distr6: R6 Object-Oriented Probability Distributions Interface in R

by Raphael Sonabend and Franz J. Király

Abstract **distr6** is an object-oriented (OO) probability distributions interface leveraging the extensibility and scalability of R6 and the speed and efficiency of **Rcpp**. Over 50 probability distributions are currently implemented in the package with 'core' methods including density, distribution, and generating functions, and more 'exotic' ones including hazards and distribution function anti-derivatives. In addition to simple distributions, **distr6** supports compositions such as truncation, mixtures, and product distributions. This paper presents the core functionality of the package and demonstrates examples for key use-cases. In addition this paper provides a critical review of the object-oriented programming paradigms in R and describes some novel implementations for design patterns and core object-oriented features introduced by the package for supporting **distr6** components.

Introduction

Probability distributions are an essential part of data science, underpinning models, simulations, and inference. Hence, they are central to computational data science. With the advent of modern machine learning and AI, it has become increasingly common to adopt a conceptual model where distributions are considered objects in their own right, as opposed to primarily represented through distribution defining functions (e.g., cdf, pdf) or random samples.

An important distinction to keep in mind is between random variables (that can be sampled from) and probability distributions. **distr6** is an interface for probability distributions and supports construction, manipulation, composition, and querying of parametrised simple and composite distributions. **distr6** is not an interface for random variables and therefore procedures such as sampling and inference are out of scope.

The conceptual model: probability distributions and random variables

We continue by explaining our conceptual model of probability distributions underpinning the design of **distr6** and delineate it from the common conceptualization of random variables. A full mathematical definition of the conceptual model is given in the next section; this section contains an intuitive introduction.

First, we invite the reader to recall some common mathematical objects and recognize that these are related but conceptually distinct:

- a random variable, distributed according to a certain distribution, e.g., $X \sim \text{Normal}(0,1)$
- the cdf of that random variable X, usually denoted by F_X, a function $F_X : \mathbb{R} \to [0,1]$
- the pdf of that random variable X, often denoted by f_X, a function $f_X : \mathbb{R} \to [0,\infty)$
- the distribution according to which X is distributed - often called 'the law of' X. This can be represented by multiple mathematical objects, such as the cdf F_X or the pdf f_X. We will call this distribution d. Note that d is not identical to either these representation functions.

Critically, we highlight that random variables and distributions are neither identical objects nor concepts. A random variable X *has* distribution d, and multiple random variables may be distributed according to d. Further, random variables are sampled from, while the distribution is only a description of probabilities for X. Thus, X and d are not identical objects. Figure 1 visually summarizes these differences.

As a possible logical consequence of the above, we adopt the conceptual model that a distribution is an abstract object, which:

- Has multiple defining representations, for example through cdf and possibly through pdf, but is not identical with any of these representations
- Possesses traits, such as being absolutely continuous over the Reals, and properties, such as skewness and symmetry.
- Can be used to define sampling laws of random variables, but is not conceptually identical with a random variable.

Abstracting distributions as objects from multiple, non-identical, representations (random variables), introduces major consequences for the conceptual model:

(i) It lends itself naturally to a class-object representation, in the computer scientific sense of object-oriented programming. Abstract distributions become classes, concrete distributions are objects, and distribution defining functions are methods of these classes. Random variables are a separate type of object.

(ii) It strongly suggests adoption of mathematical conceptualization and notation which cleanly separates distributions from random variables and distribution defining functions - in contrast to common convention where random variables or random sampling takes conceptual primacy above all.

(iii) It allows clean formulation of algorithmic manipulations involving distributions, especially higher-order constructs (truncation, huberization, etc.), as well as clean mathematical definitions.
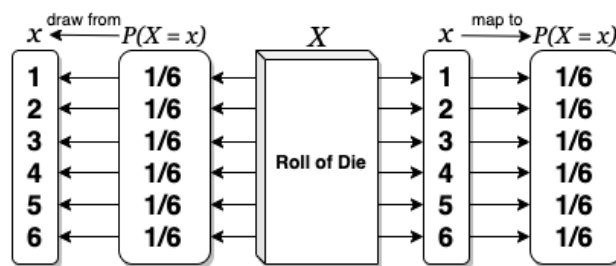
(a) Discrete Uniform Random Variable
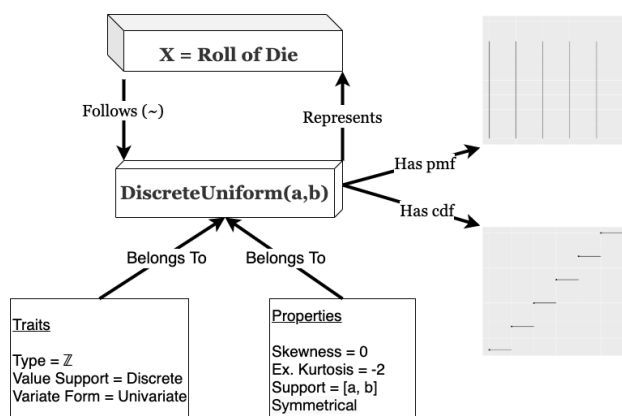
(b) Discrete Uniform Probability Distribution

Figure 1: Schematic representation of concepts "probability distribution" and "random variable". (a) Example of a random variable following a Discrete Uniform distribution. A random variable is an object in its own right, modelling the process of a random experiment. It *has* a probability distribution describing the nature of the experiment (in the case of uniform: the masses on the 6 outcomes), and can be sampled from, to obtain realizations. (b) Example of a probability distribution, the Discrete Uniform distribution. A probability distribution is an object in its own right and can serve as a template for random experiments, represented as random variables. Properties and traits such as value support, mean, variance, and functions such as pmf, cdf, are properties and traits of the probability distribution, not (direct) properties of a random variables, even if random variables can follow a given distribution.

Distributions as software objects and mathematical objects

In **distr6**, distributions are first-class objects subject to an object-oriented class-object representation. For example, a discrete uniform distribution (fig. 1b) is a 'class' with traits such as type (Naturals), and variate form (univariate). With a given parametrization, this becomes an 'object' with properties including symmetry and support. An alternative definition to the conceptual model of distributions is now provided.

On the mathematical level, we again consider distributions as objects in their own right, not being identical with a cdf, pdf, or measure, but instead 'having' these as properties.

For a set $\mathcal{Y}$ (endowed with suitable topology), we define $\mathrm{Distr}(\mathcal{Y})$ as a set containing formal objects d which are in bijection to (but not identical with) probability measures over $\mathcal{Y}$. Elements of $\mathrm{Distr}(\mathcal{Y})$ are called distributions over $\mathcal{Y}$. We further define formal symbols which, in case of existence, denote 'aspects' that such elements have, in the following way: the symbol $d.F$, for example, denotes the cdf

of d, which is to be read as the 'F' of d, with F in this case to be read as a modifier to a standard symbol d, rather than a fixed, bound, or variable symbol. In this way, we can define:

(i) $d.F$ for the cdf of d. This typically exists if $\mathcal{Y} \subseteq \mathbb{R}^n$ for some n, in which case $d.F$ is a function of type $d.F : \mathbb{R}^n \to [0, 1]$.

(ii) $d.f$ for the pdf of d. This exists if $\mathcal{Y} \subseteq \mathbb{R}^n$, and the distribution d is absolutely continuous over $\mathcal{Y}$. In this case, $d.f$ is a function of type $d.f : \mathbb{R}^n \to [0, \infty)$.

(iii) $d.P$ for the probability measure that is in bijection with d. This is a function $d.P : \mathcal{F} \to [0, 1]$, where $\mathcal{F}$ is the set of measurable sub-sets of $\mathcal{Y}$.

We would like to point out that the above is indeed a full formal mathematical definition of our notion of distribution. While distributions, defined this way, are not identical with any of the conventional mathematical objects that define them (cdf, pdf, measures), they are conceptually, formally, and notationally well-defined. Similarly, the aspects ($d.F$, $d.f$, etc) are also well-defined, since they refer to one of the conventional mathematical objects which are well-specified in dependence of the distribution (in case of existence).

This notation provides a more natural and clearer separation of distribution and random variables and allows us to talk about and denote concepts such as 'the cdf of any random variable following the distribution d' with ease ($d.F$), unlike classical notation that would see one define $X \sim d$ and then write F_X. Our notation more clearly follows the software implementation of distributions.

For example, in **distr6**, the code counterpart to defining a distribution d which is Gaussian with mean 1 and variance 2 is

```
> d <- Normal$new(1, 2)
```

The pdf and cdf of this Gaussian distribution evaluated at 2 are obtained in code as

```
> d$pdf(2)
> d$cdf(2)
```

which evaluates to 'numerics' that represent the real numbers $d.f(2)$ and $d.F(2)$.

The consideration of distributions as objects, and their conceptual distinction from random variables as objects, notably differs from conceptualization in R **stats**, which implements both distribution and random variable methods by the 'dpqr' functions. Whilst this may allow very fast generation of probabilities and values, there is no support for querying and inspection of distributions as objects. By instead treating the dpqr functions as methods that belong to a distribution object, **distr6** encapsulates all the information in R **stats** as well as distribution properties, traits, and other important mathematical methods. The object orientation principle that defines the architecture of **distr6** is further discussed throughout this manuscript.

