

S4 Classes for Distributions—a manual for packages `distr`, `distrSim`, `distrTEst` version 1.6

Peter Ruckdeschel
Matthias Kohl
Thomas Stabla
Florian Camphausen

Mathematisches Institut
Universität Bayreuth
D-95440 Bayreuth
Germany

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

21st October 2005

Abstract

"`distr`" is a package for R from version 1.8.1 onwards that is distributed under GPL license 2.0. Its own current version is 1.6. 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 methods `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 "`base`" package are implemented as subclasses of either `AbscontDistribution` or `DiscreteDistribution`, which themselves are again subclasses of `UnivariateDistribution`. By means of these classes, we may automatically generate new objects of these classes for the laws of r.v.'s under standard mathematical univariate transformations and under convolution of independent r.v.'s.

From version "`distr`" has been split up into the smaller packages "`distr`" (only distribution-classes and -methods), "`distrSim`"

(standardized treatment of simulations (also under contaminations)) and "distrTEst" (classes and methods for evaluations of statistical procedures on such simulations).

The latter two of them require package "setRNG" by Paul Gilbert to be installed from CRAN.

Contents

0	Motivation	2
1	Concept	4
2	Organization in classes	5
3	Methods	9
4	Options	13
5	System/version requirements	14
6	Details to the implementation	15
7	A general utility	15
8	Odds and Ends	16
9	Examples	17
10	Acknowledgement	40
	References	40

