R Package distrMod: S4 Classes and Methods for Probability Models

Matthias Kohl
FH Furthwangen

Peter Ruckdeschel
Fraunhofer ITWM Kaiserslautern

Abstract

This vignette is published as Kohl and Ruckdeschel (2010c). Package **distrMod** provides an object oriented (more specifically S4-style) implementation of probability models. Moreover, it contains functions and methods to compute minimum criterion estimators – in particular, maximum likelihood and minimum distance estimators.

Keywords: probability models, minimum criterion estimators, minimum distance estimators, maximum likelihood estimators, S4 classes, S4 methods.

1. Introduction

1.1. Aims of package distrMod

What is distrMod? It is an extension package for the statistical software R, (R Development Core Team 2010a) and is the latest member of a family of packages, which we call `distr`-family. The family so far consists of packages **distr**, **distrEx**, **distrEllipse**, **distrSim**, **distrTEst**, **distrTeach**, and **distrDoc**; see Ruckdeschel, Kohl, Stabla, and Camphausen (2006) and Ruckdeschel, Kohl, Stabla, and Camphausen (2010).
Package **distrMod** makes extensive use of the distribution classes of package **distr** as well as the functions and methods of package **distrEx**. Its purpose is to extend support in base R for distributions and in particular for parametric modelling by "object oriented" implementation of probability models via several new S4 classes and methods; see Section 2 and Chambers (1998) for more details. In addition, it includes functions and methods to compute minimum criterion estimators – in particular, maximum likelihood (ML[E]) (i.e. minimum negative log-likelihood) and minimum distance estimators (MDE).
Admittedly, **distrMod** is not the first package to provide infrastructure for ML estimation, we compete in some sense with such prominent functions as `fitdistr` from package **MASS** (Venables and Ripley 2002) and, already using the S4 paradigm, `mle` from package **stats4** (R Development Core Team 2010a).
Our implementation however, goes beyond the scope of these packages, as we work with distribution objects and have quite general methods available to operate on these objects.

Who should use it? It is aimed at users who want to use non-standard parametric models, allowing them to either explore these models, or fit them to data by non-standard techniques. The user will receive standardized output on which she/he may apply standard R functions

like `plot`, `show`, `confint`, `profile`.
By non-standard parametric models we mean models not in the list of explicit models covered by `fitdistr`; that is, `"Poisson"`, `"beta"`, `"cauchy"`, `"chi-squared"`, `"exponential"`, `"gamma"`, `"geometric"`, `"lognormal"`, `"logistic"`, `"negative binomial"`, `"normal"`, `"f"`, `"t"`, `"weibull"`. Standard as well as non-standard models can easily be implemented based on the infrastructure provided by packages **distr** and **distrEx**. We will demonstrate this using examples $(M2)$ and $(M4)$ specified in Section 1.2.
Non-standard techniques may include minimum criterion estimation, minimum distance estimation, a particular optimization routine not covered by `optim`/`optimize` in the MLE case, or some explicit expression for the MLE not covered by the standard class-room examples. Non-standard techniques may also stand for estimation of a (differentiable) function of the parameter as illustrated in example $(M3)$.
Despite this flexibility, we need not modify our code to cover all this. In short, we are able to implement **one** static algorithm which by S4 method dispatch dynamically takes care of various models and optimization techniques, thus avoiding redundancy and simplifying maintenance. We will explain this more precisely in Section 4.1.
All information relevant for a specific parametric model is grouped within an object of class `ParamFamily` or subclasses for which it may for instance be of interest to explore the (perhaps automatically derived, as in the case of example $(M2)$) score function and the corresponding Fisher information. The return value of the model fit, an estimate of the parameter, is an object of class `Estimator` or subclasses for which one may want to have confidence intervals, some profiling, etc. For objects of these classes we provide various methods for standard R functions; see Sections 3 and 4 for more details.

Availability The current version of package **distrMod** is 2.2 and can be found on the Comprehensive R Archive Network at `http://CRAN.R-project.org/package=distrMod`. The development version of the distr-family is located at R-Forge; see Theußl and Zeileis (2009).

1.2. Running examples

For illustrating the functionality of **distrMod**, we will use four running examples for each of which we assume i.i.d. observations X_i $(i = 1, \ldots, n,\ n \in \mathbb{N})$ distributed according to the respective P_θ:

$(M1)$ the one-dimensional normal location family $\mathcal{P} := \{P_\theta \,|\, \theta \in \mathbb{R}\}$ with $P_\theta = \mathcal{N}(\theta, 1)$. This model is L_2-differentiable (i.e. smoothly parametrized) with scores $\Lambda_\theta(x) = x - \theta$.

$(M2)$ a one-dimensional location and scale family $\mathcal{P} := \{P_\theta \,|\, \theta = (\mu, \sigma)' \in \mathbb{R} \times (0, \infty)\}$ with some non-standard P_θ. More precisely we assume,

$$X_i = \mu + \sigma V_i \qquad \text{for} \quad V_i \overset{\text{i.i.d}}{\sim} P \tag{1}$$

where $P = P_{\theta_0}$ $(\theta_0 = (0, 1)')$ is the following central distribution

$$P(dx) = p(x)\, dx, \qquad p(x) \propto e^{-|x|^3} \tag{2}$$

$\mathcal{P}$ is L_2-differentiable with scores $\Lambda_\theta(x) = (3\,\mathrm{sign}(y)y^2, 3|y|^3 - 1)/\sigma$ for $y = (x - \mu)/\sigma$.

$(M3)$ the gamma family $\mathcal{P} := \{P_\theta = \mathrm{gamma}(\theta) \,|\, \theta = (\beta, \xi)' \in (0, \infty)^2\}$ for scale parameter β and shape parameter ξ. This model is L_2-differentiable with scores $\Lambda_\theta(x) = (\frac{y-\xi}{\beta}, \log(y) - (\log\Gamma)'(\xi))$ for $y = x/\beta$ and

($M4$) a censored Poisson family: $\mathcal{P} := \{P_\theta \,|\, \theta \in (0,\infty)\}$ where $P_\theta = \mathcal{L}_\theta(X|X > 1)$ for $X \sim \text{Pois}(\theta)$, that is, we only observe counts larger or equal to 2 in a Poisson model. This model is L_2-differentiable with scores $\Lambda_\theta(x) = x/\theta - (1-e^{-\theta})/(1-(1+\theta)e^{-\theta})$.

We will estimate θ from $X_1, \ldots, X_n$ with mean squared error (MSE) as risk. This makes the MLE asymptotically optimal. Other considerations, in particular robustness issues, suggest that one should also look at alternatives. For the sake of this paper, we will limit ourselves to one alternative in each model. In model ($M1$) we will use the median as most-robust estimator, in model ($M2$) we will look at the very robust estimator $\theta_r = (\text{median}, \text{mad})$ (mad = suitable standardized MAD), while in models ($M3$) and ($M4$) we use minimum distance estimators (MDE) to the Cramér-von-Mises distance.
The four examples were chosen for the following reasons:
In Example ($M1$), nothing has to be redefined. Estimation by MDE or MLE is straightforward: We define an object of class `NormLocationFamily` and generate some data.

```
R> (N <- NormLocationFamily(mean = 3))

An object of class "NormLocationFamily"
### name:        normal location family

### distribution:        Distribution Object of Class: Norm
 mean: 3
 sd: 1

### param:        An object of class "ParamFamParameter"
name:        loc
mean:        3

### props:
[1] "The normal location family is invariant under"
[2] "the group of transformations 'g(x) = x + loc'"
[3] "with location parameter 'loc'"

R> x <- r(N)(20)
```

We compute the MLE and the Cramér-von-Mises MDE using some (preliminary) method for the computation of the asymptotic covariance of the MDE.

```
R> MLEstimator(x,N)

Evaluations of Maximum likelihood estimate:
-------------------------------------------

  3.1823802
 (0.2236068)

R> MDEstimator(x,N,distance=CvMDist,
+              asvar.fct = distrMod:::.CvMMDCovariance)
```

```
Evaluations of Minimum CvM distance estimate:
---------------------------------------------
     mean
  3.2157975
 (0.3057117)
```

Example ($M2$) illustrates the use of a "parametric group model" in the sense of Lehmann (1983, Section 1.3, pp. 19–26), and as this model is quite non-standard, we use it to demonstrate some capabilities of our generating functions. Example ($M3$) illustrates the use of a predefined S4 class; specifically, class `GammaFamily`. In this case there are various equivalent parameterizations, which in our setup can easily be transformed into each other; see Section 3.2. Example ($M4$), also available in package **distrMod** as demo `censoredPois`, illustrates a situation where we have to set up a model completely anew.