Treating distributions as objects is not unique to this package. Possibly the first instance of the object-oriented conceptualization is the **distr** (Ruckdeschel et al., 2006) family of packages. **distr6** was designed alongside the authors of **distr** in order to port some of their functionality from S4 to R6.

distr6 is the first such package to use the 'class' object-oriented paradigm **R6** (Chang, 2018), with other distribution related packages using S3 or S4. The choice of R6 over S3 and S4 is discussed in detail in section 2.5.1. This choice allows **distr6** to fully leverage the conceptual model and make use of core R6 functionality. As well as introducing fundamental object-oriented programming (OOP) principles such as abstract classes and tried and tested design patterns (Gamma et al., 1996) including decorators, wrappers, and compositors (see section 2.5.3).

Besides an overview of **distr6**'s novel approach to probability distributions in R, this paper also presents a formal comparison of the different OOP paradigms, while detailing the use of design patterns relevant to the package.

Motivating example: Higher-order distribution constructs

The strength of the object-oriented approach, both on the algorithmic and mathematical side, lies in its ability to efficiently express higher-order constructs and operations: actions between distributions, resulting in new distributions. One such example is mixture distributions (also known as spliced distributions). In the **distr6** software interface, a MixtureDistribution is a higher-order distribution depending on two or more other distributions. For example take a uniform mixture of two distributions distr1 and distr2:

```
> my_mixt <- MixtureDistribution$new(list(distr1, distr2))
```

Internally, the dependency of the constructs on the components is remembered so that `my_mixt` is not only evaluable for `cdf` (and other methods), but also carries a symbolic representation of its construction and definition history in terms of `distr1` and `distr2`.

On the mathematical side, the object-oriented formalism allows clean definitions of otherwise more obscure concepts. For example, the mixture distribution is now defined as follows.

For distributions $d_1, \ldots, d_m$ over $\mathbb{R}^n$ and weights $w_1, \ldots, w_m$, we define the mixture of $d_1, \ldots, d_m$ with weights $w_1, \ldots, w_m$ to be the unique distribution $\tilde{d}$ such that $\tilde{d}.F(x) = \sum_{i=1}^{m} w_i \cdot d_i.F(x)$ for any $x \in \mathbb{R}^n$. Note the added clarity by defining the mixture on the distribution d_i, i.e., a first-order concept in terms of distributions.

Related software

This section provides a review of other related software that implement probability distributions. This is focused on, but not limited to, software in R.

R stats, actuar, and extraDistr   The core R programming language consists of packages for basic coding and maths as well as the **stats** package for statistical functions. **stats** contains 17 common probability distributions and four lesser-known distributions. Each distribution consists of (at most) four functions: `dX,pX,qX,rX` where X represents the distribution name. These correspond to the probability density/mass, cumulative distribution, quantile (inverse cumulative distribution), and simulation functions respectively. Each is implemented as a separate function, written in C, with both inputs and outputs as numerics. The strength of these functions lies in their speed and efficiency. There is no quicker way to find, say, the pdf of a Normal distribution than to run the `dnorm` function from **stats**. However, this is the limit of the package in terms of probability distributions. As there is no designated distribution object, there is no way to query results from the distributions outside of the 'dpqr' functions.

Several R packages implement `dpqr` functions for extra probability distributions. Of particular note are the **extraDistr** (Wolodzko, 2019) and **actuar** (Dutang et al., 2008) packages that add over 60 distributions between them. Both of these packages are limited to `dpqr` functions and therefore have the same limits as R **stats**.

distr   The **distr** package was the first package in R to implement an object-oriented interface for distributions, using the S4 object-oriented paradigm. **distr** tackles the two fundamental problems of **stats** by introducing distributions as objects that can be stored and queried. These objects include important statistical results, for example, the expectation, variance, and moment generating functions of a distribution. The **distr** family of packages includes a total of five packages for object-oriented distributions in R. **distr** has two weaknesses that were caused by using the S4 paradigm. Firstly, the package relies on inheritance, which means that large class trees exist for every object, and extensibility is therefore non-trivial. The second weakness is that S4 objects are not referred to by 'pointers' but instead copies. This means that a simple mixture of two distributions is just under 0.5Mb in size (relatively quite large).

distributions3 and distributional   The **distributions3** package (Hayes and Moller-Trane, 2019) defines distributions as objects using the S3 paradigm. However, whilst **distributions3** treats probability distributions as S3 objects, it does not add any properties, traits, or methods and instead uses the objects solely for `dpqr` dispatch. In comparison to **distr**, the **distributions3** package provides fewer features for inspection or composition. More recently, **distributional** (O'Hara-Wild and Hayes, 2020) builds on the **distributions3** framework (common authors exist between the two) to focus on the vectorization of probability distributions coded as S3 objects. Similarly to **distr6**, the primary use-case of this package is for predictive modelling of distributions as objects.

mistr   The **mistr** package (Sablica and Hornik, 2020) is another recent distributions package, which is also influenced by **distr**. The sole focus of **mistr** is to add a comprehensive and flexible framework for composite models and mixed distributions. Similarly to the previous two packages, **mistr** implements distributions as S3 objects.

Distributions.jl   Despite not being a package written in R, the Julia **Distributions.jl** (Lin et al., 2019) package provided inspiration for **distr6**. **Distributions.jl** implements distributions as objects with statistical properties including expectation, variance, moment generating and characteristic functions, and many more. This package uses multiple inheritance for 'valueSupport' (discrete/continuous) and 'variateForm' (univariate/multivariate/matrixvariate). Every distribution inherits from both of these, e.g. a distribution can be 'discrete-univariate', 'continuous-multivariate', 'continuous-matrixvariate', etc. The package provides a unified and user-friendly interface, which was a helpful starting point for **distr6**.

Design principles

distr6 was designed and built around the following principles.

D1) **Unified interface** The package is designed such that all distributions, no matter how complex, have an identical user-facing interface. This helps make the package easy to navigate and the documentation simple to read. Moreover, it minimises any confusion resulting from using multiple distributions. A clear inheritance structure also allows wrappers and decorators to have the same methods as distributions, which means even complex composite distributions should be intuitive to use. Whether a user constructs a simple Uniform distribution, or a mixture of 100 Normal distributions, the same methods and fields are seen in both objects.

D2) **Separation of core/exotic and numerical/analytic** Via abstraction and encapsulation, core statistical results (common methods such as mean and variance) are separated from 'exotic' ones (less common methods such as anti-derivatives and p-norms). Similarly, implemented distributions only contain analytic results; users can impute numerical results using decorators. This separation has several benefits, including: 1) for predictive modelling with/of distributions, numerical results can take longer to compute than analytical results, and the difference between precision of analytical and numerical results can be substantial in the context of automated modelling, separation allows these differences to be highlighted and controlled; 2) separating numerical results allows an expanded interface for users to fine-tune and set their own parameters for how numerical results are computed; 3) a less-technical user can guarantee precision of results as they are unlikely to use numerical decorators; 4) a user has access to the most important distribution methods immediately after construction but is not overwhelmed by many 'exotic' methods that they may never use. Use of decorators and wrappers allows the user to manually expand the interface at any time. For example, a user can choose between an undecorated Binomial distribution, with common methods such as mean and variance, or they can decorate the distribution to additionally gain access to survival and hazard functions.

D3) **Inheritance without over-inheritance** The class structure stems from a series of a few abstract classes with concrete child classes, which allows for a sensible, but not over-complicated, inheritance structure. For example, all implemented distributions inherit from a single parent class, so common methods can be unified and only coded once; note there is no separation of distributions into 'continuous' and 'discrete' classes. By allowing extension of classes by decorators and wrappers, and not solely inheritance, the interface is highly scalable and extensible. By 'scalability', we refer to the interface's ability to grow to a large scale without additional overheads, the decorator and wrapper patterns on top of the R6 paradigm allow an (theoretically) unlimited number of distributions, wrappers, and methods without computational difficulty. By 'extensibility', we refer to the ability to extend the interface, again this is made possible by clean abstraction of distributions, wrappers, core methods, and extra methods in decorators. All decorators and wrappers in **distr6** stem from abstract classes, which in turn inherit from the `Distribution` super-class. In doing so, any method of expanding an object's interface in **distr6** (i.e. via decorators, wrappers, or inheritance) will automatically lead to an interface that inherits from the top-level class, maintaining the principle of a unified interface (D1).

D4) **Inspection and manipulation of multiple parameterisations** The design process identified that use of distributions in R **stats** is inflexible in that in the majority of cases, only one parameterisation of each distribution is allowed. This can lead to isolating users who may be very familiar with one parameterisation but completely unaware of another. For example, the use of the *precision* parameter in the Normal distribution is typically more common in Bayesian statistics, whereas using the *variance* or *standard deviation* parameters is more common in frequentist statistics. **distr6** allows the user to choose from multiple parameterisations for all distributions (where more than one parameterisation is possible/known). Furthermore, querying and updating of any parameter in the distribution is allowed, even if it was not specified in construction (section 2.4). This allows for a flexible parameter interface that can be fully queried and modified at any time.