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 $f: \mathbb{R}^2 \rightarrow \mathbb{R}$ 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
plot(Z)
p(Z)(0.4)
q(Z)(0.3)
Zs<-r(Z)(1000)
Zs
```

Comment:

Let `N` an object of class “Norm” with parameters `mean=2`, `sd=1.3` and let `P` an object of class “Pois” with parameter `lambda=1.2`. Assigning to `Z` the expression `2*N+3+P`, a new distribution object is generated —of class “AbscontDistribution” in our case— so that identifying `N`, `P`, `Z` with random variables distributed according to `N`, `P`, `Z`, $\mathcal{L}(Z) = \mathcal{L}(2 * N + 3 + P)$, and writing `p(Z)(0.4)` we get $P(Z \leq 0.4)$, `q(Z)(0.3)` the 30%-quantile of `Z`, and with `r(Z)(1000)` we generate 1000 pseudo random numbers distributed according to `Z`, while the `plot` command generates figure 1.

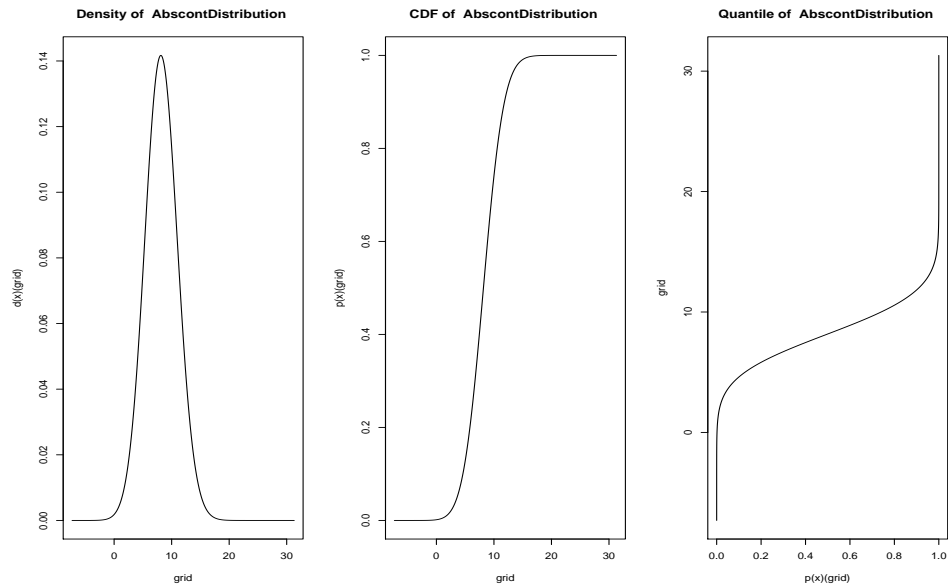


Figure 1: density, c.d.f. and quantile function of Z

1 Concept

In developing the class 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". The largest effort will in fact be the documentation. . .

We also wanted to preserve naming and notation from R-"base" 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 snippets 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. Doing so, we place

ourselves somewhere between pure object orientation where methods would be *slots* —in the language of the S4-concept, confer Chambers (1998)— and S4 where methods ”live their own life” apart from the classes: We distinguish constitutive methods, in our case `r`, `d`, `p`, and `q`, which should be regarded as integral part of a distribution object and thus be treated just as slots, and ”normal” methods which follow the S4-paradigm. To use H. Bengtsson’s terminology, confer Bengtsson (2003), we use COOP¹-style for constitutive methods, and FOOP²-style for ”normal” methods.

To illustrate how “constitutive” `r`, `d`, `p`, and `q` are, think of how to conceive a general convolution algorithm. . .

Contrary to the standard R-functions like `rnorm` we only permit length 1 for parameters like `mean`, because we see the objects as implementations of univariate random variables, for which vector-valued parameters make no sense; rather one could gather several objects with possibly different parameters to a vector/list of distributions. Of course, the original functions `rnorm` etc. remain unchanged and still allow for vector-valued parameters.

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.

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 realization space and the constitutive slots `r`, `d`, `p`, and `q`.

¹class-object-orientated programming, as e.g. in C++

²function-object-orientated programming, as in the S4-concept

2.1.1 Subclasses

To begin with, we limit ourselves to univariate distributions giving the S4-class `UnivariateDistribution`, and as typical subclasses, we introduce classes for absolutely continuous and discrete distributions—`AbscontDistribution` and `DiscreteDistribution`. The latter has a slot `support`, a vector containing the support of the distribution, which is truncated to the lower/upper `TruncQuantile` in case of an infinite support. `TruncQuantile` is a global option of “distr” described in section 4.

As subclasses of these two classes, we have implemented all parametric families which already exist in the “base” 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 in figure 2. Operations to automatically generate new slots `r`, `d`, `p`, and `q`—induced by mathematical transformations—perhaps provide the most powerful use of our package. This is discussed in some detail in subsection 3.

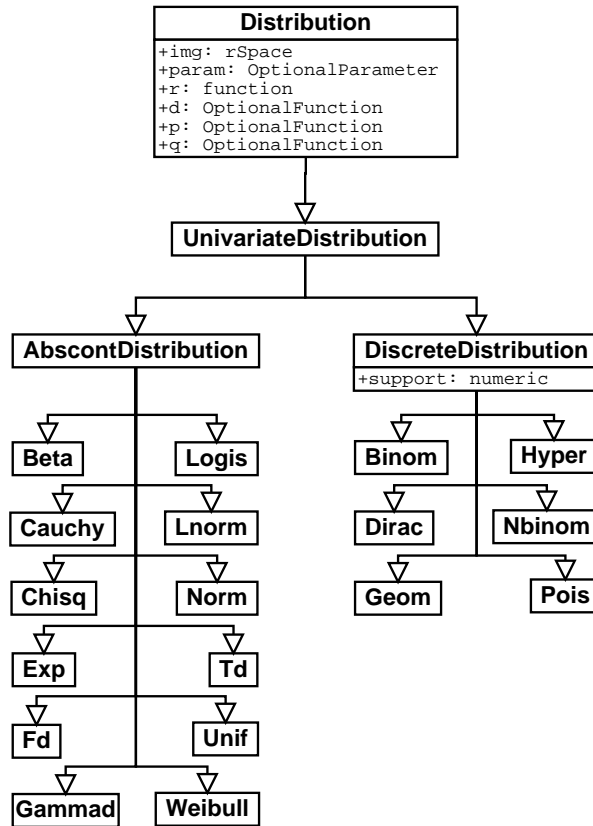
2.1.2 Parameter classes

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 3 where we do not repeat inherited slots. The most important set to be used as parameter domain/sample space (realization 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 4.

2.2 Simulation classes

From version 1.6 on we have reorganized our package. The classes and methods of this subsection are now available in the new package



slot param is filled with an object of the corresponding Parameter class

Figure 2: Inheritance relations and slots of the corresponding (sub-)classes for `Distribution` where we do not repeat inherited slots

“`distrSim`”. For the lack of a better place to describe these routines in some more detail, we keep this subsection here.

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.

Finally, coming from robust statistics we also consider situations where the majority of the data stems from an ideal situation/distribution

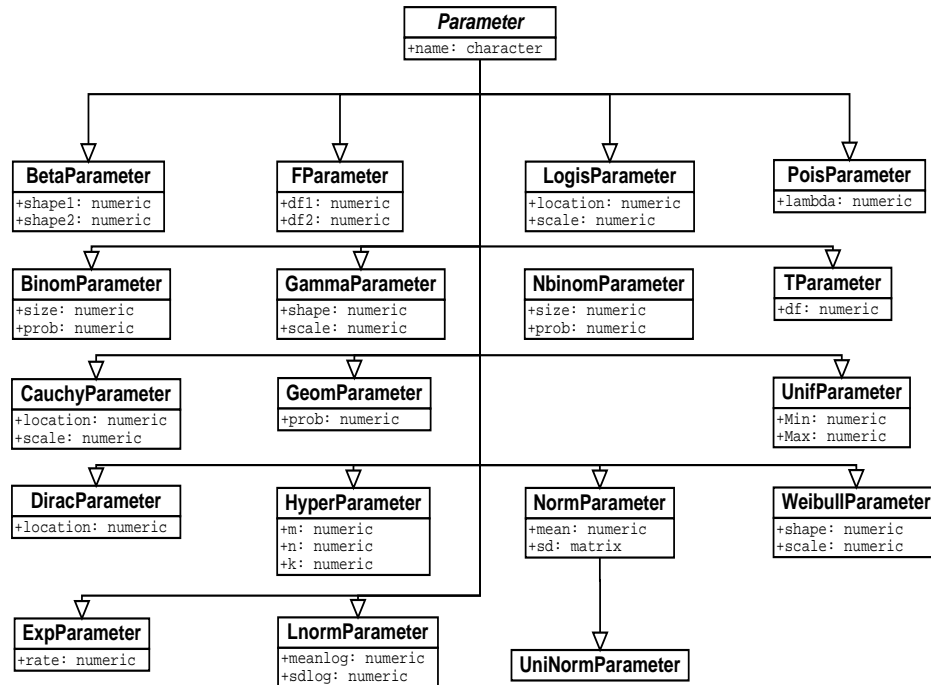


Figure 3: Inheritance relations and slots of the corresponding (sub-)classes for `Parameter`

whereas a minority comes from a contaminating source. To be able to identify ideal and contaminating observations, we also store this information in an indicator variable.

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

Schematically, the inheritance relations of class `Dataclass` as well as the slots of the corresponding (sub-)classes may be read off in figure 5 where we do not repeat inherited slots.

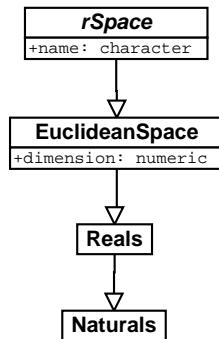


Figure 4: Inheritance relations and slots of the corresponding (sub-)classes for rSpace

