S4 classes for distributions and extensions

Peter Ruckdeschel[1] Matthias Kohl[2]

Fakultät für Mathematik und Physik

Peter.Ruckdeschel@uni-bayreuth.de

www.uni-bayreuth.de/departments/math/org/mathe7/RUCKDESCHEL

[2]StaMatS - Statistik + Mathematik Service

Matthias.Kohl@stamats.de
www.stamats.de

Frankfurter Stochastiktage
16.03.2006

Motivation I

- Situation: algorithm / program shall cope with any distribution
- How to pass a distribution as an argument?
- Construction up to now:
  - a lot of distributions implemented to R
    — Gaussian, Poisson, Exponential, Gamma, etc.
  - for each:
    - cdf [≙p]
    - density / probability function [≙d]
    - quantile function [≙q]
    - function to simulate r.v.'s [≙r]
  - Naming convention: `<prefix><Name>`

Motivation I

- Situation: algorithm / program shall cope with any distribution
- How to pass a distribution as an argument?
- Construction up to now:
  - a lot of distributions implemented to R
    — Gaussian, Poisson, Exponential, Gamma, etc.
  - for each:
    - cdf [≙p]
    - density / probability function [≙d]
    - quantile function [≙q]
    - function to simulate r.v.'s [≙r]
  - Naming convention: `<prefix><Name>`

Motivation I

- Situation: algorithm / program shall cope with any distribution
- How to pass a distribution as an argument?
- Construction up to now:
  - a lot of distributions implemented to R
    — Gaussian, Poisson, Exponential, Gamma, etc.
  - for each:
    - cdf [≙`p`]
    - density / probability function [≙`d`]
    - quantile function [≙`q`]
    - function to simulate r.v.'s [≙`r`]
  - Naming convention: `<prefix><Name>`

Motivation I

- Situation: algorithm / program shall cope with any distribution
- How to pass a distribution as an argument?
- Construction up to now:
  - a lot of distributions implemented to R
    — Gaussian, Poisson, Exponential, Gamma, etc.
  - for each:
    - cdf [$\hat{=}$`p`]
    - density / probability function [$\hat{=}$`d`]
    - quantile function [$\hat{=}$`q`]
    - function to simulate r.v.'s [$\hat{=}$`r`]
  - Naming convention: `<prefix><Name>`

Motivation I

- Situation: algorithm / program shall cope with any distribution
- How to pass a distribution as an argument?
- Construction up to now:
  - a lot of distributions implemented to R
    — Gaussian, Poisson, Exponential, Gamma, etc.
  - for each:
    - cdf [$\hat{=}$`p`]
    - density / probability function [$\hat{=}$`d`]
    - quantile function [$\hat{=}$`q`]
    - function to simulate r.v.'s [$\hat{=}$`r`]
  - Naming convention: `<prefix><Name>`

Motivation II

- e.g. to get the median of a general distribution:

```
mymedian ← function(vtlg, ...)
{ eval(parse(text =
      paste( "x = q", vtlg,
             "(1/2,...)", sep = "")))
  return(x)}
```

- better idea: having a "variable type" *distribution* and functions `p`, `d`, `q`, `r` defined for this type
- then: **q**(x) returns the quantile function $\rightsquigarrow$
  median ←**function**(X){**q**(X)(0.5)}

$\Longrightarrow$ Development of this concept in package `distr`

- packages `distr`, `distrEx`: published on `CRAN`; current version `1.6`; extensive documentation available (see references)

Motivation II

- e.g. to get the median of a general distribution:

```
mymedian ← function(vtlg, ...)
{ eval(parse(text =
      paste( "x = q", vtlg,
             "(1/2,...)", sep = "")))
  return(x)}
```

- better idea: having a "variable type" *distribution* and functions `p`, `d`, `q`, `r` defined for this type
- then: **q**(x) returns the quantile function $\rightsquigarrow$
  median ←**function**(X){**q**(X)(0.5)}

$\Longrightarrow$ Development of this concept in package `distr`

- packages `distr`, `distrEx`: published on `CRAN`; current version `1.6`; extensive documentation available (see references)

Motivation II

- e.g. to get the median of a general distribution:

```
mymedian ← function(vtlg, ...)
{ eval(parse(text =
      paste( "x = q", vtlg,
             "(1/2,...)", sep = "")))
  return(x)}
```

