

Package: quicopt (via r-universe)

October 15, 2026

Title Client for the 'Quicopt' Optimization Service

Version 0.7.0

Description Write an optimization model in R and solve it with the 'Quicopt' service <<https://quicopt.com>>: state the decisions, the objective and the constraints, and the service finds the best decision. The model is sent to the service, so no solver needs to be installed. Uncertain data, such as demand or prices, is described by probability distributions or taken from the rows of a data frame, and the model can then minimize an expected cost, guard against bad outcomes, or require a condition to hold with a given probability. Decisions that are an ordering or an assignment, such as a delivery route, can be stated directly.

License Apache License (== 2.0)

Encoding UTF-8

Imports curl, jsonlite, utils

Suggests knitr, rmarkdown

VignetteBuilder knitr

Depends R (>= 4.1)

URL <https://quicopt.github.io/quicopt-r/>,
<https://github.com/Quicopt/quicopt-r>

BugReports <https://github.com/Quicopt/quicopt-r/issues>

Roxygen list(markdown = TRUE)

Config/roxygen2/version 8.1.0

Config/pak/sysreqs libssl-dev

Repository <https://quicopt.r-universe.dev>

Date/Publication 2026-10-15 09:46:08 UTC

RemoteUrl <https://github.com/Quicopt/quicopt-r>

RemoteRef HEAD

RemoteSha c0c9a46370edbc5cf52ed4633ba644bebbbed509e

Contents

active	3
add	4
as_program	5
block	6
consets	7
constraint	8
CONTINUOUS	9
cvar	10
DEFAULT_BASE_URL	11
distribution	11
empirical	13
encode	14
encode_params	14
evaluate	15
expectation	16
expressions	17
holds	18
index_set	19
ir	20
is_random	22
item_at	23
job_status	24
lookup_table	25
minimize	26
model	27
num_var	28
parametric	30
part_of	31
part_var	32
partition_decl	33
perm_var	34
permutation_decl	35
precede	36
prob	37
program	38
rand_var	39
resample	40
scenario_max	41
set_distribution	42
set_empirical	43
set_scenarios	44
set_start	45
solve_model	46
stochastic	48
submit	49
summaries	51

<i>active</i>	3
variance	52
wire	53
Index	54

<i>active</i>	<i>Whether a variable is in use</i>
---------------	-------------------------------------

Description

`active(x)` is 1 when the variable `x` is not 0, and 0 when it is. It is meant for a variable declared with `semi = TRUE` (see `num_var()`), which is either 0 or between its bounds: a delivery of either nothing or 10 to 25 units, say. A fee charged for every delivery that takes place is then `fee * active(delivery)`, and a limit on how many deliveries take place is a constraint on `sum(active(delivery))`.

Usage

`active(x)`

Arguments

`x` A decision variable, or elements of one, such as `x[2:3]`.

Details

For such a variable, a model that is otherwise linear is still solved to a proven optimum. On any other variable, `active(x)` means the same, 1 when `x` is not 0, but the model is then solved by a search.

Value

An expression with one 0/1 element per element of `x`.

Examples

```
m <- model()
delivery <- int_var(m, "delivery", lower = 10, upper = 25, length = 3, semi = TRUE)
fee <- c(30, 10, 5)
minimize(m, sum(c(1, 2, 3) * delivery) + sum(fee * active(delivery)))
add(m, sum(delivery) >= 40) # the customer needs 40 units
add(m, sum(active(delivery)) <= 2) # at most two deliveries
```

add	<i>Add constraints to a model</i>
-----	-----------------------------------

Description

A comparison of expressions, written with `<=`, `>=` or `==`, becomes a requirement that every solution must meet: `add(m, tables + chairs <= 18)`. Here the comparison is not a test that returns `TRUE` or `FALSE`. A comparison of two vectors adds one constraint per element: with `x` and `cap` of length 3, `add(m, x <= cap)` adds 3 constraints.

Usage

```
add(m, rel, margin = 0, when = NULL)
```

Arguments

<code>m</code>	A <code>model()</code> .
<code>rel</code>	A comparison of expressions, written with <code><=</code> , <code>>=</code> or <code>==</code> .
<code>margin</code>	For a chance constraint: by how many standard errors to tighten the target. The default, 0, leaves it as written.
<code>when</code>	A binary variable that switches the constraint on.

Details

`<` and `>` are not accepted, because for a quantity that can take any value they mean the same as `<=` and `>=`. `!=` is not accepted either; `holds()` turns it into a 0/1 expression, which can be used instead.

In a model with random variables, each side of a constraint has to be one number, not one per scenario: summarize it first, for example with `expectation()` or `prob()`. A requirement on a probability, such as `add(m, prob(demand <= stock) >= 0.9)`, is called a *chance constraint*.

Value

The model, invisibly.

A safety margin on a chance constraint

The probability in a chance constraint is estimated from the scenarios, so it carries sampling error. For a target `p` and `n_scenarios` scenarios, its standard error is $\sqrt{p * (1 - p) / n_scenarios}$, about 0.013 for `p = 0.9` and `n_scenarios = 512`. A solution chosen to just meet the target on its own scenarios therefore misses the target on new scenarios about half the time (see `resample()`).

`margin = k` raises the target by `k` standard errors, so that the true probability meets the original target with a confidence of about `pnorm(k)`: 84% for `k = 1`, 98% for `k = 2`. For an upper limit, such as `prob(...) <= 0.1`, the target is lowered instead. A margin applies only when one side of the constraint is a `prob()` and the other a number strictly between 0 and 1. The standard error is computed from the number of scenarios when the model is solved, so the margin can be given before `set_scenarios()` is called.

