

# Package ‘adoptr’

October 12, 2022

**Type** Package

**Title** Adaptive Optimal Two-Stage Designs in R

**Version** 1.0.0

**Description** Optimize one or two-arm, two-stage designs for clinical trials with respect to several pre-implemented objective criteria or implement custom objectives.

Optimization under uncertainty and conditional (given stage-one outcome) constraints are supported.

See <[doi:10.1002/sim.8291](https://doi.org/10.1002/sim.8291)> and <[doi:10.18637/jss.v098.i09](https://doi.org/10.18637/jss.v098.i09)> for details.

**License** MIT + file LICENSE

**Encoding** UTF-8

**Suggests** knitr, rmarkdown, testthat, covr, rpact, vdiff, pwr, dplyr, ggplot2, tidyr, gridExtra, bookdown

**Imports** nloptr, methods, glue

**VignetteBuilder** knitr

**Collate** adoptr.R util.R DataDistribution.R BinomialDistribution.R NormalDistribution.R StudentDistribution.R Prior.R PointMassPrior.R ContinuousPrior.R TwoStageDesign.R GroupSequentialDesign.R OneStageDesign.R Scores.R constraints.R minimize.R ConditionalPower.R ConditionalSampleSize.R regularization.R CompositeScore.R MaximumSampleSize.R

**RoxygenNote** 7.1.1

**BugReports** <https://github.com/kkmann/adoptr/issues>

**URL** <https://github.com/kkmann/adoptr>

**NeedsCompilation** no

**Author** Kevin Kunzmann [aut, cre] (<<https://orcid.org/0000-0002-1140-7143>>), Maximilian Pilz [aut] (<<https://orcid.org/0000-0002-9685-1613>>)

**Maintainer** Kevin Kunzmann <[kevin.kunzmann@mrc-bsu.cam.ac.uk](mailto:kevin.kunzmann@mrc-bsu.cam.ac.uk)>

**Repository** CRAN

**Date/Publication** 2021-06-28 13:00:02 UTC

**R topics documented:**

adoptr	3
AverageN2-class	3
Binomial-class	4
bounds	6
c2	6
composite	8
condition	9
ConditionalPower-class	10
ConditionalSampleSize-class	11
Constraints	12
ContinuousPrior-class	13
cumulative_distribution_function	15
DataDistribution-class	16
expectation	16
get_initial_design	17
get_lower_boundary_design	18
GroupSequentialDesign-class	20
make_tunable	21
MaximumSampleSize-class	22
minimize	23
n1	25
N1-class	26
Normal-class	27
OneStageDesign-class	28
plot,TwoStageDesign-method	29
PointMassPrior-class	30
posterior	31
predictive_cdf	32
predictive_pdf	33
print.adoptrOptimizationResult	34
Prior-class	34
probability_density_function	35
Scores	36
simulate,TwoStageDesign,numeric-method	37
Student-class	38
subject_to	39
tunable_parameters	40
TwoStageDesign-class	41

**Description**

The **adoptr** package provides functionality to explore custom optimal two-stage designs for one- or two-arm superiority tests. For more details on the theoretical background see <doi:10.1002/sim.8291> and <doi:10.18637/jss.v098.i09>. **adoptr** makes heavy use of the S4 class system. A good place to start learning about it can be found [here](#).

**Quickstart**

For a sample workflow and a quick demo of the capabilities, see [here](#).

A more detailed description of the background and the usage of **adoptr** can be found [here](#) or here <doi:10.18637/jss.v098.i09> .

A variety of examples is presented in the validation report hosted [here](#).

**Designs**

**adoptr** currently supports [TwoStageDesign](#), [GroupSequentialDesign](#), and [OneStageDesign](#).

**Data distributions**

Currently, the only implemented data distribution is [Normal](#) (one or two arms).

**Priors**

Both [ContinuousPrior](#) and [PointMassPrior](#) are supported for the single parameter of a [DataDistribution](#).

**Scores**

See [Scores](#) for information on the basic system of representing scores. Available scores are [ConditionalPower](#), [ConditionalSampleSize](#), [Power](#), and [ExpectedSampleSize](#).

**Description**

Implements the L1-norm of the design's stage-two sample size function. The average of the stage-two sample size without weighting with the data distribution is computed. This can be interpreted as integration over a uniform prior on the continuation region.

**Usage**

```
AverageN2(label = NA_character_)

## S4 method for signature 'AverageN2,TwoStageDesign'
evaluate(s, design, optimization = FALSE, subdivisions = 10000L, ...)
```

**Arguments**

label	object label (string)
s	<a href="#">Score</a> object
design	object
optimization	logical, if TRUE uses a relaxation to real parameters of the underlying design; used for smooth optimization.
subdivisions	number of subdivisions to use for adaptive integration (only affects non-optimization code)
...	further optional arguments

**Value**

an object of class [AverageN2](#)

**See Also**

[N1](#) for penalizing n1 values

**Examples**

```
avn2 <- AverageN2()

evaluate(
  AverageN2(),
  TwoStageDesign(100, 0.5, 1.5, 60.0, 1.96, order = 5L)
) # 60
```

---

Binomial-class

*Binomial data distribution*

---

**Description**

Implements the normal approximation for a test on rates. The response rate in the control group,  $r_C$ , has to be specified by `rate_control`. The null hypothesis is:  $r_E \leq r_C$ , where  $r_E$  denotes the response rate in the intervention group. It is tested against the alternative  $r_E > r_C$ . The test statistic is given as  $X_1 = \sqrt{n}(r_E - r_C) / \sqrt{2r_0(1 - r_0)}$ , where  $r_0$  denotes the mean between  $r_E$  and  $r_C$  in the two-armed case, and  $r_E$  in the one-armed case. All priors have to be defined for the rate difference  $r_E - r_C$ .

**Usage**

```
Binomial(rate_control, two_armed = TRUE)

## S4 method for signature 'Binomial'
quantile(x, probs, n, theta, ...)

## S4 method for signature 'Binomial,numeric'
simulate(object, nsim, n, theta, seed = NULL, ...)
```

**Arguments**

rate_control	assumed response rate in control group
two_armed	logical indicating if a two-armed trial is regarded
x	outcome
probs	vector of probabilities
n	sample size
theta	distribution parameter
...	further optional arguments
object	object of class Binomial
nsim	number of simulation runs
seed	random seed

**Details**

Note that `simulate` for class `Binomial` simulates the normal approximation of the test statistic.

**Slots**

rate\_control cf. parameter 'rate\_control'

**See Also**

see [probability\\_density\\_function](#) and [cumulative\\_distribution\\_function](#) to evaluate the pdf and the cdf, respectively.

**Examples**

```
datadist <- Binomial(rate_control = 0.2, two_armed = FALSE)
```

---

bounds	<i>Get support of a prior or data distribution</i>
--------	--

---

### Description

bounds() returns the range of the support of a prior or data distribution.

### Usage

```
bounds(dist, ...)

## S4 method for signature 'PointMassPrior'
bounds(dist, ...)

## S4 method for signature 'ContinuousPrior'
bounds(dist, ...)
```

### Arguments

dist	a univariate <a href="#">distribution</a> object
...	further optional arguments

### Value

numeric of length two, c(lower, upper)

### Examples

```
bounds(PointMassPrior(c(0, .5), c(.3, .7)))
# > 0.3 0.7

bounds(ContinuousPrior(function(x) dunif(x, .2, .4), c(.2, .4)))
# > 0.2 0.4
```

---

c2	<i>Query critical values of a design</i>
----	--

---

### Description

Methods to access the stage-two critical values of a [TwoStageDesign](#). c2 returns the stage-two critical value conditional on the stage-one test statistic.