2.3 Evaluation class

From version 1.6 on we have reorganized our package. The class and methods of this subsection are now available in the new package “distrTEst”. For the lack of a better place to describe these routines in some more detail, we keep this subsection here.

When investigating properties of a new procedure (e.g. an estimator) by means of simulations, one typically evaluates this procedure on a large set of simulation runs and gets a result for each run. These results are typically not available within seconds, so that it is worth storing them. To organize all relevant information about these results, we introduce a class `Evaluation` the slots of which is filled by method `evaluate` —see subsection 3.7. Schematically, the slots of this class may be read off in figure 6.

3 Methods

3.1 Affine linear transformations

We have overloaded the operators “+”, “-”, “*”, “/” such that affine linear transformations which involve only single univariate r.v.’s are available; i.e. is expressions like $Y=(3*X+5)/4$ are permitted for an object `X` of class `AbscontDistribution` or `DiscreteDistribution` (or

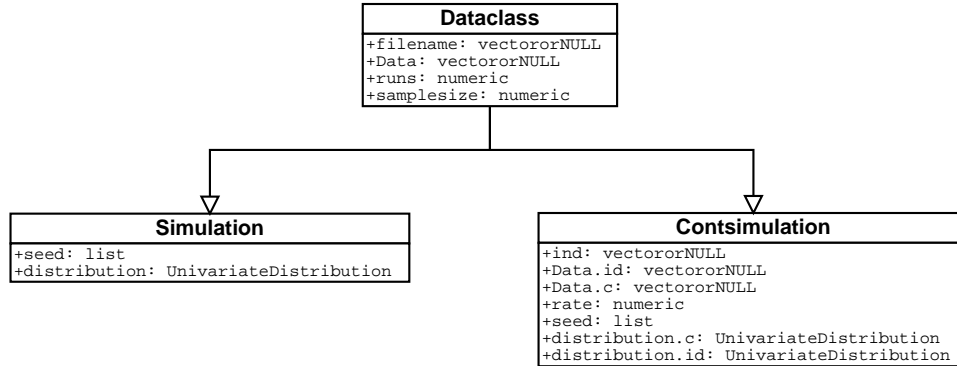


Figure 5: Inheritance relations and slots of the corresponding (sub-)classes for `Dataclass`

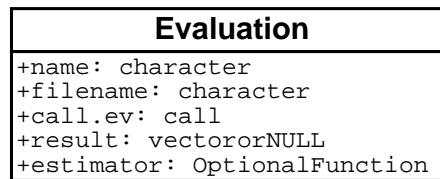


Figure 6: Slots of class `Evaluation`

some subclass), giving again an object `Y` of class `AbscontDistribution` or `DiscreteDistribution` (in general). Here the corresponding transformations of the `d`, `p`, and `q`-functions are done analytically.

3.2 The group `math` of unary mathematical operations

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

By means of `substitute`, the bodies of the `r`, `d`, `p`, `q`-slots of distri-

butions 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))`.

3.3 Construction of `d`, `p`, and `q` from `r`

In order to facilitate automatic generation of new distributions, in particular those arising as image distributions under transformations of correspondingly distributed random variables, we provide ad hoc methods that should be overloaded by more exact ones wherever possible: By means of the function `RtoDPQ` we first generate $10^{\text{RtoDPQ.e}}$, random numbers where `RtoDPQ.e` is a global option of this package and is discussed in section 4. A density estimator is evaluated along this sample, the distribution function is estimated by the empirical c.d.f. and, finally, the quantile function is produced by numerical inversion. Of course the result is rather crude as it relies on the law of large numbers only, but this way all transformations within the group `math` become available. Where laws under transformations can easily be computed exactly —as for affine linear transformations— we replace this procedure by the exactly transformed `d`, `p`, `q`-methods.

3.4 Convolution

A convolution method for two independent r.v.’s is implemented by means of explicit calculations for discrete summands, and by means of FFT³ if one of the summands is absolutely continuous. This method automatically generates the law of the sum of two independent variables/distributions X and Y of any univariate distributions —or in S4- jargon: the addition operator “+” is overloaded for two objects of class `UnivariateDistribution` and corresponding subclasses.

3.5 Overloaded generic functions

Methods `print`, `plot`, and `summary` have been overloaded for classes `Distribution`, `DataClass`, `Simulation`, `ContSimulation`, as well as `EvaluationClass` to produce “pretty” output. For `Distribution`,

³Details to be found in Kohl et al. (2005)

`plot` displays the density/probability function, the c.d.f. and the quantile function of a distribution, 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. For an object of class `EvaluationClass`, `plot` yields a boxplot of the results of the evaluation.

3.6 Simulation

From version 1.6 on we have reorganized our package. The method discussed in this subsection has been moved to the new package “`distrSim`”. For the lack of a better place to describe these routines in some more detail, we keep this subsection here.

For the classes `Simulation` and `ContSimulation`, we normally will not save the current values of the simulation, as they can easily be reproduced knowing the values of the other slots of this class. So when declaring a new object of either of the two classes, the slot `Data` will be empty (`NULL`). To fill it with the simulated values, we have to apply the method `simulate` to the object. This has to be redone whenever another slot of the object is changed. To guarantee reproducibility, we use the slot `seed`.

This slot is controlled and set through Paul Gilbert’s “`setRNG`” package. By default, `seed` is set to `setRNG()`, which returns the current “state” of the random number generator (RNG). So the user does not need to specify a value for `seed`, and nevertheless may reproduce his samples: He simply uses `simulate` to fill the `Data` slot. If the user wants to, he may also set the `seed` explicitly via the replacement function `seed()`, but has to take care of the correct format himself, confer the documentation of `setRNG`. One easy way to fill the `Data` slot of a simulation `X` with “new” random numbers is