1.3. Organization of the paper

We first explain some aspects of the specific way object orientation (OO) is realized in R. We then present the new model S4 classes and demonstrate how package **distrMod** can be used to compute minimum criterion estimators. The global options which may be set in our package and some general programming practices are given in the appendix.

2. Object orientation in S4

In R, OO is realized in the S3 class concept as introduced in Chambers (1993a,b) and by its successor, the S4 class concept, as developed in Chambers (1998, 1999, 2001) and described in detail in Chambers (2008). Of course, also R Development Core Team (2010b, Section 5) may serve as reference.
An account of some of the differences to standard OO may be found in Chambers and Temple Lang (2001), Bengtsson (2003), and Chambers (2006).
Using the terminology of Bengtsson (2003), mainstream software engineering (e.g. C++) uses *COOP* (class-object-oriented programming) style whereas the S3/S4 concept of R uses *FOOP* (function-object-oriented programming) style or, according to Chambers (2006), at least *F+COOP* (i.e. both styles).
In COOP style, methods providing access to or manipulation of an object are part of the object, while in FOOP style, they are not, but belong to *generic functions* – abstract functions which allow for arguments of varying type/class. A dispatching mechanism then decides on run-time which method best fits the *signature* of the function, that is, the types/classes of (a certain subset of) its arguments. `C++` has a similar concept, "overloaded functions" as discussed by Stroustrup (1997, Section 4.6.6).

In line with the different design of OO within R, some notions have different names in R context as well. This is in part justified by slightly different meanings; e.g., members in R are called *slots*, and constructors are called *generating functions*. In the case of the latter, the notion does mean something similar but not identical to a constructor: a generating function according to Chambers (2001) is a user-friendly wrapper to a call to `new()`, the actual constructor in the S4 system. In general it does not have the same flexibility as the full-fledged constructor in that some calling possibilities will still be reserved to a call to

`new()`.

Following the (partial) FOOP style of R, we sometimes have to deviate from best practice in mainstream OO, namely documenting the methods of each class hierarchy together as a group. Instead we document the corresponding particular methods in the help file for the corresponding generic.

Although the use of OO in the R context will certainly not be able to gain benefits using object identity, information hiding and encapsulation, the mere use of inheritance and polymorphism does provide advantages:

Polymorphism is a very important feature in interactively used languages as the user will not have to remember a lot of different function names but instead is able to say `plot` to many different objects of classes among which there need not be any inheritance structure. On the other hand, inheritance will make it possible to have a general (default) code which applies if nothing else is known while still any user may register his own particular method for a derived class, without interference of the authors of the class and generic function definitions. Of course, this could also be achieved by functional arguments, but using method dispatch we have much more control on the input and output types of the corresponding function. This is important, as common R functions neither have type checking for input arguments nor for return values. In addition to simple type checking we could even impose some refined checking by means of the S4 validity checking.

3. S4 classes: Models and parameters

3.1. Model classes

Models in Statistics and in R In Statistics, a probability model or shortly model is a family of probability distributions. More precisely, a subset $\mathcal{P} \subset \mathcal{M}_1(\mathcal{A})$ of all probability measures on some sample space $(\Omega, \mathcal{A})$. In case we are dealing with a parametric model, there is a finite-dimensional parameter domain Θ (usually an open subset of $\mathbb{R}^k$) and a mapping $\theta \mapsto P_\theta$, assigning each parameter $\theta \in \Theta$ a corresponding member of the family $\mathcal{P}$. If this parametrization is smooth, more specifically L_2-differentiable, see Rieder (1994, Section 2.3), we additionally have an L_2-derivative Λ_θ for each $\theta \in \Theta$; that is, some random variable (RV) in $L_2(P_\theta)$ and its corresponding (co)variance, the Fisher information $\mathcal{I}_\theta$. In most cases, $\Lambda_\theta = \frac{d}{d\theta} \log p_\theta$ (the classical scores) for p_θ the density of P_θ w.r.t. Lebesgue or counting measure.

One of the strengths of R (or more accurately of S) right from the introduction of S3 in Becker, Chambers, and Wilks (1988) is that models, more specifically [generalized] linear models (see functions `lm` and `glm` in package **stats**) may be explicitly formulated in terms of the language. The key advantage of this is grouping of relevant information, re-usability, and of course the formula interface (see `formula` in package **stats**) by which computations *on* the model are possible in S.

From a mathematical point of view however, these models are somewhat incomplete: In the case of `lm`, there is an implicit assumption of Gaussian errors, while in the case of `glm` only a limited number of explicit families and explicit link functions are "hard-coded". So in fact, again the user will not enter any distributional assumption.

Other models like the more elementary location and scale family (with general central distri-

bution) so far have not even been implemented.

With our distribution classes available from package **distr** we go ahead in this direction in package **distrMod**, although admittedly, up to now, we have not yet implemented any regression model or integrated any formula interface, but this will hopefully be done in the future.

Packages distr and distrEx Much of our infrastructure relies on our R packages **distr** and **distrEx** available on CRAN. Package **distr**, see Ruckdeschel *et al.* (2006, 2010), aims to provide a conceptual treatment of distributions 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, for evaluation of density, c.d.f. and quantile function of the corresponding distribution, respectively. All distributions of the **stats** package are implemented as subclasses of either `AbscontDistribution` or `DiscreteDistribution`, which themselves are again subclasses of `UnivariateDistribution`. As usual in stochastics, we identify distributions with RVs distributed accordingly. By means of these classes, we may automatically generate new objects of these classes for the laws of RVs under standard univariate mathematical transformations and under standard bivariate arithmetical operations acting on independent RVs. Here is a short example: We create objects of $\mathcal{N}(2, 1.69)$ and $\mathrm{Pois}(1.2)$ and convolve an affine transformation of them.

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

Distribution Object of Class: AbscontDistribution

R> plot(Z, cex.inner = 0.9)
```

The new distribution has corresponding slots `r`, `d`, `p` and `q`.