**Usage**

```
c2(d, x1, ...)  
  
## S4 method for signature 'TwoStageDesign,numeric'  
c2(d, x1, ...)  
  
## S4 method for signature 'OneStageDesign,numeric'  
c2(d, x1, ...)
```

**Arguments**

d	design
x1	stage-one test statistic
...	further optional arguments

**See Also**

[TwoStageDesign](#), see [n](#) for accessing the sample size of a design

**Examples**

```
design <- TwoStageDesign(  
  n1 = 25,  
  c1f = 0,  
  c1e = 2.5,  
  n2 = 50,  
  c2 = 1.96,  
  order = 7L  
)  
  
c2(design, 2.2) # 1.96  
c2(design, 3.0) # -Inf  
c2(design, -1.0) # Inf  
  
design <- TwoStageDesign(  
  n1 = 25,  
  c1f = 0,  
  c1e = 2.5,  
  n2 = 50,  
  c2 = 1.96,  
  order = 7L  
)  
  
c2(design, 2.2) # 1.96  
c2(design, 3.0) # -Inf  
c2(design, -1.0) # Inf
```

---

 composite

*Score Composition*


---

### Description

composite defines new composite scores by point-wise evaluation of scores in any valid numerical expression.

### Usage

```
composite(expr, label = NA_character_)

## S4 method for signature 'CompositeScore,TwoStageDesign'
evaluate(s, design, ...)
```

### Arguments

expr	Expression (in curly brackets); must contain at least one score variable; if multiple scores are used, they must either all be conditional or unconditional. Currently, no non-score variables are supported
label	object label (string)
s	object of class CompositeScore
design	object
...	further optional arguments

### Value

an object of class CompositeConditionalScore or CompositeUnconditionalScore depending on the class of the scores used in expr

### See Also

[Scores](#)

### Examples

```
ess <- ExpectedSampleSize(Normal(), PointMassPrior(.4, 1))
power <- Power(Normal(), PointMassPrior(.4, 1))

# linear combination:
composite({ess - 50*power})

# control flow (e.g. for and while loops)
composite({
  res <- 0
  for (i in 1:3) {
    res <- res + ess
  }
})
```



```

    }
    res
  })

# functional composition
composite({log(ess)})
cp <- ConditionalPower(Normal(), PointMassPrior(.4, 1))
composite({3*cp})

```

---

condition	<i>Condition a prior on an interval</i>
-----------	---

---

### Description

Restrict an object of class `Prior` to a sub-interval and re-normalize the PDF.

### Usage

```

condition(dist, interval, ...)

## S4 method for signature 'PointMassPrior,numeric'
condition(dist, interval, ...)

## S4 method for signature 'ContinuousPrior,numeric'
condition(dist, interval, ...)

```

### Arguments

<code>dist</code>	a univariate <a href="#">distribution</a> object
<code>interval</code>	length-two numeric vector giving the parameter interval to condition on
<code>...</code>	further optional arguments

### Value

conditional `Prior` on given interval

### Examples

```

tmp <- condition(PointMassPrior(c(0, .5), c(.3, .7)), c(-1, .25))
expectation(tmp, identity) # 0

tmp <- condition(
  ContinuousPrior(function(x) dunif(x, .2, .4), c(.2, .4)),
  c(.3, .5)
)
bounds(tmp) # c(.3, .4)

```

---

 ConditionalPower-class

 (Conditional) Power of a Design
 

---

### Description

This score evaluates  $P[X_2 > c_2(\text{design}, X_1) | X_1 = x_1]$ . Note that the distribution of  $X_2$  is the posterior predictive after observing  $X_1 = x_1$ .

### Usage

```
ConditionalPower(dist, prior, label = "Pr[x2>=c2(x1)|x1]")
```

```
Power(dist, prior, label = "Pr[x2>=c2(x1)]")
```

```
## S4 method for signature 'ConditionalPower,TwoStageDesign'
evaluate(s, design, x1, optimization = FALSE, ...)
```

### Arguments

dist	a univariate <a href="#">distribution</a> object
prior	a <a href="#">Prior</a> object
label	object label (string)
s	<a href="#">Score</a> object
design	object
x1	stage-one test statistic
optimization	logical, if TRUE uses a relaxation to real parameters of the underlying design; used for smooth optimization.
...	further optional arguments

### See Also

[Scores](#)

### Examples

```
prior <- PointMassPrior(.4, 1)
cp <- ConditionalPower(Normal(), prior)
evaluate(
  cp,
  TwoStageDesign(50, .0, 2.0, 50, 2.0, order = 5L),
  x1 = 1
)
# these two are equivalent:
expected(cp, Normal(), prior)
Power(Normal(), prior)
```

---

ConditionalSampleSize-class  
*(Conditional) Sample Size of a Design*

---

### Description

This score simply evaluates  $n(d, x_1)$  for a design  $d$  and the first-stage outcome  $x_1$ . The data distribution and prior are only relevant when it is integrated.

### Usage

```
ConditionalSampleSize(label = "n(x1)")
ExpectedSampleSize(dist, prior, label = "E[n(x1)]")

## S4 method for signature 'ConditionalSampleSize,TwoStageDesign'
evaluate(s, design, x1, optimization = FALSE, ...)
```

### Arguments

label	object label (string)
dist	a univariate <a href="#">distribution</a> object
prior	a <a href="#">Prior</a> object
s	<a href="#">Score</a> object
design	object
x1	stage-one test statistic
optimization	logical, if TRUE uses a relaxation to real parameters of the underlying design; used for smooth optimization.
...	further optional arguments

### See Also

[Scores](#)

### Examples

```
design <- TwoStageDesign(50, .0, 2.0, 50, 2.0, order = 5L)
prior <- PointMassPrior(.4, 1)

css <- ConditionalSampleSize()
evaluate(css, design, c(0, .5, 3))

ess <- ExpectedSampleSize(Normal(), prior)

# those two are equivalent
evaluate(ess, design)
```

```
evaluate(expected(css, Normal(), prior), design)
```

---

 Constraints

 Formulating Constraints
 

---

### Description

Conceptually, constraints work very similar to scores (any score can be put in a constraint). Currently, constraints of the form 'score  $\leq$  x', 'x  $\leq$  score' and 'score  $\geq$  score' are admissible.

### Usage

```
## S4 method for signature 'Constraint,TwoStageDesign'
evaluate(s, design, optimization = FALSE, ...)

## S4 method for signature 'ConditionalScore,numeric'
e1 <= e2

## S4 method for signature 'ConditionalScore,numeric'
e1 >= e2

## S4 method for signature 'numeric,ConditionalScore'
e1 <= e2

## S4 method for signature 'numeric,ConditionalScore'
e1 >= e2

## S4 method for signature 'ConditionalScore,ConditionalScore'
e1 <= e2

## S4 method for signature 'ConditionalScore,ConditionalScore'
e1 >= e2

## S4 method for signature 'UnconditionalScore,numeric'
e1 <= e2

## S4 method for signature 'UnconditionalScore,numeric'
e1 >= e2

## S4 method for signature 'numeric,UnconditionalScore'
e1 <= e2

## S4 method for signature 'numeric,UnconditionalScore'
e1 >= e2
```

```
## S4 method for signature 'UnconditionalScore,UnconditionalScore'
e1 <= e2
```

```
## S4 method for signature 'UnconditionalScore,UnconditionalScore'
e1 >= e2
```

### Arguments

s	Score object
design	object
optimization	logical, if TRUE uses a relaxation to real parameters of the underlying design; used for smooth optimization.
...	further optional arguments
e1	left hand side (score or numeric)
e2	right hand side (score or numeric)

### See Also

[minimize](#)

### Examples

```
design <- OneStageDesign(50, 1.96)

cp <- ConditionalPower(Normal(), PointMassPrior(0.4, 1))
pow <- Power(Normal(), PointMassPrior(0.4, 1))

# unconditional power constraint
constraint1 <- pow >= 0.8
evaluate(constraint1, design)

# conditional power constraint
constraint2 <- cp >= 0.7
evaluate(constraint2, design, .5)
constraint3 <- 0.7 <= cp # same as constraint2
evaluate(constraint3, design, .5)
```

---

ContinuousPrior-class *Continuous univariate prior distributions*

---

### Description

ContinuousPrior is a sub-class of [Prior](#) implementing a generic representation of continuous prior distributions over a compact interval on the real line.

