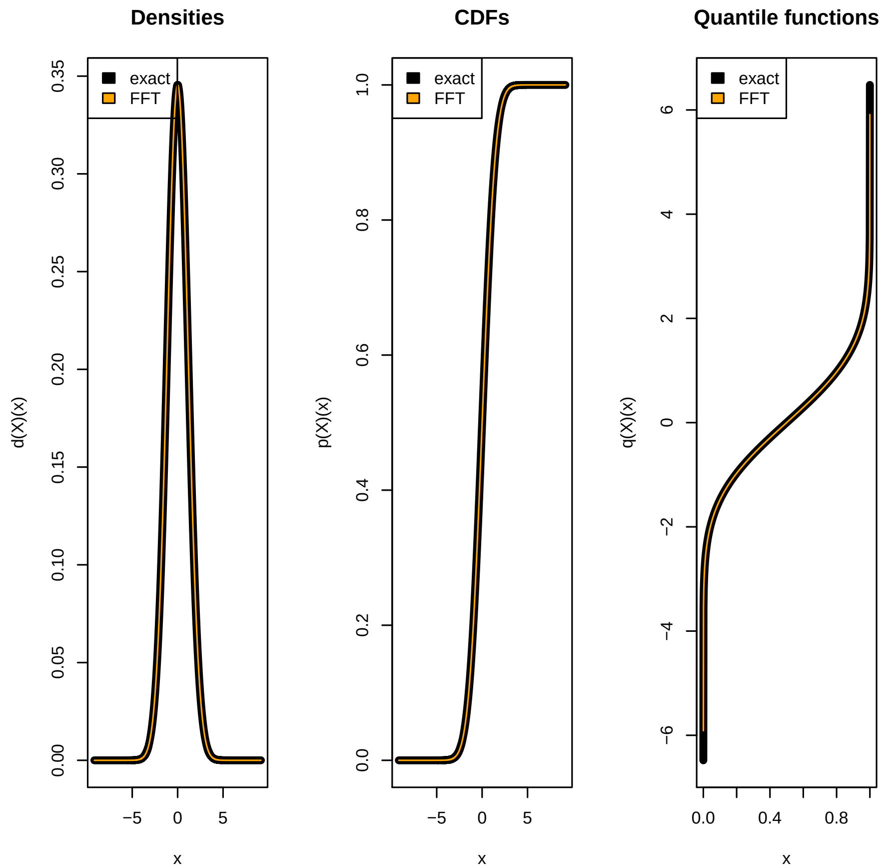
http://www.uni-bayreuth.de/departments/math/org//mathe7/DISTR/StationaryRegressorDistr.R

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)
> ## 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)))
> #############################
> ## 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"))
> ## 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

2.091529e-05 with absolute error < 5.9e-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:         1.10382e-05

> # ~2.5e-06
>
> ### meanwhile realized in package "distrEx"
> ### as KolmogorovDist(N1,N2)
>
>
> ## old distroptions
> distroptions("TruncQuantile" = oldeps)
```



12.5 Truncation and Huberization/winsorization

has been integrated to the package itself, see section 3.8

12.6 Distribution of minimum and maximum of two independent random variables

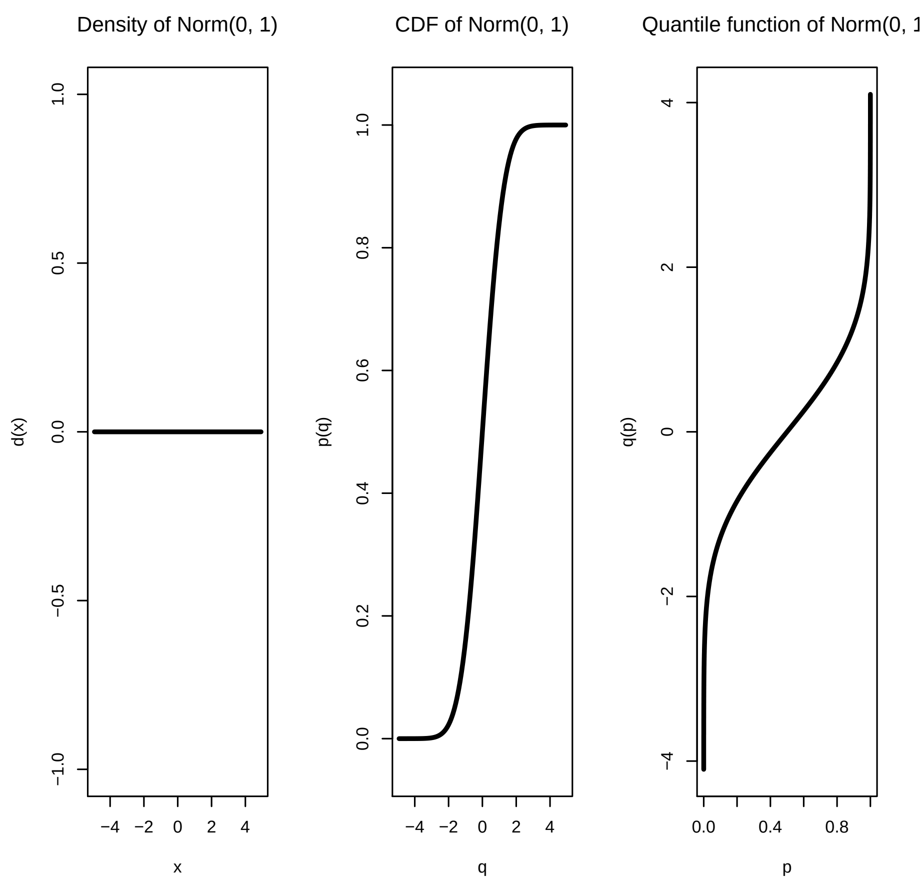
has been integrated to the package itself, see section 3.8

12.7 Instructive destructive example

Code also available under

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