- better idea: having a "variable type" *distribution* and functions `p`, `d`, `q`, `r` defined for this type
- then: **q**(x) returns the quantile function $\rightsquigarrow$
  median ←**function**(X){**q**(X)(0.5)}

$\Longrightarrow$ Development of this concept in package `distr`

- packages `distr`, `distrEx`: published on `CRAN`; current version `1.6`; extensive documentation available (see references)

Object Orientation in S/R

- particular version of object orientation:
  *F*unction-orientated- *FOOP* as opposed to *COOP*
  - methods are *not* part of object but are managed by *generic functions*
  - depending on the arguments different methods are dispatched
  - example: **plot**
- from version `1.7.0` on, R realizes the S4-class concept, c.f. Chambers[98]
- `distr`: distributions are classes with slots/members `p`, `d`, `q`, `r`
- arithmetics for distribution objects realized by *FOOP*
- CAVEAT: arithmetics operates on underlying random variables, *not* on distributions

Object Orientation in S/R

- particular version of object orientation:
  *F*unction-orientated- *FOOP* as opposed to *COOP*
  - methods are *not* part of object but are managed by *generic functions*
  - depending on the arguments different methods are dispatched
  - example: **plot**
- from version `1.7.0` on, R realizes the S4-class concept, c.f. Chambers[98]

- `distr`: distributions are classes with slots/members `p`, `d`, `q`, `r`
- arithmetics for distribution objects realized by *FOOP*
- CAVEAT: arithmetics operates on underlying random variables, *not* on distributions

Object Orientation in S/R

- particular version of object orientation:
  *F*unction-orientated- *FOOP* as opposed to *COOP*
  - methods are *not* part of object but are managed by *generic functions*
  - depending on the arguments different methods are dispatched
  - example: **plot**
- from version `1.7.0` on, R realizes the S4-class concept, c.f. Chambers[98]

- `distr`: distributions are classes with slots/members `p`, `d`, `q`, `r`
- arithmetics for distribution objects realized by *FOOP*
- CAVEAT: arithmetics operates on underlying random variables, *not* on distributions

Object Orientation in S/R

- particular version of object orientation:
  *F*unction-orientated- *FOOP* as opposed to *COOP*
  - methods are *not* part of object but are managed by *generic functions*
  - depending on the arguments different methods are dispatched
  - example: **plot**
- from version `1.7.0` on, R realizes the S4-class concept, c.f. Chambers[98]

- `distr`: distributions are classes with slots/members `p`, `d`, `q`, `r`
- arithmetics for distribution objects realized by *FOOP*
- CAVEAT: arithmetics operates on underlying random variables, *not* on distributions

Example: arithmetics for distribution objects

```
> require("distr")
Loading required package: distr
[1] TRUE
> N <- Norm(mean = 2, sd = 1.3)
> P <- Pois(lambda = 1.2)
> Z <- 2 * N + 3 + P # exact transformation
Distribution Object of Class: AbscontDistribution
> plot(Z)
> p(Z)(0.4)
[1] 0.002415384
> q(Z)(0.3)
[1] 6.70507
> r(Z)(10)
 [1] 11.072931  7.519611 10.567212  3.358912
 [5]  7.955618  9.094524  5.293376  5.536541
 [9]  9.358270 10.689527
> Znew <- sin(abs(Z)) # by simulations
> plot(Znew)
> p(Znew)(0.2)
```

Example: arithmetics for distribution objects

```
> require("distr")
Loading required package: distr
[1] TRUE
> N <- Norm(mean = 2, sd = 1.3)
> P <- Pois(lambda = 1.2)
> Z <- 2 * N + 3 + P # exact transformation
Distribution Object of Class: AbscontDistribution
> plot(Z)
> p(Z)(0.4)
[1] 0.002415384
> q(Z)(0.3)
[1] 6.70507
> r(Z)(10)
 [1] 11.072931  7.519611 10.567212  3.358912
 [5]  7.955618  9.094524  5.293376  5.536541
 [9]  9.358270 10.689527
> Znew <- sin(abs(Z)) # by simulations
> plot(Znew)
> p(Znew)(0.2)
```