**Usage**

```
ContinuousPrior(
  pdf,
  support,
  order = 10,
  label = NA_character_,
  tighten_support = FALSE,
  check_normalization = TRUE
)
```

**Arguments**

pdf	vectorized univariate PDF function
support	numeric vector of length two with the bounds of the compact interval on which the pdf is positive.
order	integer, integration order of the employed Gaussian quadrature integration rule to evaluate scores. Automatically set to <code>length(n2_pivots)</code> if <code>length(n2_pivots) == length(c2_pivots) &gt; 1</code> , otherwise <code>c2</code> and <code>n2</code> are taken to be constant in stage-two and replicated to match the number of pivots specified by <code>order</code>
label	object label (string)
tighten_support	logical indicating if the support should be tightened
check_normalization	logical indicating if it should be checked that pdf defines a density.

**Slots**

pdf	cf. parameter 'pdf'
support	cf. parameter 'support'
pivots	normalized pivots for integration rule (in [-1, 1]) the actual pivots are scaled to the support of the prior
weights	weights of of integration rule at pivots for approximating integrals over delta

**See Also**

Discrete priors are supported via [PointMassPrior](#)

**Examples**

```
ContinuousPrior(function(x) 2*x, c(0, 1))
```

---

```
cumulative_distribution_function  
      Cumulative distribution function
```

---

### Description

`cumulative_distribution_function` evaluates the cumulative distribution function of a specific distribution `dist` at a point `x`.

### Usage

```
cumulative_distribution_function(dist, x, n, theta, ...)  
  
## S4 method for signature 'Binomial,numeric,numeric,numeric'  
cumulative_distribution_function(dist, x, n, theta, ...)  
  
## S4 method for signature 'Normal,numeric,numeric,numeric'  
cumulative_distribution_function(dist, x, n, theta, ...)  
  
## S4 method for signature 'Student,numeric,numeric,numeric'  
cumulative_distribution_function(dist, x, n, theta, ...)
```

### Arguments

<code>dist</code>	a univariate <a href="#">distribution</a> object
<code>x</code>	outcome
<code>n</code>	sample size
<code>theta</code>	distribution parameter
<code>...</code>	further optional arguments

### Details

If the distribution is [Binomial](#), `theta` denotes the rate difference between intervention and control group. Then, the mean is assumed to be  $\sqrt{n}theta$ .

If the distribution is [Normal](#), then the mean is assumed to be  $\sqrt{n}theta$ .

### Examples

```
cumulative_distribution_function(Binomial(.1, TRUE), 1, 50, .3)  
  
cumulative_distribution_function(Normal(), 1, 50, .3)  
  
cumulative_distribution_function(Student(two_armed = FALSE), .75, 50, .9)
```

---

 DataDistribution-class

*Data distributions*


---

### Description

DataDistribution is an abstract class used to represent the distribution of a sufficient statistic  $x$  given a sample size  $n$  and a single parameter value  $\theta$ .

### Arguments

<code>x</code>	outcome
<code>n</code>	sample size
<code>theta</code>	distribution parameter
<code>...</code>	further optional arguments

### Details

This abstraction layer allows the representation of t-distributions (unknown variance), normal distribution (known variance), and normal approximation of a binary endpoint. Currently, the two implemented versions are [Normal-class](#) and [Binomial-class](#).

The logical option `two_armed` allows to decide whether a one-arm or a two-arm (the default) design should be computed. In the case of a two-arm design all sample sizes are per group.

### Slots

`two_armed` Logical that indicates if a two-arm design is assumed.

### Examples

```
normaldist <- Normal(two_armed = FALSE)
binomialdist <- Binomial(rate_control = .25, two_armed = TRUE)
```

---

 expectation

*Expected value of a function*


---

### Description

Computes the expected value of a vectorized, univariate function  $f$  with respect to a distribution `dist`. I.e.,  $\mathbf{E}[f(X)]$ .



**Usage**

```
expectation(dist, f, ...)  
  
## S4 method for signature 'PointMassPrior`,`function`'  
expectation(dist, f, ...)  
  
## S4 method for signature 'ContinuousPrior`,`function`'  
expectation(dist, f, ...)
```

**Arguments**

dist	a univariate <a href="#">distribution</a> object
f	a univariate function, must be vectorized
...	further optional arguments

**Value**

numeric, expected value of f with respect to dist

**Examples**

```
expectation(PointMassPrior(c(0, .5), c(.3, .7)), identity)  
# > .35  
  
expectation(  
  ContinuousPrior(function(x) dunif(x, .2, .4), c(.2, .4)),  
  identity  
)  
# > 0.3
```

---

get\_initial\_design      *Initial design*

---

**Description**

The optimization method [minimize](#) requires an initial design for optimization. The function `get_initial_design` provides an initial guess based on a fixed design that fulfills constraints on type I error rate and power. Note that a situation-specific initial design may be much more efficient.

**Usage**

```
get_initial_design(  
  theta,  
  alpha,  
  beta,  
  type = c("two-stage", "group-sequential", "one-stage"),
```

```

    dist = Normal(),
    order = 7L,
    ...
  )

```

### Arguments

theta	the alternative effect size in the normal case, the rate difference under the alternative in the binomial case
alpha	maximal type I error rate
beta	maximale type II error rate
type	is a two-stage, group-sequential, or one-stage design required?
dist	distribution of the test statistic
order	desired integration order
...	further optional arguments

### Details

The distribution of the test statistic is specified by `dist`. The default assumes a two-armed z-test.

### Examples

```

init <- get_initial_design(
  theta = 0.3,
  alpha = 0.025,
  beta = 0.2,
  type = "two-stage",
  dist = Normal(two_armed = FALSE),
  order = 7L
)

```

---

get\_lower\_boundary\_design

*Boundary designs*

---

### Description

The optimization method `minimize` is based on the package `nloptr`. This requires upper and lower boundaries for optimization. Such boundaries can be computed via `lower_boundary_design` respectively `upper_boundary_design`. They are implemented by default in `minimize`. Note that `minimize` allows the user to define its own boundary designs, too.

**Usage**

```
get_lower_boundary_design(initial_design, ...)

get_upper_boundary_design(initial_design, ...)

## S4 method for signature 'OneStageDesign'
get_lower_boundary_design(initial_design, n1 = 1, c1_buffer = 2, ...)

## S4 method for signature 'GroupSequentialDesign'
get_lower_boundary_design(
  initial_design,
  n1 = 1,
  n2_pivots = 1,
  c1_buffer = 2,
  c2_buffer = 2,
  ...
)

## S4 method for signature 'TwoStageDesign'
get_lower_boundary_design(
  initial_design,
  n1 = 1,
  n2_pivots = 1,
  c1_buffer = 2,
  c2_buffer = 2,
  ...
)

## S4 method for signature 'OneStageDesign'
get_upper_boundary_design(
  initial_design,
  n1 = 5 * initial_design@n1,
  c1_buffer = 2,
  ...
)

## S4 method for signature 'GroupSequentialDesign'
get_upper_boundary_design(
  initial_design,
  n1 = 5 * initial_design@n1,
  n2_pivots = 5 * initial_design@n2_pivots,
  c1_buffer = 2,
  c2_buffer = 2,
  ...
)

## S4 method for signature 'TwoStageDesign'
get_upper_boundary_design(
```

```

    initial_design,
    n1 = 5 * initial_design@n1,
    n2_pivots = 5 * initial_design@n2_pivots,
    c1_buffer = 2,
    c2_buffer = 2,
    ...
  )