```
> ##########################################################
> ## 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, withSweave = TRUE)
```



12.8 A simulation example

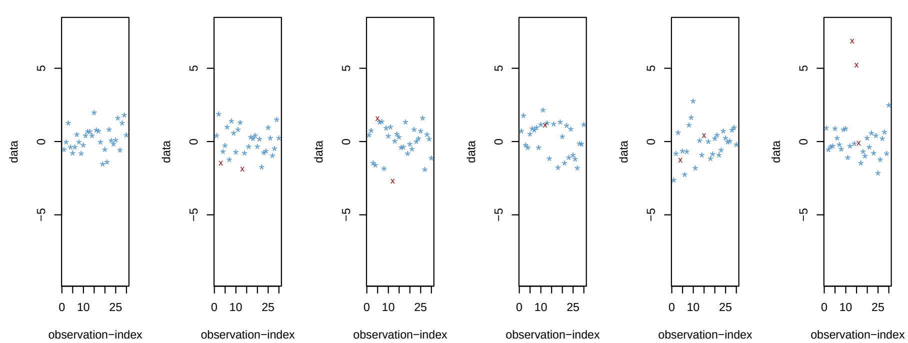
needs packages "distrSim"/"distrTEst"

Code also available under

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

```
> have.distrTEst <- suppressWarnings(require(distrTEst))
>     ### also loads distrSim
> 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:  Kind: Mersenne-Twister
       Normal Kind: Inversion
       first 6 numbers:  1347491963         0788589505        -1299697408
```

```
                              1126955172          1649141426          1204025957
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
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
```

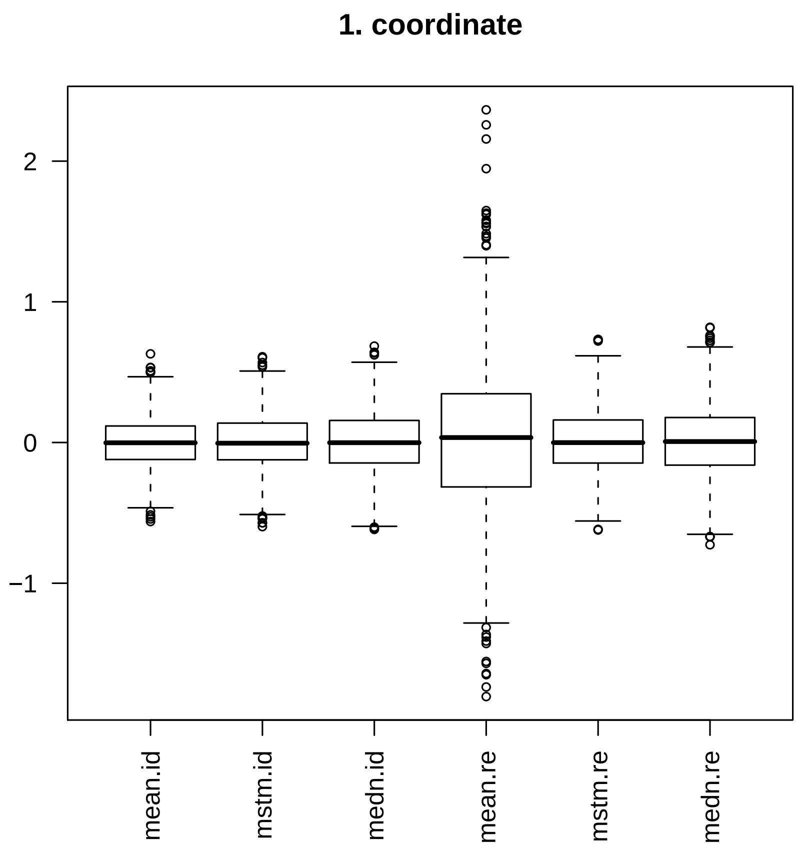