Example: arithmetics for distribution objects

```
> require("distr")
Loading required package: distr
[1] TRUE
> N <- Norm(mean = 2, sd = 1.3)
> P <- Pois(lambda = 1.2)
> Z <- 2 * N + 3 + P # exact transformation
Distribution Object of Class: AbscontDistribution
> plot(Z)
> p(Z)(0.4)
[1] 0.002415384
> q(Z)(0.3)
[1] 6.70507
> r(Z)(10)
 [1] 11.072931  7.519611 10.567212  3.358912
 [5]  7.955618  9.094524  5.293376  5.536541
 [9]  9.358270 10.689527
> Znew <- sin(abs(Z)) # by simulations
> plot(Znew)
> p(Znew)(0.2)
```

Example: arithmetics for distribution objects

```
> require("distr")
Loading required package: distr
[1] TRUE
> N <- Norm(mean = 2, sd = 1.3)
> P <- Pois(lambda = 1.2)
> Z <- 2 * N + 3 + P # exact transformation
Distribution Object of Class: AbscontDistribution
> plot(Z)
> p(Z)(0.4)
[1] 0.002415384
> q(Z)(0.3)
[1] 6.70507
> r(Z)(10)
 [1] 11.072931  7.519611 10.567212  3.358912
 [5]  7.955618  9.094524  5.293376  5.536541
 [9]  9.358270 10.689527
> Znew <- sin(abs(Z)) # by simulations
> plot(Znew)
> p(Znew)(0.2)
```

Contents of distrEx

Package distrEx extends distr and includes

- a general expectation operator to a given distribution F
- several functionals on distributions like the median, MAD and IQR
- several distances between distributions (e.g. Kolmogoroff–distance)
- (factorized) conditional distributions
- (factorized) conditional expectations

Contents of distrEx

Package distrEx extends distr and includes

- a general expectation operator to a given distribution F
- several functionals on distributions like the median, MAD and IQR
- several distances between distributions (e.g. Kolmogoroff–distance)
- (factorized) conditional distributions
- (factorized) conditional expectations

Example: expectation operator

- for a normal variable D_1 try to realize $\mathrm{E}\, D_1$, $\mathrm{E}\, D_1^2$, and for some $m_1 \in \mathbb{R}$, $\mathrm{E}(D_1 - m_1)^2$

```
require("distrEx")
D1 ← Norm(mean=2)
m1 ← E(D1)   # = 2
E(D1, function(x){ x^2 }) # E(D1^2)
```

- now —without changing the code— the same for a Poisson variable; this gives the same calls but different dispatched methods

```
D1 ← Pois(lambda=3)
m1 ← E(D1)  # = 3
E(D1, function(x){ x^2 })
```

Example: expectation operator

- for a normal variable D_1 try to realize $\mathrm{E}\, D_1$, $\mathrm{E}\, D_1^2$, and for some $m_1 \in \mathbb{R}$, $\mathrm{E}(D_1 - m_1)^2$

```
require("distrEx")
D1 ← Norm(mean=2)
m1 ← E(D1)   # = 2
E(D1, function(x){ x^2 }) # E(D1^2)
```

- now —without changing the code— the same for a Poisson variable; this gives the same calls but different dispatched methods

```
D1 ← Pois(lambda=3)
m1 ← E(D1) # = 3
E(D1, function(x){ x^2 })
```

Example: expectation operator

- for a normal variable D_1 try to realize $\mathrm{E}\, D_1$, $\mathrm{E}\, D_1^2$, and for some $m_1 \in \mathbb{R}$, $\mathrm{E}(D_1 - m_1)^2$

```
require("distrEx")
D1 ← Norm(mean=2)
m1 ← E(D1)   # = 2
E(D1, function(x){ x^2 }) # E(D1^2)
```

- now —without changing the code— the same for a Poisson variable; this gives the same calls but different dispatched methods

```
D1 ← Pois(lambda=3)
m1 ← E(D1)  # = 3
E(D1, function(x){ x^2 })
```

Illustration 1: CLT —under arbitrary distribution

- we want to illustrate the Lindeberg-Lévy theorem
- input should be any univariate distribution Distr
- notation: $X_i \overset{\text{i.i.d.}}{\sim} F$, $S_n = \sum_{i=1}^{n} X_i$, $T_n = (S_n - \mathrm{E}\, S_n)/\sqrt{\mathrm{Var}\, S_n}$
- output: sequence of length len of plots of $\mathcal{L}(T_n)$
- realized in illustrateCLT(Distr, len)
- essential code
  - a function for standardizing and centering

```
make01 <- function(x)(x-E(x))/sd(x)
```