```
seed(X) ← setRNG();
simulate(X);
```

3.7 Evaluate

From version 1.6 on we have reorganized our package. The method discussed in this subsection has been moved to the new package “`distrTEst`”.

For the lack of a better place to describe these routines in some more detail, we keep this subsection here.

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 9.8.

3.8 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 ($10^{\text{RtoDPQ.e}}$) of these.

4 Options

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

- `DefaultNrFFTGridPointsExponent`: the binary logarithm of the number of grid-points used in FFT —default 12
- `DefaultNrGridPoints`: number of grid-points used for a continuous variable —default 4096
- `DistrResolution`: the finest step length that is permitted for a grid for a discrete variable —default $1e-06$
- `RtoDPQ.e`: For simulational determination of `d`, `p` and `q`, $10^{\text{RtoDPQ.e}}$ random variables are simulated —default 5
- `TruncQuantile`: to work with compact support, random variables are truncated to their lower/upper `TruncQuantile`-quantile —default $1e-05$

All current options may be inspected by `distroptions()` and modified by `distroptions("<options-name>", <value>)`

5 System/version requirements, license, etc.

5.1 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.

5.2 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 namespaces, 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.

5.3 Dependencies

In package "distrSim", and consequently 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.

5.4 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>

6 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 Kohl et al. (2005).
- Exact convolution methods are implemented for the normal, the Poisson, the binomial the negative binomial, the Gamma (and the Exp), and the χ^2 distribution; exact formulae for scale transformations for the Exp-/Gamma-distribution
- Instances of any class transparent to the user are initialized by `<classname>([<slotname>=<value>, ...])` where except for class `DataClass` in package "distrSim" all classes have default values for all their slots; in `DataClass`, the slot `Data` has to be specified.
- As suggested in Gentleman (2003) 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 access or even modify slots `r`, `d`, `p`, and `q`, confer Example 9.7.**

7 A general utility

Following Gentleman (2003), 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`.

8 Odds and Ends

8.1 What should be done and what we could do —for version > 1.6

- application of FFT to any univariate distributions —perhaps also to be controlled by a parameter/option
- use the q-slot applied to `runif` in `simplifyr` for continuous distributions
- further exact formulae for binary arithmetic operations like `"*`
- derivation of a class `LatticeDistribution` from `DiscreteDistribution` to be able to easily apply FFT
- redo the initialize- and the math-method for discrete distributions when only slot `r` is given
- better documentation for method `evaluate` / class `Evaluation`
- adaptation of class `Evaluation` / method `evaluate` to be more flexible
- use of `initialize` in generating functions
- generating function for new distribution classes to ease inclusion of new distributions
- goodness of fit tests for distribution-objects
- use of `\S4method` in documentation
- overloading binary operators of group `Math2` for independent distributions
- defining a subgroup of `Math2` of invertible binary operators

- better use of `concept` in rd-files
- suggestion by Kouros Owzar: in case of parameters of dimension $p > 1$ —as in case of $\mathcal{N}(\mu, \sigma^2)$ — possibility to access/replace that parameter as a vector

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

- multivariate distributions
- conditional distributions
- copula

9 Examples

9.1 12-fold convolution of uniform $(0, 1)$ variables

Code available under

[http://www.uni-bayreuth.de/departments/math/org/
/mathe7/DISTR/NormApprox.R](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 Rice (1988), 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(-4, d(N)(0),
       legend = c("NormApprox", "Norm(0,1)"),
       fill = c("black", "red"))

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

par(opar)