```

### Arguments

<code>initial_design</code>	The initial design
<code>...</code>	optional arguments The values <code>c1f</code> and <code>c1e</code> from the initial design are shifted to <code>c1f - c1_buffer</code> and <code>c1e - c1_buffer</code> in <code>get_lower_boundary_design</code> , respectively, to <code>c1f + c1_buffer</code> and <code>c1e + c1_buffer</code> in <code>get_upper_boundary_design</code> . This is handled analogously with <code>c2_pivots</code> and <code>c2_buffer</code> .
<code>n1</code>	bound for the first-stage sample size <code>n1</code>
<code>c1_buffer</code>	shift of the early-stopping boundaries from the initial ones
<code>n2_pivots</code>	bound for the second-stage sample size <code>n2</code>
<code>c2_buffer</code>	shift of the final decision boundary from the initial one

### Examples

```

initial_design <- TwoStageDesign(
  n1   = 25,
  c1f  = 0,
  c1e  = 2.5,
  n2   = 50,
  c2   = 1.96,
  order = 7L
)
get_lower_boundary_design(initial_design)

```

---

GroupSequentialDesign-class

*Group-sequential two-stage designs*

---

### Description

Group-sequential designs are a sub-class of the `TwoStageDesign` class with constant stage-two sample size. See [TwoStageDesign](#) for slot details. Any group-sequential design can be converted to a fully flexible `TwoStageDesign` (see examples section).

**Usage**

```
GroupSequentialDesign(n1, c1f, c1e, n2_pivots, c2_pivots, order = NULL, ...)

## S4 method for signature 'GroupSequentialDesign'
TwoStageDesign(n1, ...)
```

**Arguments**

n1	stage one sample size or GroupSequentialDesign object to convert (overloaded from <a href="#">TwoStageDesign</a> )
c1f	early futility stopping boundary
c1e	early efficacy stopping boundary
n2_pivots	numeric of length one, stage-two sample size
c2_pivots	numeric vector, stage-two critical values on the integration pivot points
order	of the Gaussian uadrature rule to use for integration, set to length(c2_pivots) if NULL, otherwise first value of c2_pivots is repeated 'order'-times.
...	further optional arguments

**See Also**

[TwoStageDesign](#) for superclass and inherited methods

**Examples**

```
design <- GroupSequentialDesign(25, 0, 2, 25, c(1, 1.5, 2.5))
summary(design)

TwoStageDesign(design)
```

---

make\_tunable

*Fix parameters during optimization*

---

**Description**

The methods `make_fixed` and `make_tunable` can be used to modify the 'tunability' status of parameters in a [TwoStageDesign](#) object. Tunable parameters are optimized over, non-tunable ('fixed') parameters are considered given and not altered during optimization.

**Usage**

```
make_tunable(x, ...)  
  
## S4 method for signature 'TwoStageDesign'  
make_tunable(x, ...)  
  
make_fixed(x, ...)  
  
## S4 method for signature 'TwoStageDesign'  
make_fixed(x, ...)
```

**Arguments**

x	TwoStageDesign object
...	unquoted names of slots for which the tunability status should be changed.

**See Also**

[TwoStageDesign](#), [tunable\\_parameters](#) for converting tunable parameters of a design object to a numeric vector (and back), and [minimize](#) for the actual minimization procedure

**Examples**

```
design <- TwoStageDesign(25, 0, 2, 25, 2, order = 5)  
# default: all parameters are tunable (except integration pivots,  
# weights and tunability status itself)  
design@tunable  
  
# make n1 and the pivots of n2 fixed (not changed during optimization)  
design <- make_fixed(design, n1, n2_pivots)  
design@tunable  
  
# make them tunable again  
design <- make_tunable(design, n1, n2_pivots)  
design@tunable
```

---

MaximumSampleSize-class

*Maximum Sample Size of a Design*

---

**Description**

This score evaluates  $\max(n(d))$  for a design  $d$ .

**Usage**

```
MaximumSampleSize(label = "max(n(x1))")

## S4 method for signature 'MaximumSampleSize,TwoStageDesign'
evaluate(s, design, optimization = FALSE, ...)
```

**Arguments**

label	object label (string)
s	<a href="#">Score</a> object
design	object
optimization	logical, if TRUE uses a relaxation to real parameters of the underlying design; used for smooth optimization.
...	further optional arguments

**See Also**

[Scores](#) for general scores and [ConditionalSampleSize](#) for evaluating the sample size point-wise.

**Examples**

```
design <- TwoStageDesign(50, .0, 2.0, 50, 2.0, order = 5L)
mss <- MaximumSampleSize()
evaluate(mss, design)
```

---

minimize

*Find optimal two-stage design by constraint minimization*


---

**Description**

minimize takes an unconditional score and a constraint set (or no constraint) and solves the corresponding minimization problem using [nloptr](#) (using COBYLA by default). An initial design has to be defined. It is also possible to define lower- and upper-boundary designs. If this is not done, the boundaries are determined automatically heuristically.

**Usage**

```
minimize(
  objective,
  subject_to,
  initial_design,
  lower_boundary_design = get_lower_boundary_design(initial_design),
  upper_boundary_design = get_upper_boundary_design(initial_design),
  opts = list(algorithm = "NLOPT_LN_COBYLA", xtol_rel = 1e-05, maxeval = 10000),
  ...
)
```

**Arguments**

objective	objective function
subject_to	constraint collection
initial_design	initial guess (x0 for nloptr)
lower_boundary_design	design specifying the lower boundary.
upper_boundary_design	design specifying the upper boundary
opts	options list passed to nloptr
...	further optional arguments passed to <code>nloptr</code>

**Value**

a list with elements:

design	The resulting optimal design
nloptr_return	Output of the corresponding nloptr call
call_args	The arguments given to the optimization call

**Examples**

```
# Define Type one error rate
toer <- Power(Normal(), PointMassPrior(0.0, 1))

# Define Power at delta = 0.4
pow <- Power(Normal(), PointMassPrior(0.4, 1))

# Define expected sample size at delta = 0.4
ess <- ExpectedSampleSize(Normal(), PointMassPrior(0.4, 1))

# Compute design minimizing ess subject to power and toer constraints
## Not run:
minimize(

  ess,

  subject_to(
    toer <= 0.025,
    pow >= 0.9
  ),

  initial_design = TwoStageDesign(50, .0, 2.0, 60.0, 2.0, 5L)

)

## End(Not run)
```



---

n1 *Query sample size of a design*

---

### Description

Methods to access the stage-one, stage-two, or overall sample size of a [TwoStageDesign](#). `n1` returns the first-stage sample size of a design, `n2` the stage-two sample size conditional on the stage-one test statistic and `n` the overall sample size  $n_1 + n_2$ . Internally, objects of the class `TwoStageDesign` allow non-natural, real sample sizes to allow smooth optimization (cf. [minimize](#) for details). The optional argument `round` allows to switch between the internal real representation and a rounded version (rounding to the next positive integer).

### Usage

```
n1(d, ...)

## S4 method for signature 'TwoStageDesign'
n1(d, round = TRUE, ...)

n2(d, x1, ...)

## S4 method for signature 'TwoStageDesign,numeric'
n2(d, x1, round = TRUE, ...)

n(d, x1, ...)

## S4 method for signature 'TwoStageDesign,numeric'
n(d, x1, round = TRUE, ...)

## S4 method for signature 'GroupSequentialDesign,numeric'
n2(d, x1, round = TRUE, ...)

## S4 method for signature 'OneStageDesign,numeric'
n2(d, x1, ...)
```

### Arguments

<code>d</code>	design
<code>...</code>	further optional arguments
<code>round</code>	logical should sample sizes be rounded to next integer?
<code>x1</code>	stage-one test statistic

### See Also

[TwoStageDesign](#), see [c2](#) for accessing the critical values

**Examples**

```

design <- TwoStageDesign(
  n1   = 25,
  c1f  = 0,
  c1e  = 2.5,
  n2   = 50,
  c2   = 1.96,
  order = 7L
)

n1(design) # 25
design@n1 # 25

n(design, x1 = 2.2) # 75

```

---

N1-class

*Regularize n1*


---

**Description**

N1 is a class that computes the n1 value of a design. This can be used as a score in [minimize](#).