D5) **Flexible interfacing for technical and non-technical users** Throughout the design process, it was required that **distr6** be accessible to all R users. This was a challenge as R6 is a very different paradigm from S3 and S4. To reduce the learning curve, the interface is designed to be as user-friendly and flexible as possible. This includes: 1) a 'sensible default principle' such that all distributions have justified default values; 2) an 'inspection principle' with functions to list all distributions, wrappers, and decorators. As discussed in (D2), abstraction and encapsulation allow technical users to expand any distribution's interface to be as arbitrarily complex as they like, whilst maintaining a minimal representation by default. Where possible defaults are 'standard' distributions, i.e. with location 0 and scale 1, otherwise sensible defaults are identified as realistic scenarios, for example `Binomial(n = 10,p = 0.5)`.

D6) **Flexible OO paradigms** Following from (D5), we identified that R6 is still relatively new in R with only 314 out of $16,050$ packages depending on it (as of July 2020). Therefore this was acknowledged and taken into account when building the package. R6 is also the first paradigm in R with the dollar-sign notation (though S4 uses '@' notation) and with a proper construction method. Whilst new users are advised to learn the basics of R6, S3 compatibility is available for all common methods via **R62S3** (Sonabend, 2019). Users can therefore decide on calling a method via dollar-sign notation or dispatch, the example below demonstrates 'piping' and S3. As the core package is built on R6, the thin-wrappers provided by **R62S3** do not compromise the above design principles.

```
> library(magrittr)
> N <- Normal$new(mean = 2)
> N %>%
+    setParameterValue(mean = 1) %>%
+    getParameterValue("mean")
[1] 1
> pdf(N, 1:4)
[1] 0.398942280 0.241970725 0.053990967 0.004431848
```

Overview to functionality and API

distr6 1.4.3 implements 56 probability distributions, including 11 probability kernels. Individual distributions are modelled via classes that inherit from a common interface, implemented in the abstract `Distribution` parent class. The `Distribution` class specifies the abstract distribution interface for parameter access, properties, traits, and methods, such as a distribution's pdf or cdf. The most important interface points are described in Section 2.4.1

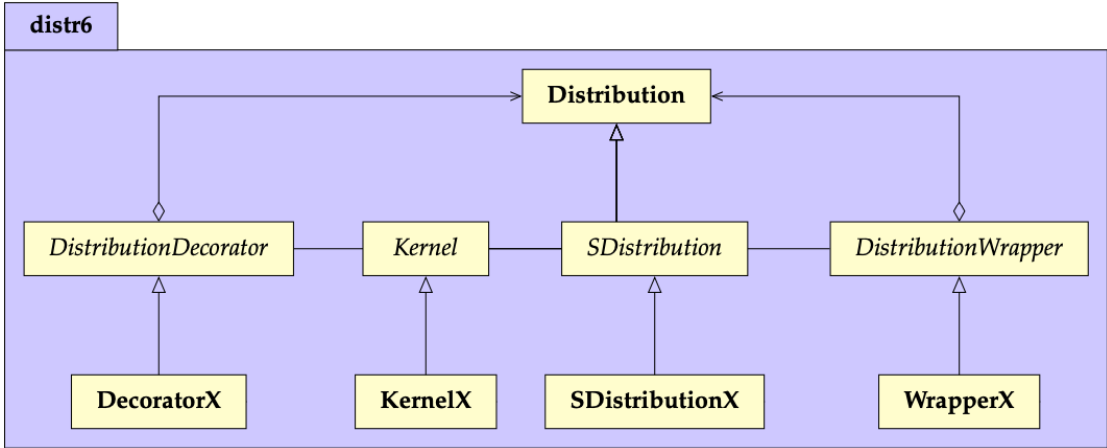| **Distribution** |
|---|
| + name : string |
| + short_name : string |
| + description : string |
| - .properties : list |
| - .traits : list |
| + public_accessors() |
| + dpqr methods() |
| + generic_math_stat_methods() |
| + ParameterSet_getters_setters() |
| + validation_methods() |

Give caption to this figure.

Concrete distributions, kernels, and wrappers are the grandchildren of `Distribution`, and children of one of the mid-layer abstract classes:

- `SDistribution`, which models abstract, generic distributions. Concrete distributions, such as `Normal`, which models the normal distribution, inherit from `SDistribution`.
- `Kernel`, which models probability kernels, such as `Triangular` and `Epanechnikov`. Probability kernels are absolutely continuous distributions over the Reals, with assumed mean 0 and variance 1.
- `DistributionWrapper`, which is an abstract parent for higher-order operations on distributions, including compositions, that is, operations that create distributions from other distributions, such as truncation or mixture.

Method	Description/Definition
pdf/cdf/quantile/rand	dpqr functions.
mean	$d.\mu = \mathbb{E}[X]$
variance	$d.\sigma^2 = \mathbb{E}[(X - d.\mu)^2]$
traits	List including value support (discrete/continuous/mixed); variate form (uni-/multi-/matrixvariate); type (mathematical domain).
properties	List including skewness ($\mathbb{E}((X - d.\mu)/d.\sigma)^3]$) and symmetry (boolean).
get/setParameterValue	Getters and setters for parameter values.
parameters	Returns the internal parameterisation set.
print/summary	Representation functions, summary includes distribution properties and traits.

Table 1: Common methods available to all classes inheriting from `Distribution`. Columns are method names and either a mathematical definition or description of a method. For mean and variance, $d.\mu$, $d.\sigma$, and $d.\sigma^2$ represent the methods mean, standard deviation, and variance associated with distribution d. The symbol X denotes a random variable with distribution d. Horizontal lines separate mathematical, property, parameter, and representation methods.

- `DistributionDecorator`, whose purpose is supplementing methods to distributions in the form of a decorator design pattern. This includes methods such as integrated cdf or squared integrals of distribution defining functions.

Give caption to this figure.

The UML diagram above visualises the key class structure of **distr6**, including the concrete `Distribution` parent class, from which all other classes in the package inherit from (with the exception of the `ParameterSet`). These abstract classes allow simple extensibility for concrete sub-classes.

The Distribution interface

The base, or top-level, class in **distr6** is the `Distribution` class. Its primary function is to act as a parent class for the implemented probability distributions and higher-order compositions. It is also utilised for the creation of custom distributions. By design, any distribution already implemented in **distr6** will have the the same interface as a user-specified custom distribution, ensuring (D1) is upheld. A table of the most important methods for a distribution are in Table 1 alongside their meaning and definitions (mathematical if possible). The two use-cases for the `Distribution` class are discussed separately.

Distribution for inheritance It is anticipated that the majority of **distr6** users will be using the package for the implemented distributions and kernels. With this in mind, the `Distribution` class defines all variables and methods common to all child classes. The most important of these are the common analytical expressions and the dpqr public methods. Every concrete implemented distribution/kernel has identical public dpqr methods that internally call private dpqr methods. This accounts for inconsistencies occurring from packages returning functions in different formats and

handling errors differently, a problem most prominent in multivariate distributions. Another example is handling of non-integer values for discrete distributions. In some packages this returns 0, in others the value is rounded down, and in others an error is returned. The dpqr functions for all distributions have unified validation checks and return types (numeric or data.table). In line with base R and other distribution packages, **distr6** implements a single pdf function to cover both probability mass and probability density functions.

```
> Normal$new()$pdf(1:2)
[1] 0.24197072 0.05399097
> Binomial$new()$cdf(1:2, lower.tail = FALSE, log.p = TRUE, simplify = FALSE)
         Binom
1: -0.01080030
2: -0.05623972
```

A key design principle in the package is separation of analytical and numerical results (D2), which is ensured by only including analytical results in implemented distributions. Missing methods in a distribution, therefore, signify that no closed-form expression for the method is available, however, all can be numerically estimated with the CoreStatistics decorator (see section 2.4.2). Ideally, all distributions will include analytical methods for the following: probability density/mass function (pdf), cumulative distribution function (cdf), inverse cumulative distribution function/quantile function (quantile), simulation function (rand), mean, variance, skewness, (excess) kurtosis, and entropy of the distribution (mean, variance, skewness, kurtosis, entropy), as well as the moment generating function (mgf), characteristic function (cf), and probability generating function (pgf). Speed is currently a limitation in **distr6**, but the use of **Rcpp** (Eddelbuettel and Francois, 2011) in all dpqr functions helps mitigate against this.