```
R> p(Z)(0.4)

[1] 0.002415387

R> q(Z)(0.3)

[1] 6.705068

R> r(Z)(5)

[1]  7.758469  6.764469  8.036305  7.842123 11.034121
```

Package **distrEx** extends **distr** by covering statistical functionals like expectation, variance or the median evaluated at distributions, as well as distances between distributions and basic support for multivariate and conditional distributions. E.g., using the distributions generated above, we can write

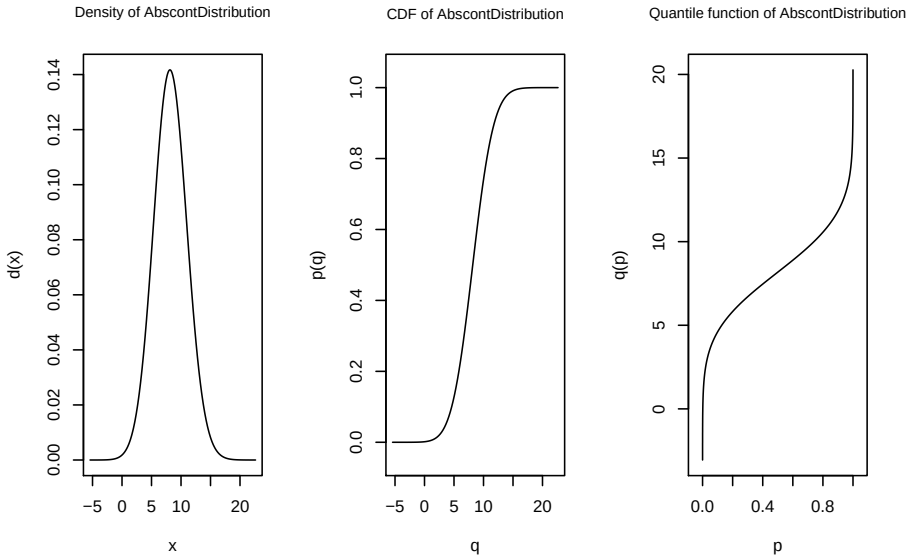

Figure 1: Plot of `Z`, an object of class `AbscontDistribution`.

```
R> library(distrEx)
R> E(N)

[1] 2

R> E(P)

[1] 1.2

R> E(Z)

[1] 8.200044

R> E(Z, fun=function(x) sin(x))

[1] 0.01937117
```

where `E(N)` and `E(P)` return the analytic value whereas the last two calls invoke some numerical computations.

Models in distrMod Based on class `Distribution` of package **distr** and its subclasses we define classes for families of probability measures in package **distrMod**. So far, we specialized this to parametric families of probability measures in class `ParamFamily`; see Figure 2. The concept however, also allows the derivation of subclasses for other (e.g. semiparametric) families of probability measures. In the case of L_2-differentiable parametric families we introduce several additional slots for scores Λ_θ and Fisher information $\mathcal{I}_\theta$. In particular, slot `L2deriv`

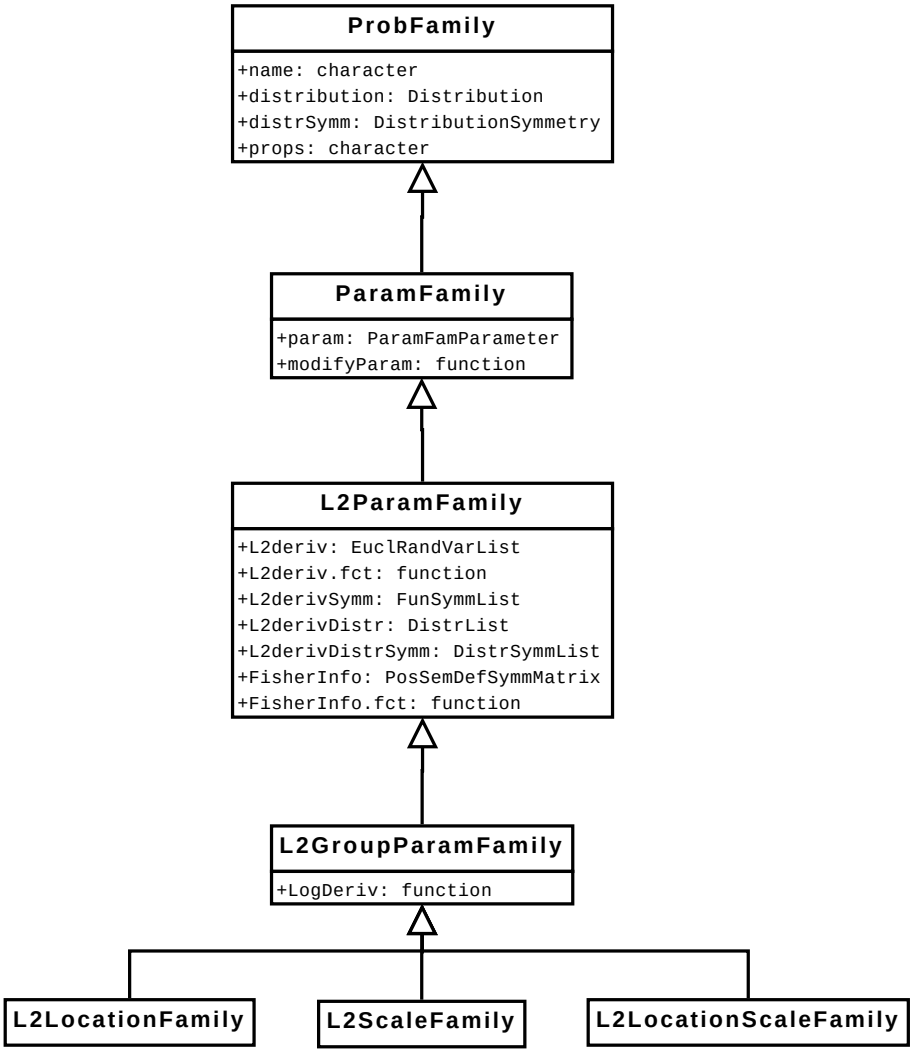

Figure 2: Inheritance relations and slots of the corresponding (sub-)classes for `ProbFamily` where we do not repeat inherited slots.

for the score function is of class `EuclRandVarList`, a class defined in package **RandVar** (Kohl and Ruckdeschel 2010a). The mother class `ProbFamily` is virtual and objects can only be created for all derived classes.

Class `ParamFamily` and all its subclasses have pairs of slots: actual value slots and functional slots, the latter following the COOP paradigm. The actual value slots like `distribution`, `param`, `L2deriv`, and `FisherInfo` are used for computations at a certain value of the parameter, while functional slots like `modifyParam`, `L2deriv.fct`, and `FisherInfo.fct` provide mappings $\Theta \to \mathcal{M}_1(\mathbb{B})$, $\theta \mapsto P_\theta$, $\Theta \to \bigcup_{\theta\in\Theta} L_2(P_\theta)$, $\theta \mapsto \Lambda_\theta$, and $\Theta \to \mathbb{R}^{k\times k}$, $\theta \mapsto \mathcal{I}_\theta$, respectively, and are needed to modify the actual parameter of the model, or to move the model from one parameter value to another. The different modifications due after a change in the parameter are grouped in S4 method `modifyModel`.

Generating functions Generating objects of class `L2ParamFamily` and derived classes

involves filling a considerable number of slots. Hence, for convenience, there are several user-friendly generating functions as displayed in Table 1.

Name	Family
`ParamFamily`	general parametric family
`L2ParamFamily`	general L_2 differentiable parametric family
`L2LocationFamily`	general L_2 differentiable location family
`L2LocationUnknownScaleFamily`	general L_2 differentiable location family with unknown scale (nuisance parameter)
`L2ScaleFamily`	general L_2 differentiable scale family
`L2ScaleUnknownLocationFamily`	general L_2 differentiable scale family with unknown location (nuisance parameter)
`L2LocationScaleFamily`	general L_2 differentiable location and scale family
`BetaFamily`	beta family
`BinomFamily`	binomial family
`GammaFamily`	gamma family
`PoisFamily`	Poisson family
`ExpScaleFamily`	exponential scale family
`GumbelLocationFamily`	Gumbel location family
`LnormScaleFamily`	log-normal scale family
`NormLocationFamily`	normal location family
`NormLocationUnknownScaleFamily`	normal location family with unknown scale (nuisance parameter)
`NormScaleFamily`	normal scale family
`NormScaleUnknownLocationFamily`	normal scale family with unknown location (nuisance parameter)
`NormLocationScaleFamily`	normal location and scale family

Table 1: Generating functions for `ParamFamily` and derived classes.

Examples In order to follow our running example ($M2$), consider the following code: we first define the (non-standard) central distribution `myD` and then generate the location and scale model. For the central distribution, the corresponding standardizing constant could be expressed in closed form in terms of the gamma function, but instead we present the more general approach in which (by argument `withS`) standardization to mass 1 is enforced by numerical integration.