**Usage**

```

N1(label = NA_character_)

## S4 method for signature 'N1,TwoStageDesign'
evaluate(s, design, optimization = FALSE, ...)

```

**Arguments**

label	object label (string)
s	<a href="#">Score</a> object
design	object
optimization	logical, if TRUE uses a relaxation to real parameters of the underlying design; used for smooth optimization.
...	further optional arguments

**Value**

an object of class [N1](#)

**See Also**

See [AverageN2](#) for a regularization of the second-stage sample size.

**Examples**

```
n1_score <- N1()

evaluate(
  N1(),
  TwoStageDesign(70, 0, 2, rep(60, 6), rep(1.7, 6))
) # 70
```

---

Normal-class

*Normal data distribution*


---

**Description**

Implements a normal data distribution for z-values given an observed z-value and stage size. Standard deviation is 1 and mean  $\theta\sqrt{n}$  where  $\theta$  is the standardized effect size. The option `two_armed` can be set to decide whether a one-arm or a two-arm design should be computed.

**Usage**

```
Normal(two_armed = TRUE)

## S4 method for signature 'Normal'
quantile(x, probs, n, theta, ...)

## S4 method for signature 'Normal,numeric'
simulate(object, nsim, n, theta, seed = NULL, ...)
```

**Arguments**

<code>two_armed</code>	logical indicating if a two-armed trial is regarded
<code>x</code>	outcome
<code>probs</code>	vector of probabilities
<code>n</code>	sample size
<code>theta</code>	distribution parameter
<code>...</code>	further optional arguments
<code>object</code>	object of class <code>Normal</code>
<code>nsim</code>	number of simulation runs
<code>seed</code>	random seed

**Details**

See [DataDistribution-class](#) for more details.

**See Also**

see [probability\\_density\\_function](#) and [cumulative\\_distribution\\_function](#) to evaluate the pdf and the cdf, respectively.

**Examples**

```
datadist <- Normal(two_armed = TRUE)
```

---

OneStageDesign-class *One-stage designs*

---

**Description**

OneStageDesign implements a one-stage design as special case of a two-stage design, i.e. as subclass of [TwoStageDesign](#). This is possible by defining  $n_2 = 0$ ,  $c = c_1^f = c_1^e$ ,  $c_2(x_1) = \text{ifelse}(x_1 < c, \text{Inf}, -\text{Inf})$ . No integration pivots etc are required (set to NaN).

**Usage**

```
OneStageDesign(n, c)

## S4 method for signature 'OneStageDesign'
TwoStageDesign(n1, order = 5L, eps = 0.01, ...)

## S4 method for signature 'OneStageDesign'
plot(x, y, ...)
```

**Arguments**

n	sample size (stage-one sample size)
c	rejection boundary ( $c = c_1^f = c_1^e$ )
n1	OneStageDesign object to convert, overloaded from <a href="#">TwoStageDesign</a>
order	integer $\geq 2$ , default is 5; order of Gaussian quadrature integration rule to use for new TwoStageDesign.
eps	numeric $> 0$ , default = .01; the single critical value c must be split in a continuation interval [c1f, c1e]; this is given by c +/- eps.
...	further optional arguments
x	design to plot
y	not used

**Details**

Note that the default [plot, TwoStageDesign-method](#) method is not supported for OneStageDesign objects.

**See Also**

[TwoStageDesign](#), [GroupSequentialDesign](#)

**Examples**

```
design <- OneStageDesign(30, 1.96)
summary(design)
design <- TwoStageDesign(design)
summary(design)
```

---

plot,TwoStageDesign-method

*Plot TwoStageDesign with optional set of conditional scores*

---

**Description**

This method allows to plot the stage-two sample size and decision boundary functions of a chosen design.

**Usage**

```
## S4 method for signature 'TwoStageDesign'
plot(x, y = NULL, ..., rounded = TRUE, k = 100)
```

**Arguments**

x	design to plot
y	not used
...	further named ConditionalScores to plot for the design and/or further graphic parameters
rounded	should n-values be rounded?
k	number of points to use for plotting

**Details**

[TwoStageDesign](#) and user-defined elements of the class [ConditionalScore](#).

**See Also**

[TwoStageDesign](#)

**Examples**

```
design <- TwoStageDesign(50, 0, 2, 50, 2, 5)
cp <- ConditionalPower(dist = Normal(), prior = PointMassPrior(.4, 1))
plot(design, "Conditional Power" = cp, cex.axis = 2)
```

---

PointMassPrior-class *Univariate discrete point mass priors*

---

### Description

PointMassPrior is a sub-class of [Prior](#) representing a univariate prior over a discrete set of points with positive probability mass.

### Usage

```
PointMassPrior(theta, mass, label = NA_character_)
```

### Arguments

theta	numeric vector of pivot points with positive prior mass
mass	numeric vector of probability masses at the pivot points (must sum to 1)
label	object label (string)

### Value

an object of class PointMassPrior, theta is automatically sorted in ascending order

### Slots

theta cf. parameter 'theta'  
mass cf. parameter 'mass'

### See Also

To represent continuous prior distributions use [ContinuousPrior](#).

### Examples

```
PointMassPrior(c(0, .5), c(.3, .7))
```

---

posterior	<i>Compute posterior distribution</i>
-----------	---------------------------------------

---

## Description

Return posterior distribution given observing stage-one outcome.

## Usage

```
posterior(dist, prior, x1, n1, ...)  
  
## S4 method for signature 'DataDistribution,PointMassPrior,numeric'  
posterior(dist, prior, x1, n1, ...)  
  
## S4 method for signature 'DataDistribution,ContinuousPrior,numeric'  
posterior(dist, prior, x1, n1, ...)
```

## Arguments

dist	a univariate <a href="#">distribution</a> object
prior	a <a href="#">Prior</a> object
x1	stage-one test statistic
n1	stage-one sample size
...	further optional arguments

## Value

Object of class [Prior](#)

## Examples

```
posterior(Normal(), PointMassPrior(0, 1), 2, 20)  
  
tmp <- ContinuousPrior(function(x) dunif(x, .2, .4), c(.2, .4))  
posterior(Normal(), tmp, 2, 20)
```

---

predictive\_cdf      *Predictive CDF*

---

### Description

predictive\_cdf() evaluates the predictive CDF of the model specified by a [DataDistribution](#) `dist` and [Prior](#) at the given stage-one outcome.

### Usage

```
predictive_cdf(dist, prior, x1, n1, ...)

## S4 method for signature 'DataDistribution,PointMassPrior,numeric'
predictive_cdf(dist, prior, x1, n1, ...)

## S4 method for signature 'DataDistribution,ContinuousPrior,numeric'
predictive_cdf(
  dist,
  prior,
  x1,
  n1,
  k = 10 * (prior@support[2] - prior@support[1]) + 1,
  ...
)
```

### Arguments

<code>dist</code>	a univariate <a href="#">distribution</a> object
<code>prior</code>	a <a href="#">Prior</a> object
<code>x1</code>	stage-one test statistic
<code>n1</code>	stage-one sample size
<code>...</code>	further optional arguments
<code>k</code>	number of pivots for crude integral approximation

### Value

numeric, value of the predictive CDF

### Examples

```
predictive_cdf(Normal(), PointMassPrior(.0, 1), 0, 20) # .5

tmp <- ContinuousPrior(function(x) dunif(x, .2, .4), c(.2, .4))
predictive_cdf(Normal(), tmp, 2, 20)
```



---

predictive_pdf	<i>Predictive PDF</i>
----------------	-----------------------

---

**Description**

predictive\_pdf() evaluates the predictive PDF of the model specified by a [DataDistribution](#) `dist` and [Prior](#) at the given stage-one outcome.