The fourth design principle of **distr6** ensures that multiple parameterisations of a given distribution can be both provided and inspected at all times. For example, the Normal distribution can be parametrised in terms of variance, standard deviation, or precision. Any of which can be called in construction with other parameters updated accordingly. If conflicting parametrisations are provided, then an error is returned. By example,

```
# set precision, others updated automatically
> Normal$new(prec = 4)
Norm(mean = 0, var = 0.25, sd = 0.5, prec = 4)
# try and set both precision and variance, results in error
> Normal$new(var = 1, prec = 2)
Error in FUN(X[[i]], ...) :
  Conflicting parametrisations detected. Only one of {var, sd, prec} should be given.
```

The same principle is used for parameter setting. The methods getParameterValue and setParameterValue are utilised for getting and setting parameter values, respectively. The former takes a single argument, the parameter name, and the second a named list of arguments corresponding to the parameter name and the value to set. The example below demonstrates this for a Gamma distribution. Here, the distribution is constructed, the shape parameter is queried, both shape and rate parameters are updated, and the latter queried, finally, the scale parameter is set, which auto-updates the rate parameter.

```
> G <- Gamma$new(shape = 1, rate = 1)
> G$getParameterValue("shape")
[1] 1
> G$setParameterValue(shape = 1, rate = 2)
> G$getParameterValue("rate")
[1] 2
> G$setParameterValue(scale = 2)
> G$getParameterValue("rate")
[1] 0.5
```

Distribution and parameter domains and types are represented by mathematical sets, implemented in **set6** (Sonabend and Kiraly, 2020). This allows for clear representation of infinite sets and, most importantly, for internal containedness checks. For example, all public dpqr methods first call the $contains method in their respective type and return an error if any points are outside the distribution's domain. As **set6** uses **Rcpp** for this method, these come at minimal cost to speed.

```
> B <- Binomial$new()
```

```
> B$pdf(-1)
Error in B$pdf(-1) :
  Not all points in {-1} lie in the distribution domain (N0).
```

These domains and types are returned along with other important properties and traits in a call to properties and traits, respectively. This is demonstrated below for the Arcsine distribution.

```
> A <- Arcsine$new()
> A$properties
$support
[0,1]


$symmetry
[1] "symmetric"

> A$traits
$valueSupport
[1] "continuous"

$variateForm
[1] "univariate"

$type
R
```

Extending distr6 with custom distributions Users of **distr6** can create temporary custom distributions using the constructor of the `Distribution` class directly. Permanent extensions, e.g., as part of an R package, should create a new concrete distribution as a child of the `SDistribution` class.

The `Distribution` constructor is given by

```
Distribution$new(name = NULL, short_name = NULL, type = NULL, support = NULL,
+ symmetric = FALSE, pdf = NULL, cdf = NULL, quantile = NULL, rand = NULL,
+ parameters = NULL, decorators = NULL, valueSupport = NULL, variateForm = NULL,
+ description = NULL)
```

The `name` and `short_name` arguments are identification for the custom distribution used for printing. `type` is a trait corresponding to scientific type (e.g., Reals, Integers,...) and `support` is the property of the distribution support. Distribution parameters are passed as a `ParameterSet` object. This defines each parameter in the distribution including the parameter default value and support. The `pdf/cdf/quantile/rand` arguments define the corresponding methods and are passed to the private `.pdf/.cdf/.quantile/.rand` methods. As above, the public methods are already defined and ensure consistency in each function. At a minimum, users have to supply the distribution name, type, and either `pdf` or `cdf`. All other information can be numerically estimated with decorators (see section 2.4.2).

```
> d <- Distribution$new(name = "Custom Distribution", type = Integers$new(),
+                       support = Set$new(1:10),
+                       pdf = function(x) rep(1/10, length(x)))
> d$pdf(1:3)
[1] 0.1 0.1 0.1
```

DistributionDecorator

Decorators add functionality to classes in object-oriented programming. These are not natively implemented in R6, and this novel implementation is therefore discussed further in section 2.5.3. Decorators in **distr6** are only 'allowed' if they have at least three methods and cover a clear use-case. This prevents too many decorators from bloating the interface. However, by their nature, they are lightweight classes that will only increase the methods in a distribution if explicitly requested by a user. Decorators can be applied to a distribution in one of three ways.

In construction:

```
> N <- Normal$new(decorators = c("CoreStatistics", "ExoticStatistics"))
```

Using the `decorate()` function:

```
> N <- Normal$new()
> decorate(N, c("CoreStatistics", "ExoticStatistics"))
```

Using the `$decorate` method inherited from the `DistributionDecorator` super-class:

```
> N <- Normal$new()
> ExoticStatistics$new()$decorate(N)
```

The first option is the quickest if decorators are required immediately. The second is the most efficient once a distribution is already constructed. The third is the closest method to true OOP but does not allow adding multiple decorators simultaneously.

Three decorators are currently implemented in **distr6**, these are briefly described.

CoreStatistics   This decorator imputes numerical functions for common statistical results that could be considered core to a distribution, e.g. the mean or variance. The decorator additionally adds generalised expectation (`genExp`) and moments (`kthmoment`) functions, which allow numerical results for functions of the form $\mathbb{E}[f(X)]$ and for crude/raw/central K moments. The example below demonstrates how the `decorate` function exposes methods from the `CoreStatistics` decorator to the Normal distribution object.

```
> n <- Normal$new(mean = 2, var = 4)
> n$kthmoment(3, type = "raw")
Error: attempt to apply non-function
> decorate(n, CoreStatistics)
> n$kthmoment(3, type = "raw")
[1] 32
```

ExoticStatistics   This decorator adds more 'exotic' methods to distributions, i.e. those that are unlikely to be called by the majority of users. For example this includes methods for the p-norm of survival and cdf functions, as well as anti-derivatives for these functions. Where possible, analytic results are exploited. For example, this decorator can implement the survival function in one of two ways: either as i) 1 minus the distribution cdf, if an analytic expression for the cdf is available, or ii) via numerical integration of the distribution.

FunctionImputation   This decorator imputes numerical expressions for the `dpqr` methods. This is the most useful for custom distributions in which only the `pdf` or `cdf` is provided. Numerical imputation is implemented via **Rcpp**.

Composite distributions

Composite distributions - that is, distributions created from other distributions - are common in advanced usage. Examples for composites are truncation, mixture, or transformation of domain. In **distr6**, a number of such composites are supported. Implementation-wise, this uses the wrapper OOP pattern, which is not native to R6 but part of our extensions to R6, discussed in section 2.5.3.

As discussed above, wrapped distributions inherit from `Distribution` thus have an identical interface to any child of `SDistribution`, with the following minor differences:

- The `wrappedModels` method provides a unified interface to access any component distribution.
- Parameters are still accessed via the same method but stored in a `ParameterSetCollection` object instead of a `ParameterSet`, thus allowing efficient representation of composite and nested parameter sets.

The Composition can be iterated and nested any number of times, consider the following example where a mixture distribution is created from two distributions that are in turn composites - a truncated Student T, and a Huberized exponential - note the parameter inspection and automatic prefixing of distribution 'short names' to the parameters for identification.

```
> M <- MixtureDistribution$new(list(
+   truncate(StudentT$new(), lower = -1, upper = 1),
+   huberize(Exponential$new(), upper = 4)
+ ))
> M$parameters()
```

```
                id    value       support                 description
1:       mix_T_df        1             R+        Degrees of Freedom
2: mix_trunc_lower       -1   R U {-Inf, +Inf}   Lower limit of truncation
3: mix_trunc_upper        1    R U {-Inf +Inf}   Upper limit of truncation
4:    mix_Exp_rate        1             R+               Arrival Rate
5:   mix_Exp_scale        1             R+                      Scale
6:   mix_hub_lower        0   R U {-Inf, +Inf}   Lower limit of huberization
7:   mix_hub_upper        4   R U {-Inf, +Inf}   Upper limit of huberization
8:    mix_weights  uniform   {uniform} U [0,1]            Mixture weights
```

Implemented compositors We summarize some important implemented compositors (tables 2 and 3), to illustrate the way composition is handled and implemented.

Class	Parameters	Type of d	Components of d
TruncatedDistribution	$a, b \in \mathbb{R}$	$\mathbb{R}$	d', type $\mathbb{R}$
HuberizedDistribution	$a, b \in \mathbb{R}$	$\mathbb{R}$, mixed	d', type $\mathbb{R}$
MixtureDistribution	$w_i \in \mathbb{R}, \sum_{i=1}^{n} w_i = 1$	$\mathbb{R}^n$	d'_i, type $\mathbb{R}^n$
ProductDistribution	-	$\mathbb{R}^N, N = \sum_{i=1}^{n} n_i$	d'_i type $\mathbb{R}^{n_i}$

Table 2: Examples of common compositors implemented in **distr6** - parameters and type. Column 2 (parameters) lists the parameters that the composite has. Column 3 (type) states the type of the resultant distribution d that is created when the class is constructed; this states the formal domain. Column 4 (components) states the number, names, and assumptions on the components, if any.