```

9.2 Comparison of exact convolution to FFT for normal distributions

Code 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 $\mathcal{L}(A + B) = \mathcal{N}(5, 13)$ — if the second argument of \mathcal{N} is the variance

```

#####
## Demo: Convolution of normal distributions
#####
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 ← distroptions("TruncQuantile")
eps ← 1e-8
distroptions("TruncQuantile", eps)
## support of A1+B1 for FFT convolution is
## [q(A1)(TruncQuantile), q(B1)(1-TruncQuantile)]

## convolution via FFT
AB1 ← A1+B1

## plots of the results
par(mfrow=c(1,3))
low ← q(AB)(1e-15)
upp ← q(AB)(1-1e-15)
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, type = "l")
title("Densities")
legend(low, d(AB)(5), 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, type = "l")
title("Cumulative_distribution_functions")
legend(low, 1.0, legend=c("exact", "FFT"),
       fill=c("black", "orange"))

```

```

## quantile functions
x ← seq(from = eps, to = 1-eps, length = 200)
plot(x, q(AB)(x), type = "l", lwd = 5)
lines(x, q(AB1)(x), col = "orange", lwd = 1, type = "l")
  # may take some time
title("Quantile_functions")
legend(0, q(AB)(1-eps), 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")
print(dv) # 4.25e-07

## 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")
  # 2.03e-07
### old distroptions
distributions("TruncQuantile", oldeps)

```

9.3 Comparison of FFT to RtoDPQ

Code 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 transfor-

mations 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, sqrt(5))  
Y ← N1 + as(N2, "AbscontDistribution")  
  # approximated with FFT  
Z ← new("AbscontDistribution", r = rnew1)  
  # approximated 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(-15, d(X)(0),  
      legend = c("Exact", "FFT-Approximation",  
                "RtoDQP-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)
  # approximated 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(-5, d(Y)(30),
       legend = c("Exact", "RtoDQP-Approximation"),
       fill = c("black", "red"))

```

9.4 Comparison of exact and approximate stationary regressor distribution

Code 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.