```
R> myD <- AbscontDistribution(d = function(x) exp(-abs(x)^3),
+                             withS = TRUE)
```

The logarithmic derivative of the density $d/dx \log p(x)$ is determined by numerical differentiation (but could have been given explicitly, too, by specifying argument `LogDeriv`). The L_2-derivative is calculated automatically using this derivative, and the corresponding Fisher information is then determined by numerical integration (but could also be specified explicitly using argument `FisherInfo.0`); see also the help file for function `L2LocationScaleFamily`. In this case we have checked accuracy to be of order $1e-6$.

```
R> (myFam <- L2LocationScaleFamily(loc = 3, scale = 2,
+                           name = "location and scale family",
+                           centraldistribution = myD))
```

```
An object of class "L2LocationScaleFamily"
### name:        location and scale family

### distribution:        Distribution Object of Class: AffLinAbscontDistribution

### param:        An object of class "ParamFamParameter"
name:        location and scale
loc:        3
scale:        2

### props:
[1] "The location and scale family is invariant under"
[2] "the group of transformations 'g(x) = scale*x + loc'"
[3] "with location parameter 'loc' and scale parameter 'scale'"
```

In the already implemented gamma family of example ($M3$), a corresponding object for this model is readily defined as

```
R> (G <- GammaFamily(scale = 1, shape = 2))

An object of class "GammaFamily"
### name:        Gamma family

### distribution:        Distribution Object of Class: Gammad
 shape: 2
 scale: 1

### param:        An object of class "ParamFamParameter"
name:        scale and shape
scale:        1
shape:        2

### props:
[1] "The Gamma family is scale invariant via the parametrization"
[2] "'(nu,shape)=(log(scale),shape)'"
```

In example ($M4$), we have to set up the model completely anew. Still, it is not too complicated, as we may use the generating function `L2ParamFamily` as illustrated in the following code:

```
R> CensoredPoisFamily <- function(lambda = 1, trunc.pt = 2){
+      name <- "Censored Poisson family"
+      distribution <- Truncate(Pois(lambda = lambda),
+                               lower = trunc.pt)
+      param0 <- lambda
+      names(param0) <- "lambda"
+      param <- ParamFamParameter(name = "positive mean",
+                                 main = param0,
```

```
+                                fixed = c(trunc.pt=trunc.pt))
+      modifyParam <- function(theta){
+                          Truncate(Pois(lambda = theta),
+                                   lower = trunc.pt)}
+      startPar <- function(x,...) c(.Machine$double.eps,max(x))
+      makeOKPar <- function(param){
+          if(param<=0) return(.Machine$double.eps)
+          return(param)
+      }
+      L2deriv.fct <- function(param){
+          lambda <- main(param)
+          fct <- function(x){}
+          body(fct) <- substitute({
+                           x/lambda-ppois(trunc.pt-1,
+                                          lambda = lambda,
+                                          lower.tail=FALSE)/
+                                    ppois(trunc.pt,
+                                          lambda = lambda,
+                                          lower.tail=FALSE)},
+                                    list(lambda = lambda))
+          return(fct)
+      }
+      res <- L2ParamFamily(name = name,
+                           distribution = distribution,
+                           param = param,
+                           modifyParam = modifyParam,
+                           L2deriv.fct = L2deriv.fct,
+                           startPar = startPar,
+                           makeOKPar = makeOKPar)
+      res@fam.call <- substitute(CensoredPoisFamily(lambda = l,
+                                                    trunc.pt = t),
+                                 list(l = lambda, t = trunc.pt))
+      return(res)
+  }
R> (CP <- CensoredPoisFamily(3,2))


An object of class "L2ParamFamily"
### name:        Censored Poisson family

### distribution:        Distribution Object of Class: LatticeDistribution

### param:        An object of class "ParamFamParameter"
name:        positive mean
lambda:        3
fixed part of param.:
        trunc.pt:        2
```

Function `ParamFamParameter()` generates the parameter of this class, and "movements" of the model when changing the parameter are realized in function `modifyParam`. More on this will be described in Section 3.2. Functions `startPar` and `makeOKPar` are helper functions for estimation which are discussed at the end of Section 4.2. The more difficult parts in the implementation concern the distribution of the observations – here package **distr** with its powerful methods for automatic generation of image distributions is very helpful – and the L_2-derivative as a function, `L2deriv.fct`. Here some off-hand calculations are inevitable.

Plotting There are also quite flexible `plot` methods for objects of class `ParamFamily`. In the case of `L2ParamFamily` the default plot consists of density, cumulative distribution function (cdf), quantile function and scores. An example is given in Figure 3; for details see the help page of `plot` in **distrMod**.