**Usage**

```
predictive_pdf(dist, prior, x1, n1, ...)

## S4 method for signature 'DataDistribution,PointMassPrior,numeric'
predictive_pdf(dist, prior, x1, n1, ...)

## S4 method for signature 'DataDistribution,ContinuousPrior,numeric'
predictive_pdf(
  dist,
  prior,
  x1,
  n1,
  k = 10 * (prior@support[2] - prior@support[1]) + 1,
  ...
)
```

**Arguments**

<code>dist</code>	a univariate <a href="#">distribution</a> object
<code>prior</code>	a <a href="#">Prior</a> object
<code>x1</code>	stage-one test statistic
<code>n1</code>	stage-one sample size
<code>...</code>	further optional arguments
<code>k</code>	number of pivots for crude integral approximation

**Value**

numeric, value of the predictive PDF

**Examples**

```
predictive_pdf(Normal(), PointMassPrior(.3, 1), 1.5, 20) # ~.343

tmp <- ContinuousPrior(function(x) dunif(x, .2, .4), c(.2, .4))
predictive_pdf(Normal(), tmp, 2, 20)
```

---

```
print.adoptrOptimizationResult
```

*Printing an optimization result*

---

**Description**

Printing an optimization result

**Usage**

```
print(x, ...)
```

**Arguments**

x	object to print
...	further arguments passed form other methods

---

```
Prior-class
```

*Univariate prior on model parameter*

---

**Description**

A Prior object represents a prior distribution on the single model parameter of a [DataDistribution](#) class object. Together a prior and data-distribution specify the class of the joint distribution of the test statistic, X, and its parameter, theta. Currently, **adoptr** only allows simple models with a single parameter. Implementations for [PointMassPrior](#) and [ContinuousPrior](#) are available.

**Details**

For an example on working with priors, see [here](#).

**See Also**

For the available methods, see [bounds](#), [expectation](#), [condition](#), [predictive\\_pdf](#), [predictive\\_cdf](#), [posterior](#)

**Examples**

```
disc_prior <- PointMassPrior(c(0.1, 0.25), c(0.4, 0.6))

cont_prior <- ContinuousPrior(
  pdf      = function(x) dnorm(x, mean = 0.3, sd = 0.2),
  support = c(-2, 3)
)
```

---

```
probability_density_function
      Probability density function
```

---

### Description

`probability_density_function` evaluates the probability density function of a specific distribution `dist` at a point `x`.

### Usage

```
probability_density_function(dist, x, n, theta, ...)

## S4 method for signature 'Binomial,numeric,numeric,numeric'
probability_density_function(dist, x, n, theta, ...)

## S4 method for signature 'Normal,numeric,numeric,numeric'
probability_density_function(dist, x, n, theta, ...)

## S4 method for signature 'Student,numeric,numeric,numeric'
probability_density_function(dist, x, n, theta, ...)
```

### Arguments

<code>dist</code>	a univariate <a href="#">distribution</a> object
<code>x</code>	outcome
<code>n</code>	sample size
<code>theta</code>	distribution parameter
<code>...</code>	further optional arguments

### Details

If the distribution is [Binomial](#), `theta` denotes the rate difference between intervention and control group. Then, the mean is assumed to be  $\sqrt{n}theta$ .

If the distribution is [Normal](#), then the mean is assumed to be  $\sqrt{n}theta$ .

### Examples

```
probability_density_function(Binomial(.2, FALSE), 1, 50, .3)

probability_density_function(Normal(), 1, 50, .3)

probability_density_function(Student(TRUE), 1, 40, 1.1)
```

---

 Scores

*Scores*


---

### Description

In `adoptr` scores are used to assess the performance of a design. This can be done either conditionally on the observed stage-one outcome or unconditionally. Consequently, score objects are either of class `ConditionalScore` or `UnconditionalScore`.

### Usage

```

expected(s, data_distribution, prior, ...)

## S4 method for signature 'ConditionalScore'
expected(s, data_distribution, prior, label = NA_character_, ...)

evaluate(s, design, ...)

## S4 method for signature 'IntegralScore,TwoStageDesign'
evaluate(s, design, optimization = FALSE, subdivisions = 10000L, ...)

```

### Arguments

<code>s</code>	<a href="#">Score</a> object
<code>data_distribution</code>	<a href="#">DataDistribution</a> object
<code>prior</code>	a <a href="#">Prior</a> object
<code>...</code>	further optional arguments
<code>label</code>	object label (string)
<code>design</code>	object
<code>optimization</code>	logical, if TRUE uses a relaxation to real parameters of the underlying design; used for smooth optimization.
<code>subdivisions</code>	maximal number of subdivisions when evaluating an integral score using adaptive quadrature ( <code>optimization = FALSE</code> )

### Details

All scores can be evaluated on a design using the `evaluate` method. Note that `evaluate` requires a third argument `x1` for conditional scores (observed stage-one outcome). Any `ConditionalScore` can be converted to a `UnconditionalScore` by forming its expected value using `expected`. The returned unconditional score is of class `IntegralScore`.

### See Also

[ConditionalPower](#), [ConditionalSampleSize](#), [composite](#)

**Examples**

```

design <- TwoStageDesign(
  n1   = 25,
  c1f  = 0,
  c1e  = 2.5,
  n2   = 50,
  c2   = 1.96,
  order = 7L
)
prior <- PointMassPrior(.3, 1)

# conditional
cp <- ConditionalPower(Normal(), prior)
expected(cp, Normal(), prior)
evaluate(cp, design, x1 = .5)

# unconditional
power <- Power(Normal(), prior)
evaluate(power, design)
evaluate(power, design, optimization = TRUE) # use non-adaptive quadrature

```

---

```

simulate,TwoStageDesign,numeric-method
Draw samples from a two-stage design

```

---

**Description**

simulate allows to draw samples from a given [TwoStageDesign](#).

**Usage**

```

## S4 method for signature 'TwoStageDesign,numeric'
simulate(object, nsim, dist, theta, seed = NULL, ...)

```

**Arguments**

object	TwoStageDesign to draw samples from
nsim	number of simulation runs
dist	data distribution
theta	location parameter of the data distribution
seed	random seed
...	further optional arguments

**Value**

`simulate()` returns a `data.frame` with `nsim` rows and for each row (each simulation run) the following columns

- `theta` The effect size
- `n1` First-stage sample size
- `c1f` Stopping for futility boundary
- `c1e` Stopping for efficacy boundary
- `x1` First-stage outcome
- `n2` Resulting second-stage sample size after observing `x1`
- `c2` Resulting second-stage decision-boundary after observing `x1`
- `x2` Second-stage outcome
- `reject` Decision whether the null hypothesis is rejected or not

**See Also**

[TwoStageDesign](#)

**Examples**

```
design <- TwoStageDesign(25, 0, 2, 25, 2, order = 5)
# draw samples assuming two-armed design
simulate(design, 10, Normal(), .3, 42)
```

---

Student-class

*Student's t data distribution*

---

**Description**

Implements exact t-distributions instead of a normal approximation

**Usage**

```
Student(two_armed = TRUE)

## S4 method for signature 'Student'
quantile(x, probs, n, theta, ...)

## S4 method for signature 'Student,numeric'
simulate(object, nsim, n, theta, seed = NULL, ...)
```

**Arguments**

two_armed	logical indicating if a two-armed trial is regarded
x	outcome
probs	vector of probabilities
n	sample size
theta	distribution parameter
...	further optional arguments
object	object of class Student
nsim	number of simulation runs
seed	random seed

**See Also**

see [probability\\_density\\_function](#) and [cumulative\\_distribution\\_function](#) to evaluate the pdf and the cdf, respectively.