```

#####
## Demo: Stationary Regressor Distribution of an AR(1)
##      Process

```

```
#####
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 to an absolutely continuous distribution
## 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 ← distroptions("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
for(i in 1:n){ Vi ← phi^i*V; H ← H + Vi }
# may take some time

## 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)(1-1e-15)
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, type = "l")
title("Densities")
legend(low, d(X)(0), 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, type = "l")
title("Cumulative_distribution_functions")
legend(low, 1.0, legend=c("exact", "FFT"),
        fill=c("black", "orange"))

## quantile functions
x ← seq(from = eps, to = 1-eps, length = 200)
plot(x, q(X)(x),type = "l", lwd = 5)
lines(x , q(H)(x), col = "orange", lwd = 1, type = "l")
title("Quantile_functions")
legend(0, q(X)(1-eps), 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-5, N1=X, N2=H)
cat("Total_variation_distance_of_densities:\t")
print(dv) # 2.7e-05

## 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")
  # 1.3e-05
### old distroptions

```

```
distroptions("TruncQuantile", oldeps)
```

9.5 Truncation and Huberization/winsorization

Code available under

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

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

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

```
require(distr)

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

setMethod("Huberize",
  signature(object = "AbscontDistribution",
    lower = "numeric", upper = "numeric"),
  function(object, lower, upper){
    ## new random number function
    rnew = function(n){
      rn = r(object)(n)
      ifelse(rn < lower, lower,
        ifelse(rn >= upper, upper, rn))
    }

    ## new cdf
    pnew = function(x)
      ifelse(x < lower, 0,
        ifelse(x >= upper, 1, p(object)(x)))

    ## new quantile function
```

```

plower = p(object)(lower)
pupper = p(object)(upper)
qnew = function(x)
  ifelse(x < plower ,
         ifelse(x < 0, NA, -Inf),
         ifelse(x >= pupper ,
                ifelse(x > 1, NA, upper),
                q(object)(x)))

  list(r = rnew , p = pnew , q = qnew)
})

# Example
# Normal(0,1)-Distribution huberized at -0.5 and 1
N = Norm()
rpq = Huberize(N, -0.5, 1)

# some huberized randomnumbers
rpq$r(10)

# cdf of huberized Normal-Distribution
# and of Normal-Distribution
x = seq(-1.5, 1.5, length = 1000)
plot(x, rpq$p(x),
      type = "l",
      lwd = 5,
      ylab = "CDF")
lines(x, p(N)(x),
      lwd = 2,
      col = "red")
legend(-1.5,1,
        legend = c("N(0,1)", "N(0,1)_huberized"),
        fill = c("red", "black"))

# quantile functions
x = seq(0, 1, length = 1000)
plot(x, rpq$q(x),
      type = "l",
      lwd = 5,
      ylab = "Quantiles",
      ylim = c(-2.5,3))

```

```

lines(x, q(N)(x),
      lwd = 2,
      col = "red")
legend(0,3,
      legend = c("N(0,1)", "N(0,1)  $\perp$ huberized"),
      fill = c("red", "black"))

require(distr)

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

setMethod("Truncate",
  signature(object = "AbscontDistribution",
    lower = "numeric", upper = "numeric"),
  function(object, lower, upper){
    ## new random number function
    rnew = function(n){
      rn = r(object)(n)
      while(TRUE){
        rn[rn < lower] = NA
        rn[rn > upper] = NA
        index = is.na(rn)
        if(!any(index)) break
        rn[index] = r(object)(sum(index))
      }
      rn
    }

    ## new cdf
    plower = p(object)(lower)
    pupper = p(object)(upper)
    pnew = function(x)
      ifelse(x < lower, 0,
        ifelse(x >= upper, 1,
          (p(object)(x) - plower)/(pupper - plower)))

    ## new density

```

```

lostmass = plower + 1 - pupper

dnew = function(x)
  ifelse(x < lower, 0,
        ifelse(x >= upper, 0,
              d(object)(x)/(1-lostmass)))

# new quantile
qfun1 ← function(x){
  if(x == 0) return(lower)
  if(x == 1) return(upper)
  fun ← function(t) pnew(t) - x
  uniroot(fun, interval = c(lower, upper))$root
}
qfun2 ← function(x)
  sapply(x, qfun1)

return(new("AbscontDistribution", r = rnew,
          d = dnew, p = pnew, q = qfun2))
})

# Example
# Normal(0,1)-Distribution truncated at -0.5 and 1

N = Norm()
Z = Truncate(N, -0.5, 1)

# the truncated Distribution

plot(Z)

# some truncated randomnumbers

r(Z)(10)

# cdf of truncated Normal-Distribution
# and of Normal-Distribution

x = seq(-1.5, 1.5, length = 1000)
plot(x, p(Z)(x),
     type = "l",

```

```

      lwd = 5,
      xlab = "",
      ylab = "CDF")
lines(x, p(N)(x),
      lwd = 2,
      col = "red")
legend(-1.5,1,
      legend = c("N(0,1)", "N(0,1)_truncated"),
      fill = c("red", "black"))