  - update in a loop starting with Sn $\leftarrow$ 0

```
Sn <- Sn + Distr
Tn <- make01(Sn)
## here: Distr is absolutely continuous
dTn <- d(Tn)(x)
```

Illustration 1: CLT —under arbitrary distribution

- we want to illustrate the Lindeberg-Lévy theorem
- input should be any univariate distribution Distr
- notation: $X_i \overset{\text{i.i.d.}}{\sim} F$, $S_n = \sum_{i=1}^{n} X_i$, $T_n = (S_n - \mathrm{E}\, S_n)/\sqrt{\mathrm{Var}\, S_n}$
- output: sequence of length len of plots of $\mathcal{L}(T_n)$
- realized in illustrateCLT(Distr, len)
- essential code
  - a function for standardizing and centering

```
make01 ← function(x)(x-E(x))/sd(x)
```

  - update in a loop starting with Sn ← 0

```
Sn ← Sn + Distr
Tn ← make01(Sn)
## here: Distr is absolutely continuous
dTn ← d(Tn)(x)
```

Illustration 1: CLT —under arbitrary distribution

- we want to illustrate the Lindeberg-Lévy theorem
- input should be any univariate distribution Distr
- notation: $X_i \overset{\text{i.i.d.}}{\sim} F$, $S_n = \sum_{i=1}^{n} X_i$, $T_n = (S_n - \mathrm{E}\, S_n)/\sqrt{\mathrm{Var}\, S_n}$
- output: sequence of length len of plots of $\mathcal{L}(T_n)$
- realized in illustrateCLT(Distr, len)
- essential code
  - a function for standardizing and centering

```
make01 <- function(x)(x-E(x))/sd(x)
```

  - update in a loop starting with Sn <- 0

```
Sn <- Sn + Distr
Tn <- make01(Sn)
## here: Distr is absolutely continuous
dTn <- d(Tn)(x)
```

Illustration 2: Minimum-distance- and ML-functionals

- we want to estimate the parameter θ in a parametric family
- methods: minimum-distance and ML
- in both cases in an optimization a member in the class is distinguished as “closest” to the data
- input: data and parametric model
- output: estimate
- implementation: parametric model as class (compare M. Kohl’s talk), which has slots
  - name, distribution ,
  - additionally: a slot modifparameter, a function realizing theta$\mapsto P_{\text{theta}}$
- generic functions MDE(**model**,**data**,**distance**), MLE(**model**,**data**)

Illustration 2: Minimum-distance- and ML-functionals

- we want to estimate the parameter θ in a parametric family
- methods: minimum-distance and ML
- in both cases in an optimization a member in the class is distinguished as “closest” to the data
- input: data and parametric model
- output: estimate
- implementation: parametric model as class (compare M. Kohl’s talk), which has slots
  - name, distribution ,
  - additionally: a slot modifparameter, a function realizing theta$\mapsto P_{\text{theta}}$
- generic functions MDE(**model**,**data**,**distance**), MLE(**model**,**data**)

Illustration 2: Minimum-distance- and ML-functionals

- we want to estimate the parameter θ in a parametric family
- methods: minimum-distance and ML
- in both cases in an optimization a member in the class is distinguished as “closest” to the data
- input: data and parametric model
- output: estimate
- implementation: parametric model as class (compare M. Kohl’s talk), which has slots
  - name, distribution ,
  - additionally: a slot modifparameter, a function realizing theta$\mapsto P_{\text{theta}}$
- generic functions MDE(**model**,**data**,**distance**), MLE(**model**,**data**)

Illustration 2: Minimum-distance- and ML-functionals

- we want to estimate the parameter θ in a parametric family
- methods: minimum-distance and ML
- in both cases in an optimization a member in the class is distinguished as “closest” to the data
- input: data and parametric model
- output: estimate
- implementation: parametric model as class (compare M. Kohl’s talk), which has slots
  - name, distribution ,
  - \+ additionally: a slot modifparameter, a function realizing theta$\mapsto P_{\text{theta}}$
- generic functions MDE(**model**,**data**,**distance**), MLE(**model**,**data**)