**Examples**

```
datadist <- Student(two_armed = TRUE)
```

---

subject\_to                      *Create a collection of constraints*

---

**Description**

subject\_to(...) can be used to generate an object of class ConstraintsCollection from an arbitrary number of (un)conditional constraints.

**Usage**

```
subject_to(...)

## S4 method for signature 'ConstraintsCollection,TwoStageDesign'
evaluate(s, design, optimization = FALSE, ...)
```

**Arguments**

...	either constraint objects (for subject_to or optional arguments passed to evaluate)
s	object of class ConstraintCollection
design	object
optimization	logical, if TRUE uses a relaxation to real parameters of the underlying design; used for smooth optimization.

**Value**

an object of class ConstraintsCollection

**See Also**

subject\_to is intended to be used for constraint specification the constraints in [minimize](#).

**Examples**

```
# define type one error rate and power
toer <- Power(Normal(), PointMassPrior(0.0, 1))
power <- Power(Normal(), PointMassPrior(0.4, 1))

# create constrain collection
subject_to(
  toer <= 0.025,
  power >= 0.9
)
```

---

tunable\_parameters      *Switch between numeric and S4 class representation of a design*

---

**Description**

Get tunable parameters of a design as numeric vector via `tunable_parameters` or update a design object with a suitable vector of values for its tunable parameters.

**Usage**

```
tunable_parameters(object, ...)

## S4 method for signature 'TwoStageDesign'
tunable_parameters(object, ...)

## S4 method for signature 'TwoStageDesign'
update(object, params, ...)

## S4 method for signature 'OneStageDesign'
update(object, params, ...)
```

**Arguments**

object	TwoStageDesign object to update
...	further optional arguments
params	vector of design parameters, must be in same order as returned by <code>tunable_parameters</code>



**Details**

The tunable slot of a `TwoStageDesign` stores information about the set of design parameters which are considered fixed (not changed during optimization) or tunable (changed during optimization). For details on how to fix certain parameters or how to make them tunable again, see `make_fixed` and `make_tunable`.

**See Also**

`TwoStageDesign`

**Examples**

```
design <- TwoStageDesign(25, 0, 2, 25, 2, order = 5)
tunable_parameters(design)
design2 <- update(design, tunable_parameters(design) + 1)
tunable_parameters(design2)
```

---

TwoStageDesign-class *Two-stage designs*

---

**Description**

`TwoStageDesign` is the fundamental design class of the `adoptr` package. Formally, we represent a generic two-stage design as a five-tuple  $(n_1, c_1^f, c_1^e, n_2(\cdot), c_2(\cdot))$ . Here,  $n_1$  is the first-stage sample size (per group),  $c_1^f$  and  $c_1^e$  are boundaries for early stopping for futility and efficacy, respectively. Since the trial design is a two-stage design, the elements  $n_2(\cdot)$  (stage-two sample size) and  $c_2(\cdot)$  (stage-two critical value) are functions of the first-stage outcome  $X_1 = x_1$ .  $X_1$  denotes the first-stage test statistic. A brief description on this definition of two-stage designs can be read [here](#). For available methods, see the 'See Also' section at the end of this page.

**Usage**

```
TwoStageDesign(n1, ...)

## S4 method for signature 'numeric'
TwoStageDesign(n1, c1f, c1e, n2_pivots, c2_pivots, order = NULL, ...)

## S4 method for signature 'TwoStageDesign'
summary(object, ..., rounded = TRUE)
```

**Arguments**

<code>n1</code>	stage-one sample size
<code>...</code>	further optional arguments
<code>c1f</code>	early futility stopping boundary

<code>c1e</code>	early efficacy stopping boundary
<code>n2_pivots</code>	numeric vector, stage-two sample size on the integration pivot points
<code>c2_pivots</code>	numeric vector, stage-two critical values on the integration pivot points
<code>order</code>	integer, integration order of the employed Gaussian quadrature integration rule to evaluate scores. Automatically set to <code>length(n2_pivots)</code> if <code>length(n2_pivots) == length(c2_pivots) &gt; 1</code> , otherwise <code>c2</code> and <code>n2</code> are taken to be constant in stage-two and replicated to match the number of pivots specified by <code>order</code>
<code>object</code>	object to show
<code>rounded</code>	should rounded n-values be used?

### Details

summary can be used to quickly compute and display basic facts about a TwoStageDesign. An arbitrary number of names `UnconditionalScore` objects can be provided via the optional arguments `...` and are included in the summary displayed using `print`.

### Slots

<code>n1</code>	cf. parameter 'n1'
<code>c1f</code>	cf. parameter 'c1f'
<code>c1e</code>	cf. parameter 'c1e'
<code>n2_pivots</code>	vector of length 'order' giving the values of <code>n2</code> at the pivot points of the numeric integration rule
<code>c2_pivots</code>	vector of length <code>order</code> giving the values of <code>c2</code> at the pivot points of the numeric integration rule
<code>x1_norm_pivots</code>	normalized pivots for integration rule (in <code>[-1, 1]</code> ) the actual pivots are scaled to the interval <code>[c1f, c1e]</code> and can be obtained by the internal method <code>adoptr:::scaled_integration_pivots(design)</code>
<code>weights</code>	weights of of integration rule at <code>x1_norm_pivots</code> for approximating integrals over <code>x1</code>
<code>tunable</code>	named logical vector indicating whether corresponding slot is considered a tunable parameter (i.e. whether it can be changed during optimization via <code>minimize</code> or not; cf. <code>make_fixed</code> )

### See Also

For accessing sample sizes and critical values safely, see methods in `n` and `c2`; for modifying behaviour during optimization see `make_tunable`; to convert between S4 class representation and numeric vector, see `tunable_parameters`; for simulating from a given design, see `simulate`; for plotting see `plot, TwoStageDesign-method`. Both `group-sequential` and `one-stage designs` (!) are implemented as subclasses of `TwoStageDesign`.

**Examples**

```
design <- TwoStageDesign(50, 0, 2, 50.0, 2.0, 5)
pow    <- Power(Normal(), PointMassPrior(.4, 1))
summary(design, "Power" = pow)
```

# Index

- $\leq$ , ConditionalScore, ConditionalScore-method (Constraints), [12](#)
  - $\leq$ , ConditionalScore, numeric-method (Constraints), [12](#)
  - $\leq$ , UnconditionalScore, UnconditionalScore-method (Constraints), [12](#)
  - $\leq$ , UnconditionalScore, numeric-method (Constraints), [12](#)
  - $\leq$ , numeric, ConditionalScore-method (Constraints), [12](#)
  - $\leq$ , numeric, UnconditionalScore-method (Constraints), [12](#)
  - $\geq$ , ConditionalScore, ConditionalScore-method (Constraints), [12](#)
  - $\geq$ , ConditionalScore, numeric-method (Constraints), [12](#)
  - $\geq$ , UnconditionalScore, UnconditionalScore-method (Constraints), [12](#)
  - $\geq$ , UnconditionalScore, numeric-method (Constraints), [12](#)
  - $\geq$ , numeric, ConditionalScore-method (Constraints), [12](#)
  - $\geq$ , numeric, UnconditionalScore-method (Constraints), [12](#)
- adoptr, [3](#), [41](#)
- AverageN2, [4](#), [26](#)
- AverageN2 (AverageN2-class), [3](#)
- AverageN2-class, [3](#)
- Binomial, [15](#), [35](#)
- Binomial (Binomial-class), [4](#)
- Binomial-class, [4](#)
- bounds, [6](#), [34](#)
- bounds, ContinuousPrior-method (bounds), [6](#)
- bounds, PointMassPrior-method (bounds), [6](#)
- c2, [6](#), [25](#), [42](#)
- c2, OneStageDesign, numeric-method (c2), [6](#)
- c2, TwoStageDesign, numeric-method (c2), [6](#)
- composite, [8](#), [36](#)
- condition, [9](#), [34](#)
- condition, ContinuousPrior, numeric-method (condition), [9](#)
- condition, PointMassPrior, numeric-method (condition), [9](#)
- ConditionalPower, [3](#), [36](#)
- ConditionalPower (ConditionalPower-class), [10](#)
- ConditionalPower-class, [10](#)
- ConditionalSampleSize, [3](#), [23](#), [36](#)
- ConditionalSampleSize (ConditionalSampleSize-class), [11](#)
- ConditionalSampleSize-class, [11](#)
- ConditionalScore, [29](#)
- ConstraintCollection (subject\_to), [39](#)
- Constraints, [12](#)
- ContinuousPrior, [3](#), [30](#), [34](#)
- ContinuousPrior (ContinuousPrior-class), [13](#)
- ContinuousPrior-class, [13](#)
- cumulative\_distribution\_function, [5](#), [15](#), [28](#), [39](#)
- cumulative\_distribution\_function, Binomial, numeric, numeric, (cumulative\_distribution\_function), [15](#)
- cumulative\_distribution\_function, Normal, numeric, numeric, (cumulative\_distribution\_function), [15](#)
- cumulative\_distribution\_function, Student, numeric, numeric, (cumulative\_distribution\_function), [15](#)
- DataDistribution, [3](#), [32–34](#), [36](#)
- DataDistribution (DataDistribution-class), [16](#)
- DataDistribution-class, [16](#)
- distribution, [6](#), [9–11](#), [15](#), [17](#), [31–33](#), [35](#)