# density of truncated Normal-Distribution
# and of Normal-Distribution

x = seq(-1.5, 1.5, length = 1000)
plot(x, d(Z)(x),
      type = "l",
      lwd = 5,
      xlab = "",
      ylab = "density")
lines(x, d(N)(x),
      lwd = 2,
      col = "red")
legend(-1.5,0.7, legend = c("N(0,1)", "N(0,1)_truncated"),
      fill = c("red", "black"))

```

9.6 Distribution of minimum and maximum of two independent random variables

Code available under

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

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

```

require(distr)

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

```

```

setMethod("Minimum",
  signature(e1 = "AbscontDistribution",
    e2 = "AbscontDistribution"),
  function(e1, e2){
    ## new random number function
    rnew ← function(n){
      rn1 ← r(e1)(n)
      rn2 ← r(e2)(n)

      ifelse(rn1 < rn2, rn1, rn2)
    }

    ## new cdf
    pnew ← function(x){
      p1 ← p(e1)(x)
      p2 ← p(e2)(x)
      p1 + p2 - p1 * p2
    }

    ## new density
    dnew ← function(x){
      d1 ← d(e1)(x)
      d2 ← d(e2)(x)
      p1 ← p(e1)(x)
      p2 ← p(e2)(x)
      d1 + d2 - d1 * p2 - p1 * d2
    }

    ## new quantile function
    lower1 ← q(e1)(0)
    lower2 ← q(e2)(0)
    upper1 ← q(e1)(1)
    upper2 ← q(e2)(1)
    lower ← min(lower1, lower2)
    upper ← min(upper1, upper2)

    maxquantile = min(q(e1)(1 - 1e-6),
      q(e2)(1 - 1e-6))
    minquantile = min(q(e1)(1e-6),
      q(e2)(1e-6))
  }

```

```

    qfun1 ← function(x){
      if(x == 0) return(lower)
      if(x == 1) return(upper)
      fun ← function(t) pnew(t) - x
      uniroot(fun ,
        interval = c(maxquantile ,
                      minquantile))$root
    }
    qfun2 ← function(x)
      sapply(x, qfun1)

    return(new(" AbscontDistribution" , r = rnew ,
              d = dnew, p = pnew, q = qfun2))
  })

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

setMethod("Maximum" ,
  signature(e1 = " AbscontDistribution" ,
            e2 = " AbscontDistribution" ),
  function(e1, e2){
    ## new random number function
    rnew ← function(n){
      rn1 ← r(e1)(n)
      rn2 ← r(e2)(n)

      ifelse(rn1 > rn2, rn1, rn2)
    }

    ## new cdf
    pnew ← function(x){
      p1 ← p(e1)(x)
      p2 ← p(e2)(x)
      p1 * p2
    }

    ## new density
    dnew ← function(x){
      d1 ← d(e1)(x)

```

```

    d2 ← d(e2)(x)
    p1 ← p(e1)(x)
    p2 ← p(e2)(x)
    d1 * p2 + p1 * d2
  }

  ## new quantile function
  lower1 ← q(e1)(0)
  lower2 ← q(e2)(0)
  upper1 ← q(e1)(1)
  upper2 ← q(e2)(1)
  lower ← max(lower1, lower2)
  upper ← max(upper1, upper2)

  maxquantile = max(q(e1)(1 - 1e-6), q(e2)(1 - 1e-6))
  minquantile = max(q(e1)(1e-6), q(e2)(1e-6))

  qfun1 ← function(x){
    if(x == 0) return(lower)
    if(x == 1) return(upper)
    fun ← function(t) pnew(t) - x
    uniroot(fun, interval = c(maxquantile,
                              minquantile))$root
  }
  qfun2 ← function(x)
    sapply(x, qfun1)

  return(new("AbscontDistribution", r = rnew,
            d = dnew, p = pnew, q = qfun2))
})

```

Example

```

N ← Norm(mean = 0, sd = 1)
U ← Unif(Min = 0, Max = 1)

Y ← Maximum(N, U)
plot(Y)

Z ← Minimum(N,U)

```

`plot(Z)`

9.7 Instructive destructive example

Code available under

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

9.8 A simulation example

needs packages "distrSim"/"distrTEst"

Code available under

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

```
require(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)

psim ← function(theta, y, m0){
  mean(pmin(pmax(-m0, y - theta), m0))
}
mestimator ← function(x, m = 0.7) {
  uniroot(psim,
          low = -20,
          up = 20,
          tol = 1e-10,
          y = x,
          m0 = m,
          maxiter = 20)$root
}