Illustration 2: Minimum-distance- and ML-functionals II

essential code

- to fit a distribution distr to **data** according to criterium(distr ,**data**) we use

```
fitParam ← function(model, data0, criterium ....)
  { #define a function in theta to be optimized:
    ftoOptimize ← function(theta)
       {Ptheta  ← modifparameter(model)(theta)
        criterium(Ptheta,data0)          }
    #in one dimension use  "optimize", in higher "optim"
      {if(dimension(param(model)) == 1)
          theta ← optimize(f = ftoOptimize,
                           interval = searchinterval0, ...)$minimum
       else
          theta ← optim(par = initval0,
                              f = ftoOptimize, ...)$par   }
  return(theta)}
```

- criterium can be the negative log-likelihood or a distance of the theoretical distribution to the empirical distribution like Kolmogoroff-distance

Illustration 2: Minimum-distance- and ML-functionals II

essential code

- to fit a distribution distr to **data** according to criterium(distr ,**data**) we use

```
fitParam ← function(model, data0, criterium ....)
  { #define a function in theta to be optimized:
    ftoOptimize ← function(theta)
       {Ptheta  ← modifparameter(model)(theta)
        criterium(Ptheta,data0)          }
    #in one dimension use  "optimize", in higher "optim"
      {if(dimension(param(model)) == 1)
          theta ← optimize(f = ftoOptimize,
                           interval = searchinterval0, ...)$minimum
       else
          theta ← optim(par = initval0,
                              f = ftoOptimize, ...)$par   }
  return(theta)}
```

- criterium can be the negative log-likelihood or a distance of the theoretical distribution to the empirical distribution like Kolmogoroff-distance

Illustration 3: Deconvolution I

- Situation: $X \sim K$, $\varepsilon \sim F$, stoch. independent; $Y = X + \varepsilon$
- goal: reconstruction X by means of Y
- methods: $\mathrm{E}[X|Y]$, $\mathrm{postmode}(X|Y)$
- input: any univariate distributions $K =$ Regr, $F =$ Error
- output: mappings $y \mapsto \mathrm{E}[X|Y = y]$, $\mathrm{postmode}(X|Y = y)$
- realized by means of ProgncondDistribution(Regr, Error)

$\rightsquigarrow$ generates $\mathcal{L}(X|Y = y)$ where y is coded as parameter cond

Illustration 3: Deconvolution I

- Situation: $X \sim K$, $\varepsilon \sim F$, stoch. independent; $Y = X + \varepsilon$
- goal: reconstruction X by means of Y
- methods: $\mathrm{E}[X|Y]$, $\mathrm{postmode}(X|Y)$

- input: any univariate distributions $K =$ Regr, $F =$ Error
- output: mappings $y \mapsto \mathrm{E}[X|Y = y]$, $\mathrm{postmode}(X|Y = y)$

- realized by means of ProgncondDistribution(Regr, Error)

$\rightsquigarrow$ generates $\mathcal{L}(X|Y = y)$ where y is coded as parameter cond

Illustration 3: Deconvolution I

- Situation: $X \sim K$, $\varepsilon \sim F$, stoch. independent; $Y = X + \varepsilon$
- goal: reconstruction X by means of Y
- methods: $\mathrm{E}[X|Y]$, $\mathrm{postmode}(X|Y)$

- input: any univariate distributions $K =$ Regr, $F =$ Error
- output: mappings $y \mapsto \mathrm{E}[X|Y = y]$, $\mathrm{postmode}(X|Y = y)$

- realized by means of ProgCondDistribution(Regr, Error)
- ⇝ generates $\mathcal{L}(X|Y = y)$ where y is coded as parameter cond

Illustration 3: Deconvolution I

- Situation: $X \sim K$, $\varepsilon \sim F$, stoch. independent; $Y = X + \varepsilon$
- goal: reconstruction X by means of Y
- methods: $\mathrm{E}[X|Y]$, $\mathrm{postmode}(X|Y)$

- input: any univariate distributions $K =$ Regr, $F =$ Error
- output: mappings $y \mapsto \mathrm{E}[X|Y = y]$, $\mathrm{postmode}(X|Y = y)$

- realized by means of PrognCondDistribution(Regr, Error)

$\rightsquigarrow$ generates $\mathcal{L}(X|Y = y)$ where y is coded as parameter cond