```
R> layout(matrix(c(1,1,2,2,3,3,4,4,4,5,5,5), nrow = 2,
+         byrow = TRUE))
R> plot(myFam, mfColRow = FALSE, cex.inner = 1,
+       inner = c("density", "cdf", "quantile function",
+                 "location part", "scale part"))
```

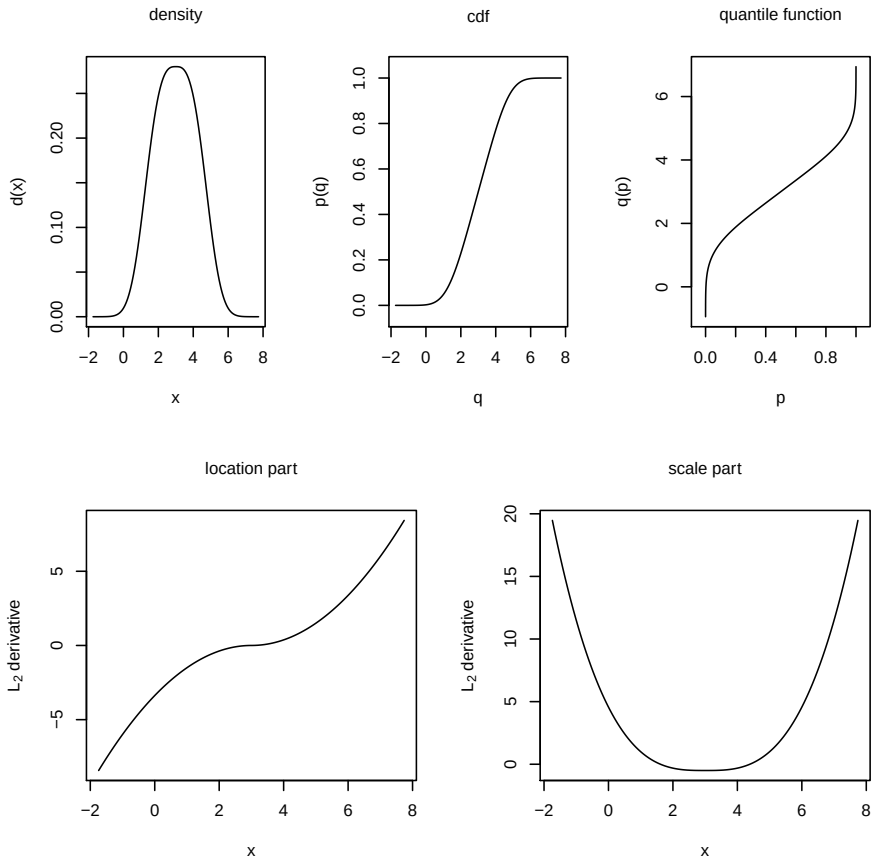

Figure 3: Plot of `L2ParamFamily` object.

3.2. Parameter in a parametric family: class `ParamFamParameter`

At first glance, the fact that the distribution classes of package **distr** already contain a slot `param` which is either `NULL` or of S4 class `Parameter` could lead us to question why we need the infrastructure provided by package **distrMod**. In fact, for class `ParamFamily`, we define S4-class `ParamFamParameter` as subclass of class `Parameter` since we additionally want to allow for partitions and transformations.
As always, there is a generating function with the same name for this class; that is, a function `ParamFamParameter()` which already showed up in the code for the implementation of model ($M4$).
This new class is needed since in many (estimation) applications, it is not the whole parameter of a parametric family which is of interest, but rather parts of it, while the rest of it is either known and fixed or has to be estimated as a nuisance parameter. In other situations, we are interested in a (smooth) transformation of the parameter. All these cases are realized in the design of class `ParamFamParameter` which has slots `name`, the name of the parameter, `main`, the interesting aspect of the parameter, `nuisance`, an unknown part of the parameter of secondary interest, but which has to be estimated, for example for confidence intervals, and `fixed`, a known and fixed part of the parameter (see also Figure 4). In addition, it has a

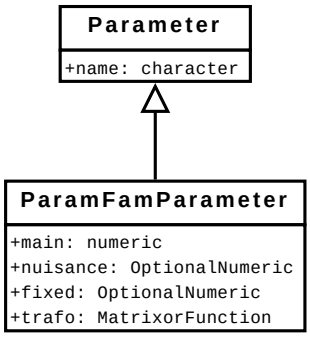

Figure 4: Inheritance relations and slots of `ParamFamParameter` where we do not repeat inherited slots.

slot `trafo` (an abbreviation of "transformation") which is also visible in class `Estimate` (see Figure 5). Slot `trafo` for instance may be used to realize partial influence curves, see Rieder (1994, Definition 4.2.10), where one is only interested in some possibly lower dimensional smooth (not necessarily linear or coordinate-wise) aspect/transformation τ of the parameter θ.
To be coherent with the corresponding *nuisance* implementation, we use the following convention: The full parameter θ is split up coordinate-wise into a main parameter θ', a nuisance parameter θ'', and a fixed, known part θ'''.
Without loss of generality, we restrict ourselves to the case that transformation τ only acts on the main parameter θ' – in case we want to transform the whole parameter, we have to assume both nuisance parameter θ'' and known part of the parameter θ''' have length zero.

Implementation Slot `trafo` can contain either 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 $\tau(\theta)$, it will also return the corresponding derivative matrix $\frac{\partial}{\partial\theta}\tau(\theta)$. More specifically, the return value of this function is a list with entries `fval`, the function value $\tau(\theta)$, and `mat`, the derivative matrix.
In the case that `trafo` is a matrix D, we interpret it as such a derivative matrix $\frac{\partial}{\partial\theta}\tau(\theta)$, and correspondingly, $\tau(\theta)$ is the linear mapping $\tau(\theta) = D\,\theta$.
According to the signature, the return value of accessor function/method `trafo` varies. For signatures `(ParamFamily,ParamFamParameter)`, `(Estimate,ParamFamParameter)`, and `(Estimate,missing)`, the result is a list with entries `fct`, the function τ, and `mat`, the matrix $\frac{\partial}{\partial\theta}\tau(\theta)$. Function τ will then return a list with entries `fval` and `mat` mentioned above. For signatures `(ParamFamily,missing)` and `(ParamFamParameter,missing)`, `trafo` will just return the corresponding matrix.

Movements in parameter space and model space Our implementation of models has both function components providing mappings $\theta \mapsto \text{Component}(\theta)$ (like `L2deriv.fct`) and components evaluated at an actual parameter value θ (like `L2deriv`). When we "move" a model object from θ to θ', i.e. when we change the reference parameter of this model object from θ to θ', the latter components have to be modified accordingly. To this end, there are `modifyModel` methods for classes `L2ParamFamily`, `L2LocationFamily`, `L2ScaleFamily`, `L2LocationScaleFamily`, `ExpScaleFamily`, and `GammaFamily`, where the second argument to dispatch on has to be of class `ParamFamParameter` but this probably will be extended in the future. The code for example ($M3$), that is to signature `model="GammaFamily"`, for instance, reads

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

Internally, the default `modifyModel` method makes use of slots `modifParam` (to move the distribution of the observations) and `L2deriv.fct` (to move the L_2-derivative). For internal reasons, these two functions have different implementations of the parameter as argument: `L2deriv.fct`, only internally used by our routines, requires an instance of class `ParamFamParameter`. In contrast to this, `modifyParam` uses a representation of the parameter as slot `main` of class `ParamFamParameter`; that is, simply as a numeric vector, because its results are passed on to non-**distrMod**-code. This inconsistency, which also holds for the passed-on functional slots `makeOKPar` and `startPar`, might confuse some users and will hence probably be changed in a subsequent package version.

Example Our implementation of the gamma family follows the parametrization of the R core functions `d/p/q/rgamma`. Hence, we use the parameters `("scale","shape")` (in this order). Package **MASS** (Venables and Ripley 2002) however uses the (equivalent) parametrization

("shape","rate"), where rate=1/scale. To be able to compare the results obtained for the MLE by fitdistr and by our package, we therefore use the corresponding transformation $\tau(\text{scale}, \text{shape}) = (\text{shape}, 1/\text{scale})$. More specifically, this can be done as follows

```
R> mtrafo <- function(x){
+       nms0 <- c("scale","shape")
+       nms <- c("shape","rate")
+       fval0 <- c(x[2], 1/x[1])
+       names(fval0) <- nms
+       mat0 <- matrix(c(0, -1/x[1]^2, 1, 0), nrow = 2,
+                      dimnames = list(nms,nms0))
+       list(fval = fval0, mat = mat0)
+  }
R> (G.MASS <- GammaFamily(scale = 1, shape = 2, trafo = mtrafo))

An object of class "GammaFamily"
### name:        Gamma family

### distribution:        Distribution Object of Class: Gammad
 shape: 2
 scale: 1

### param:        An object of class "ParamFamParameter"
name:        scale and shape
scale:        1
shape:        2

### props:
[1] "The Gamma family is scale invariant via the parametrization"
[2] "'(nu,shape)=(log(scale),shape)'"
```

4. Minimum criterion estimation

4.1. Implementations in R so far

To better appreciate the generality of our object oriented approach, let us contrast it with the two already mentioned implementations:

fitdistr Function fitdistr comes with arguments: x, densfun, start (and ...) and returns an object of S3-class fitdistr which is a list with components estimate, sd, and loglik. As starting estimator start in case ($M2$), we select median and MAD, where for the latter we need a consistency constant to obtain a consistent estimate for scale. In package **distrMod** this is adjusted automatically.
Due to symmetry about the location parameter, this constant is just the inverse of the upper quartile of the central distribution, obtainable via

```
R> (mad.const <- 1/q(myD)(0.75))
```

```
[1] 2.187639
```

Then, function `fitdistr` from package **MASS** may be called as

```
R> set.seed(19)
R> x <- r(distribution(myFam))(50)
R> mydf <- function(x, loc, scale){
+      y <- (x-loc)/scale; exp(-abs(y)^3)/scale
+  }
R> Med <- median(x)
R> MAD <- mad(x, constant = mad.const)
R> c(Med, MAD)