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

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

opar ← par()

par(mfcol=c(1,3))

```

```

yrange.id = range(result(result.id.median))
boxplot(result(result.id.mean), ylim = yrange.id)
title("Results_for_Mean_under_ideal_conditions")
boxplot(result(result.id.mest), ylim = yrange.id)
title("Results_for_Huber-Estimator_under_ideal_conditions")
boxplot(result(result.id.median), ylim = yrange.id)
title("Results_for_Median_under_ideal_conditions")

yrange.cont = range(result(result.cont.mean))
boxplot(result(result.cont.mean), ylim = yrange.cont)
title("Results_for_Mean_under_worse_conditions")
boxplot(result(result.cont.mest), ylim = yrange.cont)
title("Results_for_Huber-Estimator_under_worse_conditions")
boxplot(result(result.cont.median), ylim = yrange.cont)
title("Results_for_Median_under_worse_conditions")

par(opar)

```

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...

9.9 Expectation of a given function under a given distribution

Code available under

```

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

```

As in examples 9.5 and 9.6, we illustrate the use of package "distr" by implementing a general evaluation of expectation and variance under a given distribution.

```

require(distr)

if(!isGeneric("E"))
  setGeneric("E",
    function(object, fun) standardGeneric("E"))

setMethod("E",
  signature(object = "AbscontDistribution",

```

```

        fun = "function"),
function(object, fun){
  integrand ← function(x) fun(x) * d(object)(x)
  return(integrate(integrand,
                    lower = q(object)(0),
                    upper = q(object)(1))$value)
})

setMethod("E",
  signature(object = "DiscreteDistribution",
            fun = "function"),
function(object, fun){
  supp = support(object)
  return(sum(fun(supp) * d(object)(supp)))
})

```

Example

```

id ← function(x) x
sq ← function(x) x^2

```

Expectation and Variance of Binom(6, 0.5)

```

B ← Binom(6, 0.5)
E(B, id)
E(B, sq) - E(B, id)^2

```

Expectation and Variance of Norm(1, 1)

```

N ← Norm(1, 1)
E(N, id)
E(N, sq) - E(N, id)^2

```

9.10 n -fold convolution of absolutely continuous distributions

Code 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.

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

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

#####
## Function for n-fold convolution
## -- absolute continuous distribution --
#####
setMethod("convpow",
  signature(D1 = "AbscontDistribution",
    N = "numeric"),
  function(D1, N){
    if((N < 1)||(!identical(floor(N), N)))
      stop("N has to be a natural greater than 0")

    m ← DefaultNrFFTGridPointsExponent

    lower ← ifelse((q(D1)(0) > - Inf),
      q(D1)(0), q(D1)(TruncQuantile))
    upper ← ifelse((q(D1)(1) < Inf),
      q(D1)(1), q(D1)(1 - TruncQuantile))

    M ← 2^m
    h ← (upper-lower)/M
    if(h > 0.01)
      warning(paste("Grid for approxfun too wide,",
        "increase DefaultNrFFTGridPointsExponent"))
    x ← seq(from = lower, to = upper, by = h)
    p1 ← p(D1)(x)
    p1 ← p1[2:(M + 1)] - p1[1:M]

    ## computation of DFT
    pn ← c(p1, numeric((N-1)*M))
  })

```

```

fftpn ← fft(pn)

## convolution theorem for DFTs
pn ← Re(fft(fftpnN, 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)

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

## cdf with continuity correction h/2
pnfun1 ← approxfun(x = x+0.5*h, y = pn, yleft = 0,
                   yright = pn[i.max+1])
pnfun2 ← function(x) pnfun1(x) / pn[i.max+1]

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

```

```

    y ← replace(y, ind2[ind2 != 0], yright)
    return(y)
  }
  options(w0)

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

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

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

## convolution power
N ← 10

## convolution via FFT
AN ← convpow(A, N)

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

## plots of the results
eps ← distroptions("TruncQuantile")
par(mfrow=c(1,3))
low ← q(AN1)(eps)
upp ← q(AN1)(1-eps)
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(low, d(AN)(0), 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("Cumulative_distribution_functions")
legend(low, 1.0, 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(0, q(AN1)(1-eps), legend=c("exact", "FFT"),
        fill=c("black", "orange"))

```

10 Acknowledgement

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`.

References

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

Chambers J.M. (1998): *Programming with data. A guide to the S language*. Springer. <http://cm.bell-labs.com/stat/Sbook/index.html>. 1

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>. 6,
7

Kohl, M., Ruckdeschel, P., and 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>.
3, 6

Rice J.A. (1988): *Mathematical statistics and data analysis..* Wadsworth & Brooks/Cole Advanced Books & Software., Pacific Grove, California. 9.1