```
> have.distrTEst <- suppressWarnings(require("distrTEst"))
> if (have.distrTEst)
+    { psim <- function(theta,y,m0){
+        mean(pmin(pmax(-m0, y - theta), m0))
+        }
+      mestimator <- function(x, m = 0.7) {
+        uniroot(f = psim,
+                lower = -20,
+                upper = 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)
+
+       print(elist)
+       summary(elist)
+       plot(elist, cex = 0.7, las = 2)
+     } 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   0.259  0.109  0.103  0.163 -0.130 ...
 $ mean.re: num   0.259  0.408  0.086  1.454 -1.154 ...
----------------------------------
An Evaluation Object
estimator: mestimator
Result: 'data.frame':        1000 obs. of  2 variables:
 $ mstm.id: num   0.222  0.136  0.200  0.386 -0.181 ...
 $ mstm.re: num   0.2221  0.0455  0.2671  0.6004 -0.2145 ...
----------------------------------
An Evaluation Object
```

```
estimator: median
Result: 'data.frame':        1000 obs. of  2 variables:
 $ medn.id: num   0.1405  0.2698  0.2363  0.4760 -0.0661 ...
 $ medn.re: num   0.1405  0.2482  0.3147  0.8155 -0.0661 ...
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.   :-0.5616794   Min.   :-1.80570
 1st Qu.:-0.1203394   1st Qu.:-0.31472
 Median :-0.0019811   Median : 0.03525
 Mean   : 0.0007992   Mean   : 0.02197
 3rd Qu.: 0.1171683   3rd Qu.: 0.34652
 Max.   : 0.6301414   Max.   : 2.36457
----------------------------------
name of Evaluation: object
estimator: mestimator
Result:
    mstm.id            mstm.re
 Min.   :-0.597801   Min.   :-0.6218213
 1st Qu.:-0.122779   1st Qu.:-0.1459503
 Median :-0.004868   Median :-0.0007455
 Mean   : 0.004798   Mean   : 0.0117289
 3rd Qu.: 0.137703   3rd Qu.: 0.1601655
 Max.   : 0.609735   Max.   : 0.7330214
----------------------------------
name of Evaluation: object
estimator: median
Result:
    medn.id            medn.re
 Min.   :-0.617694   Min.   :-0.726790
 1st Qu.:-0.145335   1st Qu.:-0.160639
 Median :-0.001357   Median : 0.006762
 Mean   : 0.003618   Mean   : 0.011830
 3rd Qu.: 0.156489   3rd Qu.: 0.177588
 Max.   : 0.685331   Max.   : 0.819182
```



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   0.222  0.136  0.200  0.386 -0.181 ...
 $ mstm.re: num   0.2221  0.0455  0.2671  0.6004 -0.2145 ...
```

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.   :-0.5616794   Min.   :-1.80570
 1st Qu.:-0.1203394   1st Qu.:-0.31472
 Median :-0.0019811   Median : 0.03525
 Mean   : 0.0007992   Mean   : 0.02197
 3rd Qu.: 0.1171683   3rd Qu.: 0.34652
 Max.   : 0.6301414   Max.   : 2.36457
----------------------------------
name of Evaluation: object
estimator: mestimator
Result:
    mstm.id             mstm.re
 Min.   :-0.597801   Min.   :-0.6218213
 1st Qu.:-0.122779   1st Qu.:-0.1459503
 Median :-0.004868   Median :-0.0007455
 Mean   : 0.004798   Mean   : 0.0117289
 3rd Qu.: 0.137703   3rd Qu.: 0.1601655
 Max.   : 0.609735   Max.   : 0.7330214
----------------------------------
name of Evaluation: object
estimator: median
Result:
    medn.id             medn.re
 Min.   :-0.617694   Min.   :-0.726790
 1st Qu.:-0.145335   1st Qu.:-0.160639
 Median :-0.001357   Median : 0.006762
 Mean   : 0.003618   Mean   : 0.011830
 3rd Qu.: 0.156489   3rd Qu.: 0.177588
 Max.   : 0.685331   Max.   : 0.819182
```

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. . .

12.9 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 12.5 and 12.6, 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)
+    {
+      # Example
```