[1] 3.088471 1.803579

R> fitdistr(x, mydf, start = list("loc" = Med, "scale" = MAD))

      loc        scale
  3.1636282   1.6053346
 (0.1269680) (0.1310642)
```

mle Function `mle` from package **stats4** on the other hand, has arguments `minuslogl` (without data argument!), `start`, `method` (and `...`) and returns an object of S4-class `mle` with slots `call`, `coef`, `full`, `vcov`, `min`, `details`, `minuslogl`, and `method`. The MLE for model $(M2)$ may then be computed via

```
R> ll <- function(loc,scale){-sum(log(mydf(x,loc,scale)))}
R> mle(ll, start = list("loc" = Med, "scale" = MAD))

Call:
mle(minuslogl = ll, start = list(loc = Med, scale = MAD))

Coefficients:
     loc    scale
3.163628 1.605335
```

There are further packages with implementations for MLE like **bbmle** (Bolker 2010), **fitdistrplus** (Delignette-Muller, Pouillot, Denis, and Dutang 2010), and **maxLik** (Toomet and Henningsen 2010). Package **bbmle** provides modifications and extensions for the `mle` classes of the **stats4** package. The implementation is very similar to package **stats4** and the computation of the MLE is based on function `optim`. As the name of package **fitdistrplus** already suggests, it contains an extension of the function `fitdistr` with a very similar implementation. That is, there are analogous `if` clauses like in function `fitdistr` and if none of the special cases apply, numerical optimization via `optim` is performed. In addition to `fitdistr`, a user-supplied function can be used for optimization. Package **maxLik** includes tools for computing MLEs by means of numerical optimization via functions `optim` and `nlm`, respectively.
There is also at least one package which can be used to compute MDEs. With function `mde` of

package **actuar** (Dutang, Goulet, and Pigeon 2008) one can minimize the Cramér-von-Mises distance for individual and grouped data. Moreover, for grouped data a modified χ^2-statistic as well as the squared difference between the theoretical and empirical limited expected value can be chosen. The optimization is again performed by function `optim`.

4.2. Estimators in package distrMod

In our framework, it is possible to implement a general, dispatchable minimum criterion estimator (MCE); that is, an estimator minimizing a criterion incorporating the observations / the empirical distribution and the model. Examples of MCEs comprise MLEs (with neg. Log-likelihood as criterion) and MDEs where the criterion is some distance between the empirical distribution and P_θ.

We have implemented MCEs in form of the function `MCEstimator`, and also provide functions `MDEstimator` and `MLEstimator` – essentially just wrapper functions for `MCEstimator`.
`MCEstimator` takes as arguments the observations `x`, the parametric family `ParamFamily`, the criterion, and a starting parameter `startPar` (and again some optional ones). It returns an object of class `MCEstimate` (a subclass of class `Estimate`) with slots `estimate`, `criterion`, `samplesize`, `asvar` (and some more); for more detailed information see the help page of class `MCEstimate`.

The main achievement of our approach is that estimators are available for *any* object of class `ParamFamily` (e.g., Poisson, beta, gamma and many more). At the same time, using S4 inheritance this generality does not hinder specialization to particular models: internal dispatch on run-time according to argument `ParamFamily` allows the definition of new specialized methods *without modifying existing code*, a feature whose importance should not be underestimated when we are dealing with distributed collaborative package development.
Specialized methods for the computation of these MCEs can easily be established as described below. In particular, this can be done externally to our package, and nonetheless all our infrastructure is fully available right from the beginning; i.e., a unified interfacing/calling function, unified plotting, printing, interface to confidence intervals etc.

Limitations of the other implementations In principle, the last few points could already have been realized within the elder S3 approach as realized in `fitdistr`. However, there are some subtle limitations which apply to function `fitdistr` and function `mle` as well as to the other implementations: while the general R optimization routines/interfaces `optim`/`optimize`/`nlm` clearly stand out for their generality and their implementation quality, "general" numerical optimization to determine an MLE in almost all cases is bound to the use of these routines.
There are cases however, where either other optimization techniques could be more adequate (e.g. in estimating the integer-valued degrees-of-freedom parameter in a χ^2-distribution), or where numerical optimization is not necessary, because optima can be determined analytically. The first case may to a certain extend be handled using package **fitdistrplus** where one can specify a user-supplied function for optimization. However, for the second case it would be necessary to change existing code; e.g., convince the authors Brian Ripley and Bill Venables to append another `if`-clause in `fitdistr`.
The above is also true for MDE. The numerical optimization in the case of function `mde` of package **actuar** is restricted to `optim` and it would be necessary to change the existing code if one for example wants to use another distance (e.g. Kolmogorov distance).

As already mentioned, the approach realized in package **distrMod** does not have these limitations. As an example one can quite easily insert "intermediate" layers like group models; e.g., location (and/or scale) models. In the S4-inheritance structure for class `ParamFamily`, one can define a corresponding subclass $\mathcal{S}$ and "intermediate" general methods for $\mathcal{S}$ which are more special than `optim`/`optimize`/`nlm`, but still apply by default for more than one model, and which – if desired – could then again be overridden by more special methods applying to subclasses of $\mathcal{S}$.
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.

```
R> negLoglikelihood <- function(x, Distribution){
+      res <- -sum(log(Distribution@d(x)))
+      names(res) <- "Negative Log-Likelihood"
+      return(res)
+  }
R> MCEstimator(x = x, ParamFamily = myFam,
+               criterion = negLoglikelihood)

Evaluations of Minimum criterion estimate:
------------------------------------------
      loc        scale
  3.1635895   1.6053251
 (0.1303646) (0.1310812)
```

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

Instance-individually Using the first layer, we may specify model-individual starting values/search intervals by slot `startPar` of class `ParamFamily`, pass on special control parameters to functions `optim`/`optimize` by a `...` argument in function `MCEstimator`, and we may enforce valid parameter values by specifying function slot `makeOKPar` of class `ParamFamily`. In addition, one can specify a penalty value penalizing invalid parameter values. For an example, see the code to example ($M4$) on page 10.

Class-individually In some situations, one would rather like to define rules for groups of models or to be even more flexible. This can be achieved using the class-individual layer: In order to use method dispatch to find the "right" function to determine the MCE, we define subclasses to class `ParamFamily` as e.g., in the case of class `PoisFamily`. In general, these subclasses will not have any new slots, but merely serve as the basis for a refined method dispatching. As an example, the code to define class `PoisFamily` simply is

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

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

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

Specialized methods may then be defined for these subclases. So far, in package **distrMod** we have particular `validParameter` methods for classes `ParamFamily`, `L2ScaleFamily`,

`L2LocationFamily` and `L2LocationScaleFamily`; e.g., the code to signature `L2ScaleFamily` simply is

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

Class-individual routines are realized by calling `mceCalc` and `mleCalc` within function `MCEstimator` and `MLEstimator`, respectively; e.g.,

```
R> setMethod("mleCalc", signature(x = "numeric",
+                                 PFam = "NormLocationScaleFamily"),
+            function(x, PFam){
+                n <- length(x)
+                c(mean(x), sqrt((n-1)/n)*sd(x))
+            })
```

and the maximum likelihood estimator in example $(M2)$ can easily be computed as follows.