Class	$d.F(x)$	$d.f(x)$
TruncatedDistribution	$\frac{d'.F(x) - d'.F(a)}{d'.F(b) - d'.F(a)}$	$\frac{d'.f(x)}{d'.P([a,b])}$
HuberizedDistribution	$d'.F(x) + \mathbb{1}[x = b] \cdot d'.P(b)$	(no pdf, since mixed)
MixtureDistribution	$\sum_{i=1}^{N} w_i \cdot d'_i.F(x)$	$\sum_{i=1}^{N} w_i \cdot d'_i.f(x)$ (if exists)
ProductDistribution	$\prod_{i=1}^{N} d'_i.F(x)$	$\prod_{i=1}^{N} d'_i.f(x)$ (if exists)

Table 3: Common compositors implemented in **distr6** with mathematical definitions. Column 2 defines the resultant cdf, $d.F$, in terms of the component cdf, as implemented in the cdf method of the compositor in the same row. Column 3 defines the resultant pdf, $d.f$, in terms of the component pdf, as implemented in the pdf method of the compositor in the same row. Truncation (row 1) is currently implemented for the left-open interval $(a, b]$ only. For Huberization (row 2), the resultant distribution is in general not absolutely continuous and hence the pdf does not exist. d' resp. d'_i are the component distributions, as defined in table 2.

Example code to obtain a truncated or huberized distribution is below. Here, we construct a truncated normal with truncation parameters -1 and 1, and a huberized Binomial with bounding parameters 2 and 5.

```
> TN <- truncate(Normal$new(), lower = -1, upper = 1)
> TN$cdf(-2:2)
[1] 0.0 0.0 0.5 1.0 1.0
> class(TN)
[1] "TruncatedDistribution" "DistributionWrapper" "Distribution" "R6"

> HB <- huberize(Binomial$new(), lower = 2, upper = 5)
> HB$cdf(1:6)
[1] 0.0000000 0.0546875 0.1718750 0.3769531 1.0000000 1.0000000
> HB$median()
[1] 5
```

Vectorization of distributions A special feature of **distr6** is that it allows vectorization of distributions, i.e. vectorised representation of multiple distributions in an array-like structure. This is primarily done for computational efficiency with general best R practice of vectorisation. Vectorisation of **distr6** distributions is implemented via the `VectorDistribution` which is logically treated as a compositor.

Mathematically, a `VectorDistribution` is simply a vector of component distributions $d_1, \ldots, d_N$ that allows vectorised evaluation. Two kinds of vectorised evaluation are supported - paired and product-wise vectorization, which we illustrate in the case of cdfs.

- Paired vectorised evaluation of the cdfs $d_1.F, \ldots, d_N.F$ at numbers $x_1, \ldots, x_N$, yields a real vector $(d_1.F(x_1), \ldots, d_N.F(x_N))$ via the cdf method.
- Product vectorised evaluation of the cdfs $d_1.F, \ldots, d_N.F$ at numbers $x_1, \ldots, x_M$, yields a real $(N \times M)$ matrix, with (i, j)-th entry $d_i.F(x_j)$.

In practical terms, paired evaluation is the evaluation of N distributions at N points (which may be unique or different). So by example for three distributions d_1, d_2, d_3, paired evaluation of their cdfs at $(x_1, x_2, x_3) = (4, 5, 6)$ respectively results in $(d_1.F(x_1), d_2.F(x_2), d_3.F(x_3)) = (d_1.F(4), d_2.F(5), d_3.F(6))$. In **distr6**:

```
> V <- VectorDistribution$new(distribution = "Normal", params = data.frame(mean = 1:3))
> V$cdf(4, 5, 6)
        Norm1     Norm2     Norm3
1: 0.9986501 0.9986501 0.9986501
```

In contrast, product-wise evaluation evaluates N distributions at the same M points. Product-wise evaluation of the cdfs of d_1, d_2, d_3 at $(x_1, x_2, x_3) = (4, 5, 6)$ results in

$$\begin{pmatrix} d_1.F(x_1) & d_1.F(x_2) & d_1.F(x_3) \\ d_2.F(x_1) & d_2.F(x_2) & d_2.F(x_3) \\ d_3.F(x_1) & d_3.F(x_2) & d_3.F(x_3) \end{pmatrix} = \begin{pmatrix} d_1.F(4) & d_1.F(5) & d_1.F(6) \\ d_2.F(4) & d_2.F(5) & d_2.F(6) \\ d_3.F(4) & d_3.F(5) & d_3.F(6) \end{pmatrix}$$

In **distr6**:

```
> V <- VectorDistribution$new(distribution = "Normal", params = data.frame(mean = 1:3))
> V$cdf(4:6)
        Norm1     Norm2     Norm3
1: 0.9986501 0.9772499 0.8413447
2: 0.9999683 0.9986501 0.9772499
3: 0.9999997 0.9999683 0.9986501
```

The `VectorDistribution` wrapper allows for efficient vectorisation across *both* the distributions *and* points to evaluate, which we believe is a feature unique to **distr6** among distribution frameworks in R. By combing product and paired modes, users can evaluate any distribution in the vector at any point. In the following example, Normal(1, 1) is evaluated at (1,2) and Normal(2, 1) is evaluated at (3,4):

```
> V <- VectorDistribution$new(distribution = "Normal", params = data.frame(mean = 1:2))
> V$pdf(1:2, 3:4)
        Norm1      Norm2
1: 0.3989423 0.24197072
2: 0.2419707 0.05399097
```

Further, common composites such as `ProductDistribution` and `MixtureDistribution` inherit from `VectorDistribution`, allowing for efficient vector dispatch of pdf and cdf methods. Inheriting from `VectorDistribution` results in identical constructors and methods. Thus a minor caveat is that users could evaluate a product or mixture at different points for each distribution, which is not a usual use case in practice.

Two different choices of constructors are provided, the first 'distlist' constructor passes distribution *objects* into the constructor, whereas the second passes a reference to the distribution *class* along with the parameterisations. Therefore the first allows different types of distributions but is vastly slowly as the various methods have to be calculated individually, whereas the second only allows a single class of distribution at a time, but is much quicker in evaluation. In the example below, the mixture uses the second constructor and the product uses the first.

```
> M <- MixtureDistribution$new(distribution = "Degenerate",
```

```
+                                params = data.frame(mean = 1:10))
> M$cdf(1:5)
[1] 0.1 0.2 0.3 0.4 0.5
> class(M)
[1] "MixtureDistribution" "VectorDistribution"  "DistributionWrapper" "Distribution"
[5] "R6"

> P <- ProductDistribution$new(list(Normal$new(), Exponential$new(), Gamma$new()))
> P$cdf(1:5)
[1] 0.3361815 0.7306360 0.9016858 0.9636737 0.9865692
```

Design patterns and object-oriented programming

This paper has so far discussed the API and functionality in **distr6**. This section discusses object-oriented programming (OOP), firstly a brief introduction to OOP and OOP in R and then the package's contributions to the field.

S3, S4, and R6

R has four major paradigms for object-oriented programming: S3, S4, reference classes (R5), and most recently, R6. S3 and S4 are known as functional object-oriented programming (FOOP) paradigms whereas R5 and R6 move towards class object-oriented programming (COOP) paradigms (R6) (Chambers, 2014). One of the main differences (from a user perspective) is that methods in COOP are associated with a class whereas in FOOP, methods are associated with generic functions. In the first case methods are called by first specifying the object and in the second, a dispatch registry is utilised to find the correct method to associate with a given object.

S3 introduces objects as named structures, which in other languages are often referred to as 'typed lists'. These can hold objects of any type and can include meta-information about the object itself. S3 is the dominant paradigm in R for its flexibility, speed, and efficiency. As such it is embedded deep in the infrastructure of R and single dispatch is behind a vast majority of the base functionality, which is a key part of making R easily readable. S3 is a FOOP paradigm in which functions are part of a dispatch system and consist of a generic function that is external to any object and a specific method registered to a 'class'. However the term 'class' is slightly misleading as no formal class structure exists (and by consequence no formal construction or inheritance) and as such S3 is not a formal OOP language[1].

S4 formalises S3 by introducing: class-object separation, a clear notion of construction, and multiple inheritance (Chambers, 2014). S4 has more syntax for the user to learn and a few more steps in class and method definitions. As a result S4 syntax is not overly user-friendly and S3 is used vastly more than S4 (Chambers, 2014).

There is a big jump from S3 and S4 to R6 as they transition from functional- to class-object-oriented programming. This means new notation, semantics, syntax, and conventions. The key changes are: 1) introducing methods and fields that are associated with classes not functions; 2) mutable objects with copy-on-modify semantics; and 3) new dollar-sign notation. In the first case this means that when a class is defined, all the methods are defined as existing within the class, and these can be accessed at any time after construction. Methods are further split into *public* and *private*, as well as *active bindings* which incorporates the abstraction part of OOP. The mutability of objects and change to copy-on-modify means that to create an independent copy of an object, the new method `clone(deep = TRUE)` has to be used, which would be familiar to users who know more classical OOP but very different to most R users. Finally methods are accessed via the dollar-sign, and not by calling a function on an object.