- evaluate (Scores), 36
- evaluate, AverageN2, TwoStageDesign-method  
(AverageN2-class), 3
- evaluate, CompositeScore, TwoStageDesign-method  
(composite), 8
- evaluate, ConditionalPower, TwoStageDesign-method  
(ConditionalPower-class), 10
- evaluate, ConditionalSampleSize, TwoStageDesign-method  
(ConditionalSampleSize-class), 11
- evaluate, Constraint, TwoStageDesign-method  
(Constraints), 12
- evaluate, ConstraintsCollection, TwoStageDesign-method  
(subject\_to), 39
- evaluate, IntegralScore, TwoStageDesign-method  
(Scores), 36
- evaluate, MaximumSampleSize, TwoStageDesign-method  
(MaximumSampleSize-class), 22
- evaluate, N1, TwoStageDesign-method  
(N1-class), 26
- expectation, 16, 34
- expectation, ContinuousPrior, function-method  
(expectation), 16
- expectation, PointMassPrior, function-method  
(expectation), 16
- expected (Scores), 36
- expected, ConditionalScore-method  
(Scores), 36
- ExpectedSampleSize, 3
- ExpectedSampleSize  
(ConditionalSampleSize-class), 11
- get\_initial\_design, 17
- get\_lower\_boundary\_design, 18
- get\_lower\_boundary\_design, GroupSequentialDesign-method  
(get\_lower\_boundary\_design), 18
- get\_lower\_boundary\_design, OneStageDesign-method  
(get\_lower\_boundary\_design), 18
- get\_lower\_boundary\_design, TwoStageDesign-method  
(get\_lower\_boundary\_design), 18
- get\_upper\_boundary\_design  
(get\_lower\_boundary\_design), 18
- get\_upper\_boundary\_design, GroupSequentialDesign-method  
(get\_lower\_boundary\_design), 18
- get\_upper\_boundary\_design, OneStageDesign-method  
(get\_lower\_boundary\_design), 18
- get\_upper\_boundary\_design, TwoStageDesign-method  
(get\_lower\_boundary\_design), 18
- group-sequential, 42
- GroupSequentialDesign, 3, 29
- GroupSequentialDesign  
(GroupSequentialDesign-class), 20
- GroupSequentialDesign-class, 20
- make\_fixed, 41, 42
- make\_fixed (make\_tunable), 21
- make\_fixed, TwoStageDesign-method  
(make\_tunable), 21
- make\_tunable, 21, 41, 42
- make\_tunable, TwoStageDesign-method  
(make\_tunable), 21
- MaximumSampleSize  
(MaximumSampleSize-class), 22
- MaximumSampleSize-class, 22
- minimize, 13, 17, 18, 22, 23, 25, 26, 40, 42
- n, 7, 42
- n (n1), 25
- n, TwoStageDesign, numeric-method (n1), 25
- N1, 4, 26
- N1 (N1-class), 26
- n1, 25
- n1, TwoStageDesign-method (n1), 25
- N1-class, 26
- n2 (n1), 25
- n2, GroupSequentialDesign, numeric-method  
(n1), 25
- n2, OneStageDesign, numeric-method (n1), 25
- n2, TwoStageDesign, numeric-method (n1), 25
- nloptr, 24
- norm-method, 15, 35
- Normal (Normal-class), 27
- Normal-class, 27
- one-stage designs, 42
- OneStageDesign, 3
- OneStageDesign (OneStageDesign-class), 28
- OneStageDesign-class, 28
- plot, OneStageDesign-method  
(OneStageDesign-class), 28
- plot, TwoStageDesign-method, 29
- PointMassPrior, 3, 14, 34

- PointMassPrior (PointMassPrior-class), 30
- PointMassPrior-class, 30
- posterior, 31, 34
- posterior, DataDistribution, ContinuousPrior, numeric-method (posterior), 31
- posterior, DataDistribution, PointMassPrior, numeric-method (posterior), 31
- Power, 3
- Power (ConditionalPower-class), 10
- predictive\_cdf, 32, 34
- predictive\_cdf, DataDistribution, ContinuousPrior, numeric-method (predictive\_cdf), 32
- predictive\_cdf, DataDistribution, PointMassPrior, numeric-method (predictive\_cdf), 32
- predictive\_pdf, 33, 34
- predictive\_pdf, DataDistribution, ContinuousPrior, numeric-method (predictive\_pdf), 33
- predictive\_pdf, DataDistribution, PointMassPrior, numeric-method (predictive\_pdf), 33
- print, 42
- print (print.adoptrOptimizationResult), 34
- print.adoptrOptimizationResult, 34
- Prior, 9–11, 13, 30–33, 36
- Prior (Prior-class), 34
- Prior-class, 34
- probability\_density\_function, 5, 28, 35, 39
- probability\_density\_function, Binomial, numeric-method (probability\_density\_function), 35
- probability\_density\_function, Normal, numeric-method (probability\_density\_function), 35
- probability\_density\_function, Student, numeric-method (probability\_density\_function), 35
- quantile, Binomial-method (Binomial-class), 4
- quantile, Normal-method (Normal-class), 27
- quantile, Student-method (Student-class), 38
- Score, 4, 10, 11, 13, 23, 26, 36
- Scores, 3, 8, 10, 11, 23, 36
- simulate, 42
- simulate, Binomial, numeric-method (Binomial-class), 4
- simulate, Normal, numeric-method (Normal-class), 27
- simulate, Student, numeric-method (Student-class), 38
- simulate, TwoStageDesign, numeric-method, 37
- Student (Student-class), 38
- Student-class, 38
- subject\_to, 39
- summary, TwoStageDesign-method (TwoStageDesign-class), 41
- tunable\_parameters, 22, 40, 42
- tunable\_parameters, TwoStageDesign-method (tunable\_parameters), 40
- TwoStageDesign, 3, 6, 7, 20–22, 25, 28, 29, 37, 38, 41
- TwoStageDesign (TwoStageDesign-class), 41
- TwoStageDesign, GroupSequentialDesign-method (GroupSequentialDesign-class), 20
- TwoStageDesign, numeric-method (TwoStageDesign-class), 41
- TwoStageDesign, OneStageDesign-method (OneStageDesign-class), 28
- TwoStageDesign-class, 41
- UnconditionalScore, 42
- update, OneStageDesign-method (tunable\_parameters), 40
- update, TwoStageDesign-method (tunable\_parameters), 40