```
R> MLEstimator(x = x, ParamFamily = myFam)

Evaluations of Maximum likelihood estimate:
-------------------------------------------
      loc        scale
  3.1635895   1.6053251
 (0.1303646) (0.1310812)
```

Similarly we could evaluate the Kolmogorov-MDE in this model:

```
R> MDEstimator(x = x, ParamFamily = myFam,
+              distance = KolmogorovDist)

Evaluations of Minimum Kolmogorov distance estimate:
----------------------------------------------------
estimate:
     loc    scale
3.186672 1.713888
```

Note that the last two calls would be identical (only replacing argument `myFam`) for examples $(M3)$ and $(M4)$.

Functions mceCalc and mleCalc Both `mceCalc` and `mleCalc` dispatch according to their arguments `x` and `PFam`. For `mceCalc`, so far there is only a method for signature

(numeric,ParamFamily), while mleCalc already has several particular methods for argument PFam of classes ParamFamily, BinomFamily, PoisFamily, NormLocationFamily, NormScaleFamily, and NormLocationScaleFamily. To date, in both mceCalc and mleCalc, argument x must inherit from class numeric, but we plan to allow for more general classes (e.g. data.frames) in subsequent versions. Note that for technical reasons, mleCalc must have an extra ... argument to cope with different callings from MLEstimator. Additional arguments are of course possible. The return value must be a list with prescribed structure. To this end, function meRes() can be used as a helper to produce this structure. For example the mleCalc-method for signature numeric,NormScaleFamily is

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

Coercion to class mle We also provide a coercion to class mle from package **stats4**, hence making profiling by the profile-method therein possible. In order to be able to do so, we need to fill a functional slot criterion.fct of class MCEstimate. In many examples this is straightforward, but in higher dimensions, helper function get.criterion.fct can be useful; e.g., it handles the general case for signature ParamFamily.
The values of functions MCEstimator, MDEstimator, and MLEstimator are objects of S4-class MCEstimate which inherits from S4-class Estimate; see Figure 5 for more details.

4.3. Confidence intervals and profiling

We also provide particular methods for functions confint and, as already mentioned in the previous section, profile of package **stats**. Moreover, by adding an argument method to the signature of confint, we gain more flexibility. Note that the addition of an extra argument was only possible by masking method confint. This masking is done in a way that it reproduces exactly the **stats** behaviour when called with the corresponding arguments, however. This additional argument is for example used in package **ROptEst** (Kohl and Ruckdeschel 2010b) where one can determine robust (asymptotic) confidence intervals which are uniformly valid on neighborhoods and may incorporate bias in various ways. Also, in principle, one-sided confidence intervals are possible, as well as confidence intervals produced by other techniques like bootstrap.

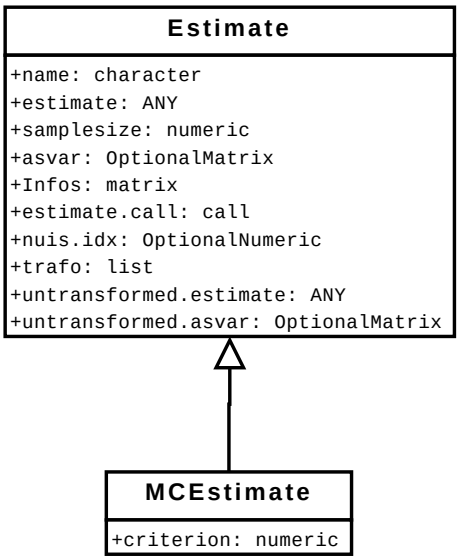

Figure 5: Inheritance relations and slots of the corresponding (sub-)classes for `Estimate` where we do not repeat inherited slots.

To make confidence intervals available for objects of class `MCEstimate`, there is a method `confint`, which produces confidence intervals (of class `Confint`). As an example consider the following: We first generate some data and determine the Cramér-von-Mises MDE as starting estimator for `fitdistr`. We then compute the MLE using `fitdistr` and `MLEstimator`.

```
R> set.seed(19)
R> y <- rgamma(50, scale = 3, shape = 2)
R> (MDest <- MDEstimator(x = y, ParamFamily = G.MASS,
+                        distance = CvMDist))

Evaluations of Minimum CvM distance estimate:
---------------------------------------------
estimate:
    shape      rate
2.5992847 0.3831308

R> fitdistr(x = y, densfun = "gamma",
+           start = list("shape" = estimate(MDest)[1],
+                        "rate" = estimate(MDest)[2]))

     shape         rate
  2.61870805   0.39973025
 (0.49401949) (0.08310676)

R> (res <- MLEstimator(x = y, ParamFamily = G.MASS))
```

```
Evaluations of Maximum likelihood estimate:
-------------------------------------------
     shape         rate
  2.61752800   0.39953520
 (0.49379501) (0.08307007)

R> (ci <- confint(res))

A[n] asymptotic (CLT-based) confidence interval:
          2.5 %    97.5 %
shape 1.6497076 3.5853484
rate  0.2367209 0.5623495
```

And we can do some profiling. The results are given in Figure 6.