Below the three paradigms are contrasted semantically with a toy example to create a 'duck' class with a method 'quack':

S3

```
> quack <- function(x) UseMethod("quack", x)
> duck <- function(name) return(structure(list(name = name), class = "duck"))
> quack.duck <- function(x) cat(x$name, "QUACK!")
> quack(duck("Arthur"))
Arthur QUACK!
```

---

[1]http://adv-r.had.co.nz/OO-essentials.html

S4

```
> setClass("duck", slots = c(name = "character"))
> setGeneric("quack", function(x) {
+   standardGeneric("quack")
+ })
> setGeneric("duck", function(name) {
+    standardGeneric("duck")
+ })
> setMethod("duck", signature(name = "character"),
+            definition = function(name){
+              new("duck", name = name)
+            })
> setMethod("quack",
+            definition = function(x) {
+            cat(x@name, "QUACK!")
+            })
> quack(duck("Ford"))
Ford QUACK!
```

R6

```
> duck <- R6::R6Class("duck", public = list(
+   initialize = function(name) private$.name = name,
+   quack = function() cat(private$.name, "QUACK!")),
+   private = list(.name = character(0)))
> duck$new("Zaphod")$quack()
Zaphod QUACK!
```

The example clearly highlights the extra code introduced by S4 and the difference between the S3 dispatch and R6 method system.

Comparing the paradigms   There is no doubt that R6 is the furthest paradigm from conventional R usage, and as such, there is a steep learning curve for the majority of R users. However, R6 will be most natural for users coming to R from more traditional OOP languages. In contrast, S3 is a natural FOOP paradigm that will be familiar to all R users (even if they are not aware that S3 is being used). S4 is an unfortunate midpoint between the two, which, whilst being very useful, is not particularly user-friendly in terms of programming classes and objects.

distr was developed soon after S4 was released and is arguably one of the best case studies for how well S4 performs. Whilst S4 formalizes S3 to allow for a fully OO interface to be developed, its dependence on inheritance forces design decisions that quickly become problematic. This is seen in the large inheritance trees in **distr** in which one implemented distribution can be nested five child classes deep. This is compounded by the fact that S4 does not use pointer objects but instead nests objects internally. Therefore, **distr** has problems with composite distributions in that they quickly become very large in size. For example, a mixture of two distributions can easily be around 0.5Mb, which is relatively large. In contrast, R6 introduces pointers, which means that a wrapped object simply points to its wrapped component and does not copy it needlessly.

Whilst a fully object-oriented interface can be developed in S3 and S4, they do not have the flexibility of R6, which means that in the long run, extensibility and scalability can be problematic. R6 forces R users to learn a paradigm that they may not be familiar with, but packages like **R62S3** allow users to become acquainted with R6 on a slightly shallower learning curve. Speed differences for the three paradigms are formally compared on the example above using **microbenchmark** (Mersmann, 2019); the results are in table 4. The R6 example is compared both including construction of the class, duck$new("Zaphod")$quack(), and without construction, d$quack(), where d is the object constructed before comparison. A significant 'bottleneck' is noted when construction is included in the comparison. However, despite this, S4 is still significantly the slowest.

Design patterns

In the simplest definition, 'design patterns' are abstract solutions to common coding problems. They are probably most widely known due to the book 'Design Patterns Elements of Reusable Object-Oriented Software' (*Design Patterns*) (Gamma et al., 1996). **distr6** primarily makes use of the following design patterns

Paradigm	mean (μs)	cld
S3	73.44	a
S4	276.17	c
R6	187.70	b
R6*	38.32	a

Table 4: Comparing S3, S4, and R6 in calling a method. R6 is tested both including object construction (R6) and without (R6*). 'cld' is the significance testing from **microbenchmark** where 'a' is the fastest and 'c' the slowest. Experiment conducted with **R6** version 2.4.1, **microbenchmark** version 1.4.7, and R version 4.0.2 (2020-06-22) on platform: x86_64-apple-darwin17.0 (64-bit) running under: macOS Catalina 10.15.3.

- Abstract Factory
- Decorator
- Composite
- Strategy

Strategy   The strategy pattern is common in modelling toolboxes, in which multiple algorithms can be used to solve a problem. This pattern defines an abstract class for a given problem and concrete classes that each implement different strategies, or algorithms, to solve the problem. For example, in the context of mathematical integration (a common problem in R), one could use Simpson's rule, Kronrod's, or many others. These can be specified by an integrate abstract class with concrete sub-classes simpson and kronrod.

Give caption to this figure.
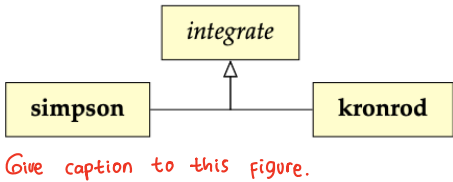
Composite   The composite pattern defines a collection of classes that have [with] an identical interface when treated independently or when composed into a single class with constituent parts. To the user, this means that only one interface needs to be learnt [learned] in order to interact with composite or individual classes. A well-built composite pattern allows users to construct complex classes with several layers of composition, and yet still be able to make use of a single interface. By inheriting from a parent class, each class and composite share a common interface. Composition is a powerful design principle that allows both modification of existing classes and reduction of multiple classes (Király et al., 2021).

Decorator   Decorators add additional responsibilities to an object without making any other changes to the interface. An object that has been decorated will be identical to its un-decorated counter-part except with additional methods. This provides a useful alternative to inheritance. Whereas inheritance can lead to large tree structures in which each sub-class inherits from the previous and contains all previous methods, decorators allow the user to pick and choose with responsibilities to add. The figure below demonstrates how this is useful in a shopping cart example. The top of the figure demonstrates using inheritance, in which each sub-class adds methods to the Cart parent class. By the Tax child class, there are a total of five methods in the interface. At the bottom of the figure, the decorator pattern demonstrates how the functionality for adding items and tax is separated and can be added separately.
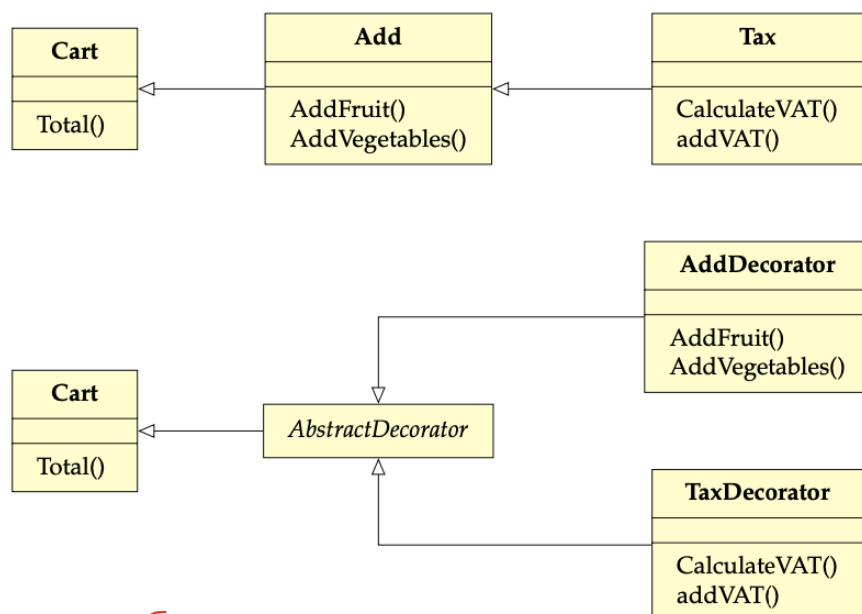
Give caption to this figure.

Contributions to R6

In order to implement **distr6**, several contributions were made to the R6 paradigm, to extend its abilities and to implement the design patterns discussed above.

Abstract classes R6 did not have a concept of abstract classes, which meant that patterns such as adapters, composites, and decorators, could not be directly implemented without problems. This is produced in **distr6** with the abstract function, which is placed in the first line of all abstract classes. In the example below, obj expects the self argument from R6 classes, and class is the name of the class, getR6Class is a custom function for returning the name of the class of the given object.

```
abstract <- function(obj, class) {
  if (getR6Class(obj) == class) {
      stop(sprintf("%s is an abstract class that can't be initialized.", class))
    }
}
```

For example in decorators the following line is placed at the top of the initialize function:

```
abstract(self, "DistributionDecorator")
```

Decorators The typical implementation of decorators is to have an abstract decorator class with concrete decorators inheriting from this, each with their own added responsibilities. In **distr6** this is made possible by defining the DistributionDecorator abstract class (see above) with a public decorate method. Concrete decorators are simply R6 classes where the public methods are the ones to 'copy' to the decorated object.

```
> DistributionDecorator
<DistributionDecorator> object generator
  Public:
    packages: NULL
    initialize: function ()
    decorate: function (distribution, ...)
    clone: function (deep = FALSE)

> CoreStatistics
<CoreStatistics> object generator
  Inherits from: <DistributionDecorator>
  Public:
    mgf: function (t)
```

```
    cf: function (t)
    pgf: function (z)
```

When the `$decorate` method from a constructed decorator object is called, the methods are simply copied from the decorator environment to the object environment. The `decorator()` function simplifies this for the user.