Illustration 3: Deconvolution II

essential code

- filling of the slots `r`, `d`, `p`, `q` for some machine-eps

```
rf←function(n,cond) cond − r(Error)(n)
df←function(x,cond) d(Regr)(x)∗d(Error)(cond−x)
qf←function(x,cond) cond−q(Error)(1−x)
pf←function(x,cond) integrate(df, low=q(Error)(eps), up=x,
                              cond=cond)$value
```

- conditional expectation $\mathrm{E}[X|Y = y]$

```
PXy ← PrognCondDistribution(Regr, Error)
E(PXy, cond=y)
```

- posterior mode $\mathrm{postmode}(X|Y = y)$

```
post.mod ← function(cond, e1) {
    optimize(f = d(PXy), c(q(PXy)(eps, cond),
        q(PXy)(1−eps, cond)), cond = cond)$maximum}
```

Illustration 3: Deconvolution II

essential code

- filling of the slots `r`, `d`, `p`, `q` for some machine-eps

```
rf←function(n,cond) cond − r(Error)(n)
df←function(x,cond) d(Regr)(x)∗d(Error)(cond−x)
qf←function(x,cond) cond−q(Error)(1−x)
pf←function(x,cond) integrate(df, low=q(Error)(eps), up=x,
                              cond=cond)$value
```

- conditional expectation $\mathrm{E}[X|Y=y]$

```
PXy ← ProgncondDistribution(Regr, Error)
E(PXy, cond=y)
```

- posterior mode $\mathrm{postmode}(X|Y=y)$

```
post.mod ← function(cond, e1) {
    optimize(f = d(PXy), c(q(PXy)(eps, cond),
        q(PXy)(1−eps, cond)), cond = cond)$maximum}
```

Illustration 3: Deconvolution II

essential code

- filling of the slots `r`, `d`, `p`, `q` for some machine-eps

```
rf←function(n,cond) cond − r(Error)(n)
df←function(x,cond) d(Regr)(x)∗d(Error)(cond−x)
qf←function(x,cond) cond−q(Error)(1−x)
pf←function(x,cond) integrate(df, low=q(Error)(eps), up=x,
                              cond=cond)$value
```

- conditional expectation $\mathrm{E}[X|Y=y]$

```
PXy ← PrognCondDistribution(Regr, Error)
E(PXy, cond=y)
```

- posterior mode $\mathrm{postmode}(X|Y=y)$

```
post.mod ← function(cond, e1) {
    optimize(f = d(PXy), c(q(PXy)(eps, cond),
        q(PXy)(1−eps, cond)), cond = cond)$maximum}
```

Illustration 3: Deconvolution II

essential code

- filling of the slots `r`, `d`, `p`, `q` for some machine-eps

```
rf←function(n,cond) cond − r(Error)(n)
df←function(x,cond) d(Regr)(x)∗d(Error)(cond−x)
qf←function(x,cond) cond−q(Error)(1−x)
pf←function(x,cond) integrate(df, low=q(Error)(eps), up=x,
                              cond=cond)$value
```

- conditional expectation $\mathrm{E}[X|Y = y]$

```
PXy ← PrognCondDistribution(Regr, Error)
E(PXy, cond=y)
```

- posterior mode $\mathrm{postmode}(X|Y = y)$

```
post.mod ← function(cond, e1) {
    optimize(f = d(PXy), c(q(PXy)(eps, cond),
        q(PXy)(1−eps, cond)), cond = cond)$maximum}
```

Bibliography

J. M. Chambers.
Programming with Data. A guide to the S language.
Springer, 1998.
URL http://cm.bell-labs.com/cm/ms/departments/sia/Sbook/.

R Development Core Team.
R*: A language and environment for statistical computing*.
R Foundation for Statistical Computing, Vienna, Austria, 2005
URL http://www.R-project.org.

M. Kohl, P. Ruckdeschel, and T. Stabla.
General Purpose Convolution Algorithm for Distributions in S4-Classes by means of FFT. Technical Report. Feb. 2005.
URL http://www.uni-bayreuth.de/departments/math/org/mathe7/RUCKDESCHEL/pubs/comp.pdf

P. Ruckdeschel, M. Kohl, T. Stabla, and F. Camphausen.
S4 Classes for Distributions.
Appears at R-News. Also available as manual for packages distr, distrSim, distrTEst version 1.6, Oct. 2005.
URL http://www.uni-bayreuth.de/departments/math/org/mathe7/DISTR.