```
R> par(mfrow=c(2,1))
R> plot(profile(res))
```

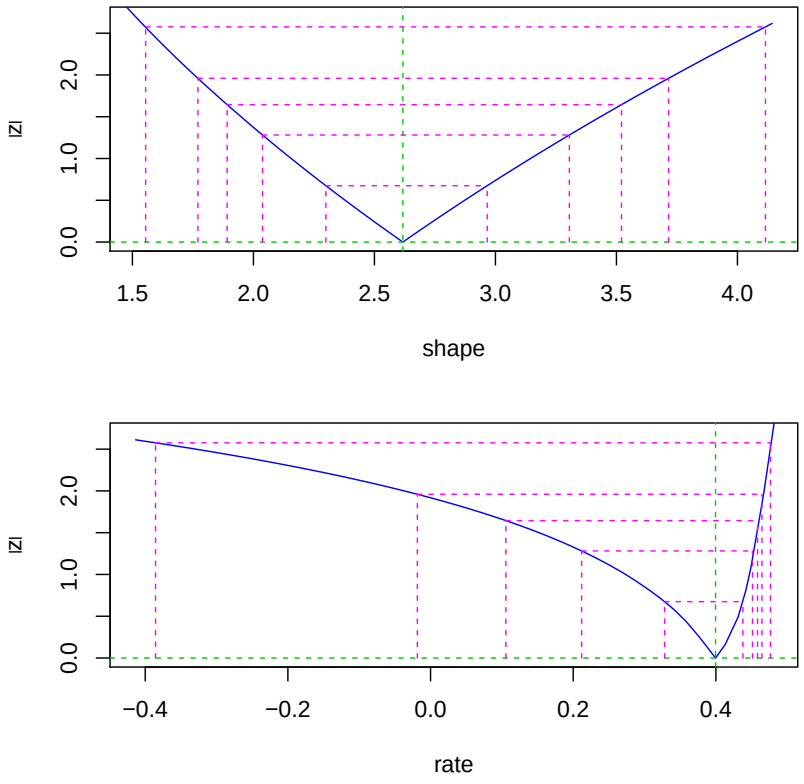

Figure 6: Profiling: behavior of objective function near the solution.

Note again that only minimal changes in the preceding **distrMod**-code would be necessary to also apply the code to examples $(M3)$ and $(M4)$.

4.4. Customizing the level of detail in output

For class `Confint` as well as for class `Estimate` we have particular `show` and `print` methods where you may scale the output. This scaling can either be done by setting global options with `distrModOptions` (see Appendix A), or, in the case of `print`, locally by the extra argument `show.details`. The default value of `show.details` is `"maximal"`.
Note that this departure from functional programming style is necessary, as `show` does not allow for additional arguments.

Value of `show.details`	**Object of class** `MCEstimate`	**Object of class** `Confint`
`"minimal"`	parameter estimates and estimated standard errors	lower and upper confidence limits for each parameter
`"medium"`	call, sample size, asymptotic (co)variance, and criterion	call, sample size, and type of estimator
`"maximal"`	untransformed estimate, asymptotic (co)variance of untransformed estimate, transformation, and derivative matrix	transformation, and derivative matrix

Table 2: The level of detail in output. In case of `"medium"` and `"maximal"` only the additional output is specified.

5. Conclusions

Package **distrMod** represents the most flexible implementation of minimum criterion estimators for univariate distributions, including maximum likelihood and minimum distance estimators, available in R so far. These estimators are provided for any object of S4 class `ParamFamily` and by S4 inheritance can be adapted to particular models without modifying existing code. In addition it contains infra-structure in terms of unified interfacing/calling functions, unified plotting, printing, interface to confidence intervals and more.

References

Becker RA, Chambers JM, Wilks AR (1988). *The New S language. A Programming Environment for Data Analysis and Graphics*. Wadsworth & Brooks/Cole Advanced Books & Software, Pacific Grove, CA.

Bengtsson H (2003). "The **R.oo** package - Object-Oriented Programming with References Using Standard R Code." In K Hornik, F Leisch, A Zeileis (eds.), *Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003)*. Vienna, Austria. ISSN 1609-395X. URL http://www.ci.tuwien.ac.at/Conferences/DSC-2003/.

Bolker B (2010). ***bbmle**: Tools for General Maximum Likelihood Estimation*. R package version 0.9.5.1, URL http://CRAN.R-project.org/package=bbmle.

Chambers JM (1993a). "Classes and Methods in S. I: Recent Developments." *Computational Statistics*, **8**(3), 167–184.

Chambers JM (1993b). "Classes and Methods in S. II: Future Directions." *Computational Statistics*, **8**(3), 185–196.

Chambers JM (1998). *Programming with Data. A Guide to the S Language*. Springer-Verlag.

Chambers JM (1999). "Computing with Data: Concepts and Challenges." *The American Statistician*, **53**(1), 73–84.

Chambers JM (2001). "Classes and Methods in the S Language." *Technical report*, omegahat.org. URL http://www.omegahat.org/RSMethods/Intro.pdf.

Chambers JM (2006). "How S4 Methods Work." *Technical report*, r-project.org. URL http://developer.r-project.org/howMethodsWork.pdf.

Chambers JM (2008). *Software for Data Analysis. Programming with R*. Springer-Verlag.

Chambers JM, Temple Lang D (2001). "Object Oriented Programming in R." *R News*, **1**(3), 17–19. URL http://CRAN.R-project.org/doc/Rnews/.

Delignette-Muller ML, Pouillot R, Denis JB, Dutang C (2010). ***fitdistrplus**: Help to Fit of a Parametric Distribution to Non-Censored or Censored Data*. R package version 0.1-3, URL http://CRAN.R-project.org/package=fitdistrplus.

Dutang C, Goulet V, Pigeon M (2008). "**actuar**: An R Package for Actuarial Science." *Journal of Statistical Software*, **25**(7), 1–37. URL http://www.jstatsoft.org/.

Gentleman R (2008). *R Programming for Bioinformatics*. Chapman & Hall/CRC.

Kohl M, Ruckdeschel P (2010a). ***RandVar**: Implementation of Random Variables*. R package version 0.7, URL http://robast.r-forge.r-project.org/.

Kohl M, Ruckdeschel P (2010b). ***ROptEst**: Optimally Robust Estimation*. R package version 0.7, URL http://robast.r-forge.r-project.org/.

Kohl M, Ruckdeschel P (2010c). "R Package **distrMod**: S4 Classes and Methods for Probability Models." *Journal of Statistical Software*, **35**(10), 1–27. URL http://www.jstatsoft.org/v35/i10/.

Lehmann E (1983). *Theory of Point Estimation*. John Wiley & Sons.

R Development Core Team (2010a). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org.

R Development Core Team (2010b). *R Language definition – Version 2.11.1*. URL http://CRAN.R-project.org/doc/manuals/R-lang.pdf.

Rieder H (1994). *Robust Asymptotic Statistics*. Springer-Verlag.

Ruckdeschel P, Kohl M, Stabla T, Camphausen F (2006). "S4 Classes for Distributions." *R News*, **6**(2), 2–6. URL http://CRAN.R-project.org/doc/Rnews.

Ruckdeschel P, Kohl M, Stabla T, Camphausen F (2010). *S4 Classes for Distributions—a Manual for Packages* ***distr****,* ***distrSim****,* ***distrTEst****,* ***distrEx****,* ***distrMod****, and* ***distrTeach****.* Vignette contained in package **distrDoc**, URL http://CRAN.R-project.org/package=distrDoc.

Stroustrup B (1997). *The C++ Programming Language*. 3rd edition. Addison-Wesley.

Theußl S, Zeileis A (2009). "Collaborative Software Development Using R-Forge." *The R Journal*, **1**(1), 9–14. URL http://journal.R-project.org/2009-1/RJournal_2009-1_Theussl+Zeileis.pdf.

Toomet O, Henningsen A (2010). ***maxLik****: Maximum Likelihood Estimation*. With contributions from Spencer Graves. R package version 0.7-2, URL http://CRAN.R-project.org/package=maxLik.

Venables W, Ripley B (2002). *Modern Applied Statistics with S*. 4th edition. Springer-Verlag. URL http://www.stats.ox.ac.uk/pub/MASS4.

A. Global options

Analogously to the `options` command in R package **base** one may specify a number of global "constants" to be used within the package via `distrModoptions`/`getdistrModOption`. These include

- `use.generalized.inverse.by.default` which is a logical variable giving the default value for argument `generalized` of our method `solve` in package **distrMod**. This argument decides whether our method `solve` is to use generalized inverses if the original `solve`-method from package **base** fails. If the option is set to `FALSE`, in the case of failure, and unless argument `generalized` is not explicitly set to `TRUE`, `solve` will throw an error as is the **base**-method behavior. The default value of the option is `TRUE`.
- `show.details` controls the level of detail of method `show` for objects of the classes of the **distr** family of packages. Possible values are
  - `"maximal"`: all information is shown
  - `"minimal"`: only the most important information is shown
  - `"medium"`: somewhere in the middle; see actual `show`-methods for details.

  The default value is `"maximal"`. For more details see Table 2.

B. Following good programming practices

There are several new S4 classes in package **distrMod**. With respect to inspection and modification of the class slots we follow the suggestion of Section 3.4 in Gentleman (2008). That is, there are accessor functions/methods for all slots included in the new classes. Moreover, we implemented replacement functions/methods for those slots which are intended to be modifiable by the user.

One could also use the `@`-operator to modify or access slots. However, this operator relies on the implementation details of the class, and hence, this may lead to difficulties if the class implementation has to be changed. In addition, as no checking is invoked one may easily produce inconsistent objects.

We also implemented generating functions for all non-virtual classes which shall ease the definition of objects for the user; see Chambers (1998, Section 1.6). These functions in most cases have the same name as the class. By obtaining convenience via generating functions and not via new `initialize`-methods, we see the advantage that the default `initialize`-methods called by means of `new` remain valid and can be used for programming.

Finally, there are `show`-methods for all new S4 classes which display the essential information of instantiated objects; see Chambers (1998, Section 1.6).

Acknowledgments

Both authors contributed equally to this work. We thank the referees and the associate editor for their helpful comments.

Affiliation:

Matthias Kohl
Hochschule Furtwangen
Fakultät Maschinenbau und Verfahrenstechnik
Jakob-Kienzle-Strasse 17
78054 Villingen-Schwenningen
E-mail: Matthias.Kohl@hs-furtwangen.de

Peter Ruckdeschel
TU Kaiserslautern
FB Mathematik
P.O.Box 3049
67653 Kaiserslautern, Germany
and
Fraunhofer-ITWM
Fraunhofer-Platz 1
67663 Kaiserslautern, Germany
E-mail: Peter.Ruckdeschel@itwm.fraunhofer.de