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

```
[1] 3
[1] 1.5
[1] 0.9999998
[1] 0.9999944
```

12.10 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.
From version 1.9 this is integrated to package "distr".

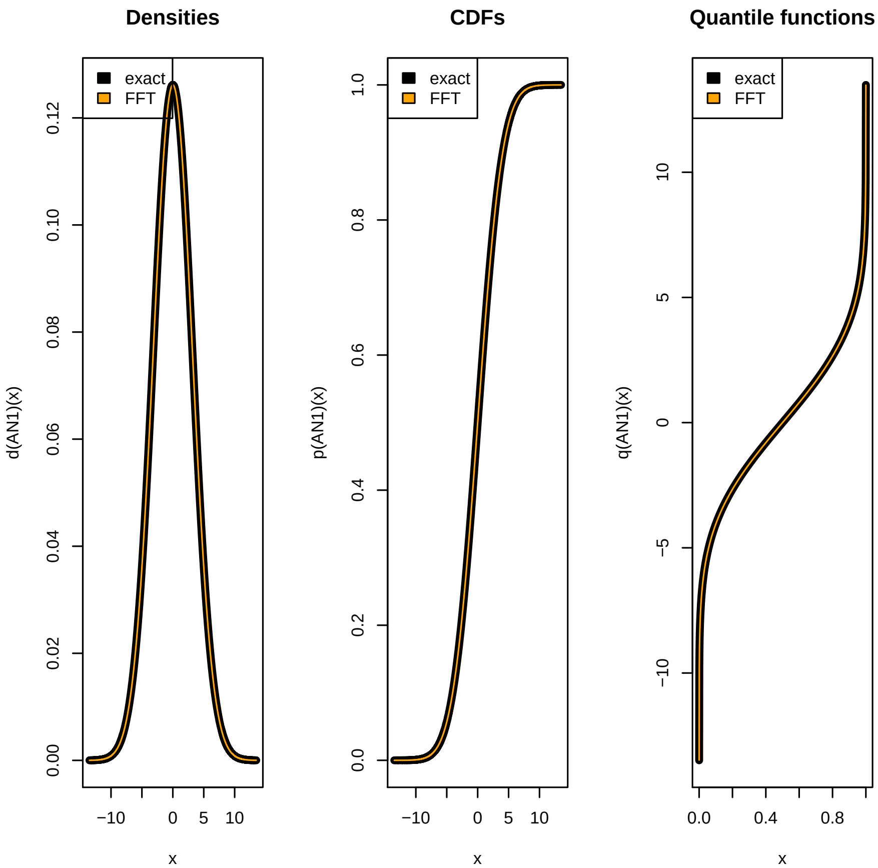
```
> ##########################################################
> ## Demo: n-fold convolution of absolutely continuous
> ##       probability distributions
> ##########################################################
> require(distr)
> if(!isGeneric("convpow"))
+     setGeneric("convpow",
+     function(D1,...) 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=""))
+             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])
+
+     ##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))
+ })

[1] "convpow"

> ## initialize a normal distribution
> A <- Norm(mean=0, sd=1)
> ## convolution power
> N <- 10
> ## 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"))
```



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/ 7

[2] Chambers J.M. (1998): *Programming with data. A guide to the S language*. Springer. http://cm.bell-labs.com/stat/Sbook/index.html 7

[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 68

[4] Kohl M. (2005): *Numerical Contributions to the Asymptotic Theory of Robustness*. Dissertation, Universität Bayreuth. See also http://stamats.de/ThesisMKohl.pdf 53, 55

[5] Kohl M., Ruckdeschel P. and Stabla T. (2004): General Purpose Convolution Algorithm for Distributions in S4-Classes by means of FFT. unpublished manual 7, 27, 68

[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. 47

[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. 70

[8] Venables, W. N. and Ripley, B. D. (2002): *Modern Applied Statistics with S. Fourth Edition.* Springer, New York. 58

[9] 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 4