Composite and wrappers   The composite pattern is made use of in what **distr6** calls 'wrappers'. Again, this is implemented via an abstract class (`DistributionWrapper`) with concrete sub-classes.

```
> DistributionWrapper
<DistributionWrapper> object generator
  Inherits from: <Distribution>
  Public:
    initialize: function (distlist = NULL, name, short_name, description, support,
    wrappedModels: function (model = NULL)
    setParameterValue: function (..., lst = NULL, error = "warn")
  Private:
    .wrappedModels: list

> TruncatedDistribution
<TruncatedDistribution> object generator
  Inherits from: <DistributionWrapper>
  Public:
    initialize: function (distribution, lower = NULL, upper = NULL)
    setParameterValue: function (..., lst = NULL, error = "warn")
  Private:
    .pdf: function (x, log = FALSE)
    .cdf: function (x, lower.tail = TRUE, log.p = FALSE)
    .quantile: function (p, lower.tail = TRUE, log.p = FALSE)
    .rand: function (n)
```

Wrappers in **distr6** alter objects by modifying either their public or private methods. Therefore an 'unwrapped' distribution looks identical to a 'wrapped' one, despite inheriting from different classes. This is possible via two key implementation strategies: 1) on construction of a wrapper, parameters are prefixed with a unique ID, meaning that all parameters can be accessed at any time; 2) the `wrappedModels` public field allows access to the original wrapped distributions. These two factors allow any new method to be called either by reference to `wrappedModels` or by using `$getParameterValue` with the newly prefixed parameter ID. This is demonstrated in the `.pdf` private method of the `TruncatedDistribution` wrapper (slightly abridged).

```
.pdf = function(x, log = FALSE) {
      dist <- self$wrappedModels()[[1]]
      lower <- self$getParameterValue("trunc_lower")
      upper <- self$getParameterValue("trunc_upper")

        pdf <- numeric(length(x))
        pdf[x > lower & x <= upper] <- dist$pdf(x[x > lower & x <= upper]) /
          (dist$cdf(upper) - dist$cdf(lower))

      return(pdf)
    }
```

As the public `pdf` is the same for all distributions, and this is inherited by wrappers, only the private `.pdf` method needs to be altered.

Examples

This final section looks at concrete short examples for four key use cases.

Constructing and querying distributions

The primary use case for the majority of users will be in constructing distributions in order to query their results and visualise their shape.

Below a distribution (Binomial) is constructed and queried for its distribution-specific traits and parameterisation-specific properties.

```
> b <- Binomial$new(prob = 0.1, size = 5)
> b$setParameterValue(size = 6)
> b$getParameterValue("size")
> b$parameters()
> b$properties
> b$traits
```

Specific methods from the distribution are queried as well.

```
> b$mean()
> b$entropy()
> b$skewness()
> b$kurtosis()
> b$cdf(1:5)
```

The distribution is visualised by plotting it's density, distribution, inverse distribution, hazard, cumulative hazard, and survival function; the output is in figure 2.

```
> plot(b, fun = "all")
```

Analysis of empirical data

distr6 can also serve as a toolbox for analysis of empirical data by making use of the three 'empirical' distributions: `Empirical`, `EmpricalMV`, and `WeightedDiscrete`.

First an empirical distribution is constructed with samples from a standard exponential distribution.

```
> E <- Empirical$new(samples = rexp(10000))
```

The summary function is used to quickly obtain key information about the empirical distribution.

```
> summary(E)

Empirical Probability Distribution.

  Quick Statistics
    Mean: 0.105954
    Variance: 1.140673
    Skewness: 0.05808027
    Ex. Kurtosis: -0.473978

 Support: (-2.50, -2.19,...,2.27, 2.66)  Scientific Type: R

 Traits: discrete; univariate
 Properties: asymmetric; platykurtic; positive skew
```

The distribution is compared to a (standard) Normal distribution and then (standard) Exponential distribution; output in figure 3.

```
> qqplot(E, Normal$new(), xlab = "Empirical", ylab = "Normal")
> qqplot(E, Exponential$new(), xlab = "Empirical", ylab = "Exponential")
```

The CDF of a bivariate empirical distribution is visualised; output in figure 4.

```
> plot(EmpiricalMV$new(data.frame(rnorm(100, mean = 3), rnorm(100))), fun = "cdf")
```

Learning from custom distributions

Whilst empirical distributions are useful when data samples have been generated, custom distributions can be used to build an entirely new probability distribution, though here we use a simple discrete uniform distribution. This example highlights the power of decorators to estimate distribution results without manual computation of every possible method. The output demonstrates the precision and accuracy of these results.

Below, a custom distribution is created and by including the decorators argument, all further methods are imputed numerically. The distribution is summarised for properties, traits and common results (this is possible with the 'CoreStatistics' decorator). The summary is identical to the analytic `DiscreteUniform` distribution.

```
> U <- Distribution$new(
+        name = "Discrete Uniform",
+        type = set6::Integers$new(), support = set6::Set$new(1:10),
+        pdf = function(x) ifelse(x < 1 | x > 10, 0, rep(1/10,length(x))),
+        decorators = c("CoreStatistics", "ExoticStatistics", "FunctionImputation"))
> summary(U)

Discrete Uniform

  Quick Statistics
    Mean: 5.5
    Variance: 8.25
    Skewness: 0
    Ex. Kurtosis: -1.224242

 Support: {1, 2,...,9, 10}  Scientific Type: Z

 Traits: discrete; univariate
 Properties: asymmetric; platykurtic; no skew

 Decorated with:  CoreStatistics, ExoticStatistics, FunctionImputation
```

The CDF and simulation function are called (numerically imputed with the `FunctionImputation` decorator), the hazard function from the `ExoticStatistics` decorator, and the `kthmoment` function from the `CoreStatistics` decorator.

```
> U$cdf(1:10)
 [1] 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
> U$rand(10)
 [1]  8 10  5  8  5 10  6  7  1  4
> U$hazard(2)
[1] 0.125
> U$kthmoment(2)
[1] 8.25
```

Composite distribution modelling

Composite distributions are an essential part of any distribution software, the following example demonstrates two types of composites: composition via distribution transformation (truncation) and composition via mixtures and vectors.

First, a Binomial distribution is constructed and truncated between 1 and 5, the CDF of the new distribution is queried.

```
> TB <- truncate(
   Binomial$new(size = 20, prob = 0.5),
   lower = 1,
   upper = 5
   )
> round(TB$cdf(0:6), 4)
[1] 0.0000 0.0000 0.0088 0.0613 0.2848 1.0000 1.0000
```

Next, a vector distribution is constructed of two Normal distributions, with respective means 1 and 2 and unit standard deviation. The parameters are queried (some columns suppressed).

```
> V <- VectorDistribution$new(distribution = "Normal",
+                             params = data.frame(mean = 1:2))
> V$parameters()
          id value support
1: Norm1_mean     1       R
2:  Norm1_var     1      R+
3:   Norm1_sd     1      R+
4: Norm1_prec     1      R+
5: Norm2_mean     2       R
6:  Norm2_var     1      R+
7:   Norm2_sd     1      R+
8: Norm2_prec     1      R+
```

Vectorisation is possible across distributions, samples, and both. In the example below, the first call to $pdf evaluates both distributions at (1, 2), the second call evaluates the first at (1) and the second at (2), and the third call evaluates the first at (1, 2) and the second at (3, 4).

```
> V$pdf(1:2)
       Norm1     Norm2
1: 0.3989423 0.2419707
2: 0.2419707 0.3989423
> V$pdf(1, 2)
       Norm1     Norm2
1: 0.3989423 0.3989423
> V$pdf(1:2, 3:4)
       Norm1      Norm2
1: 0.3989423 0.24197072
2: 0.2419707 0.05399097
```

Finally, a mixture distribution with uniform weights is constructed from a $Normal(2, 1)$ distribution and an $Exponential(1)$.

```
> MD <- MixtureDistribution$new(
+   list(Normal$new(mean = 2, sd = 1),
+        Exponential$new(rate = 1)
+   )
+ )
> MD$pdf(1:5)
[1] 0.304925083 0.267138782 0.145878896 0.036153303 0.005584898
> MD$cdf(1:5)
[1] 0.3953879 0.6823324 0.8957788 0.9794671 0.9959561
> MD$rand(5)
[1] 3.6664473 0.1055126 0.6092939 0.8880799 3.4517465
```

Future work

Whilst **distr6** fulfils its primary purpose as an R6 interface for probability distributions with basic features, it is not consider 'feature-complete' as it currently lacks many of the important features included in **distr** and other related software. **distr6** is in constant development and has an active GitHub with open issues and projects. Some concrete short-term goals include:

- A more generalised convolution interface
- Expanding truncation to other interval types (currently only left-open is supported)
- Extending the FunctionImputation decorator to work on higher-order distributions as well as to improve speed and accuracy.
- Exposing further internal functionality for more user-control over numerical results.
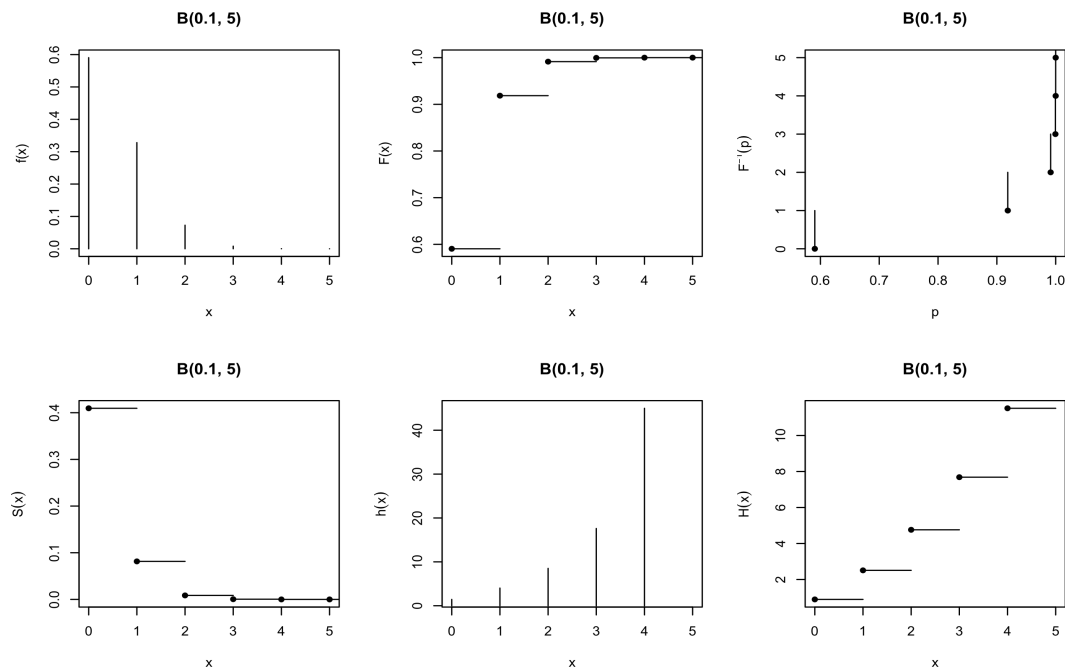
Figure 2: Visualising a Binomial(0.1, 5) distribution, b, with `plot(b, fun = "all")`. Functions clockwise from top-left: probability mass, cumulative distribution, quantile (inverse cdf), cumulative hazard, hazard, and survival.

Summary

distr6 introduces a robust and scalable object-oriented interface for probability distributions to R and aims to be the first-stop for class object-oriented probability distributions in R. By making use of R6, every implemented distribution is clearly defined with properties, traits, and analytic results. Whilst R **stats** is limited to very basic `dpqr` functions for representing evaluated distributions, **distr6** ensures that probability distributions are treated as complex mathematical objects.

Future updates of the package will include adding further numerical approximation strategies in the decorators to allow users to choose different methods (instead of being forced to use one). Additionally, the extensions to R6 could be abstracted into an independent package in order to better benefit the R community.

distr6 is released under the MIT licence on GitHub and CRAN. Extended documentation, tutorials, and examples are available at `https://alan-turing-institute.github.io/distr6/`. Code quality is monitored and maintained by an extensive suite of unit tests on multiple continuous integration systems.
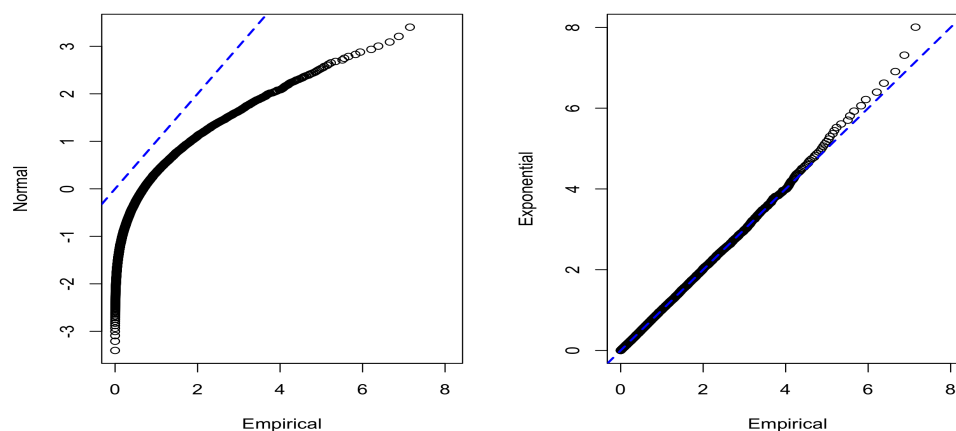
Acknowledgments

Figure 3: Q-Q plots comparing an unknown Empirical distribution, E, to theoretical Normal (left) and Exponential (right) distributions with `qqplot(E, Normal$new())` and `qqplot(E, Exponential$new())` respectively.

Bibliography

J. M. Chambers. Object-Oriented Programming, Functional Programming and R. *Statist. Sci.*, 29(2): 167–180, 2014. ISSN 0883-4237. URL https://projecteuclid.org:443/euclid.ss/1408368569. [p]

W. Chang. R6: Classes with Reference Semantics, 2018. URL https://cran.r-project.org/package=R6. [p]

C. Dutang, V. Goulet, and M. Pigeon. actuar: An R Package for Actuarial Science. *Journal of Statistical Software*, 25(7):38, 2008. [p]

D. Eddelbuettel and R. Francois. Rcpp: Seamless R and C++ Integration. *Journal of Statistical Software*, 40(8):1–18, 2011. URL http://www.jstatsoft.org/v40/i08/. [p]

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Software, 1996. ISSN ISBN: 0-201-63361-2. [p]

A. Hayes and R. Moller-Trane. distributions3: Probability Distributions as S3 Objects, 2019. URL https://cran.r-project.org/package=distributions3. [p]

F. J. Király, M. Löning, A. Blaom, A. Guecioueur, and R. Sonabend. Designing Machine Learning Toolboxes: Concepts, Principles and Patterns. *arXiv*, jan 2021. URL http://arxiv.org/abs/2101.04938. [p]

D. Lin, J. M. White, S. Byrne, D. Bates, A. Noack, J. Pearson, A. Arslan, K. Squire, D. Anthoff, T. Papamarkou, M. Besançon, J. Drugowitsch, and M. Schauer. JuliaStats/Distributions.jl: a Julia package for probability distributions and associated functions. 2019. doi: 10.5281/zenodo.2647458. [p]

O. Mersmann. microbenchmark: Accurate Timing Functions, 2019. URL https://cran.r-project.org/package=microbenchmark. [p]

M. O'Hara-Wild and A. Hayes. distributional: Vectorised Probability Distributions, 2020. URL https://cran.r-project.org/package=distributional. [p]

P. Ruckdeschel, M. Kohl, T. Stabla, and F. Camphausen. S4 Classes for Distributions, 2006. URL https://cran.r-project.org/package=distr. [p]

L. Sablica and K. Hornik. mistr: A Computational Framework for Mixture and Composite Distributions. *The R Journal*, 12(1):283, 2020. ISSN 2073-4859. doi: 10.32614/RJ-2020-003. URL https://journal.r-project.org/archive/2020/RJ-2020-003/index.html. [p]

R. Sonabend. R62S3: Automatic Method Generation from R6, may 2019. URL https://cran.r-project.org/package=R62S3. [p]
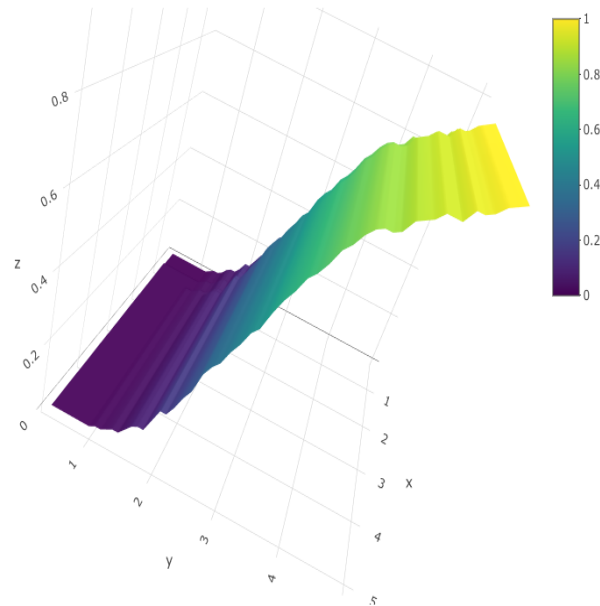
Figure 4: Visualising a bivariate empirical distribution, `E`, with `plot("E", fun = "cdf")`.

R. Sonabend and F. J. Kiraly. set6: R6 Mathematical Sets Interface. *Journal of Open Source Software*, 5(55): 2598, nov 2020. ISSN 2475-9066. doi: 10.21105/joss.02598. URL https://joss.theoj.org/papers/10.21105/joss.02598. [p]

T. Wolodzko. extraDistr: Additional Univariate and Multivariate Distributions, 2019. URL https://cran.r-project.org/package=extraDistr. [p]

Raphael Sonabend
Department of Statistical Science
University College London
Gower Street
London, WC1E 6BT, United Kingdom
0000-0001-9225-4654
raphael.sonabend.15@ucl.ac.uk

Franz J. Király
Department of Statistical Science
University College London
Gower Street
London, WC1E 6BT, United Kingdom
f.kiraly@ucl.ac.uk