A constraint with an on-off switch

when = b, with b a binary variable from `bin_var()`, makes the constraint apply only in solutions in which b is 1. For example, `add(m, output <= 0, when = closed)` forces the output to 0 only if the plant is closed. b is a single variable, or a vector variable with one element per constraint.

A model with a switch is solved by a search, so its status is "heuristic". The same holds for `holds()`, `max()` and `min()`. When every variable of such a model is an integer or a binary variable, each needs finite bounds; a model with a continuous variable or with random variables has no such limit.

Examples

```
m <- model()
tables <- num_var(m, "tables", lower = 0)
chairs <- num_var(m, "chairs", lower = 0)
add(m, 3 * tables + chairs <= 41)           # hours of carpentry
add(m, tables + chairs <= 18)              # wood

# a chance constraint, without and with a safety margin
shop <- model()
stock <- num_var(shop, "stock", lower = 0, upper = 200)
demand <- rand_var(shop, "demand", normal(100, 15))
set_scenarios(shop, 512, seed = 42)
add(shop, prob(demand <= stock) >= 0.9)     # demand met on 90% of scenarios
add(shop, prob(demand <= stock) >= 0.9, margin = 2) # the target raised to about 0.927
```

as_program

Convert a model to a program

Description

A program is the model written out as plain R lists: every variable, expression and constraint, in the form that `encode()` turns into the bytes sent to the service. `solve()` makes this conversion itself, so you only need `as_program()` to look at exactly what a model sends, or to work with the `program()` functions directly.

Usage

```
as_program(m)
```

Arguments

m A `model()`.

Value

A `program()`.

Examples

```
m <- model()
x <- num_var(m, "x", lower = 0, upper = 4)
maximize(m, 3 * x)
str(as_program(m)$vars)
```

block

Group the variables of a subproblem

Description

Many models are made of small subproblems of the same kind: one per customer to supply, per project to staff, per product to make. Each has a few variables and constraints of its own, such as the customer's demand, and the subproblems interact mostly through the objective, for example through a warehouse's opening fee that several customers share.

Usage

```
block(m, name, vars)
```

Arguments

m	A <code>model()</code> .
name	The block's name, unique among the model's blocks.
vars	The block's variables: a decision variable, elements of one (<code>x[1:3]</code>), or several combined with <code>c()</code> .

Details

`block()` declares the variables of one such subproblem. A constraint whose variables all belong to one block, such as one customer's demand, is then that block's own constraint, and it holds in every solution the service returns. Declaring the blocks of a model changes nothing else about it. A constraint that involves variables of several blocks, such as a person's working hours over several projects, holds in a solution that is reported feasible.

A variable can be in at most one block, and a variable in a block needs finite bounds.

Value

The model, invisibly.

Examples

```
# Three warehouses supply three customers. Each delivery is none or 10 to 25
# units and costs 5 plus a price per unit; a warehouse that delivers anything
# pays its opening fee.
demand <- c(40, 30, 20) # units, customers 1 to 3
price <- rbind(c(1, 3, 4), c(3, 1, 3), c(4, 2, 1)) # per unit, price[warehouse, customer]
opening <- c(60, 50, 40) # fee of warehouses 1 to 3
m <- model()
ship <- int_var(m, "ship", lower = 10, upper = 25, length = 9, semi = TRUE) # none, or 10 to 25
to <- function(c) ship[3 * (c - 1) + 1:3] # to customer c, from warehouses 1 to 3
from <- function(w) ship[w + c(0, 3, 6)] # from warehouse w, to customers 1 to 3
for (c in 1:3) {
  block(m, paste0("customer_", c), to(c))
  add(m, sum(to(c)) >= demand[c])
}
sent <- c(sum(from(1)), sum(from(2)), sum(from(3))) # units leaving each warehouse
minimize(m, sum(as.vector(price) * ship) # price per unit
              + 5 * sum(active(ship)) # 5 per delivery
              + sum(opening * holds(sent >= 1))) # a warehouse that ships pays its fee

## Not run:
res <- solve(m)
res$objective # 300
res$solution # warehouse 2 delivers nothing
# ship[1] ship[2] ship[3] ship[4] ship[5] ship[6] ship[7] ship[8] ship[9]
#      25      0      15      10      0      20      0      0      20

## End(Not run)
```

consets

Constraint sets, for building a program by hand

Description

In a `program()`, a constraint states that an expression lies in a set:

Usage

```
zero()
```

```
nonneg()
```

```
indicator(bin, inner)
```

Arguments

<code>bin</code>	The binary variable that switches the constraint on, as an <code>ir_var()</code> node.
<code>inner</code>	The set the expression must lie in when <code>bin</code> is 1: <code>zero()</code> or <code>nonneg()</code> .

Details

- `zero()`: the expression equals 0;
- `nonneg()`: the expression is at least 0;
- `indicator(bin, inner)`: the expression lies in the set `inner` whenever the binary variable `bin` is 1, and is unrestricted when it is 0.

So $x + 2 * y \leq 5$ is written as $5 - (x + 2 * y)$ in `nonneg()`, and $x == 3$ as $x - 3$ in `zero()`. `add()` makes this conversion for you.

Value

A plain list without a class, to be passed as the set of a `constraint()`: `list(kind = "zero")`, `list(kind = "nonneg")`, or for `indicator()` a list with `kind = "indicator"` and the fields `bin` and `inner`.

Examples

```
# x + 2 * y <= 5
constraint(ir_apply("-", list(ir_const(5),
                             ir_apply("+", list(ir_var("x"),
                                                  ir_apply("*", list(ir_const(2), ir_var("y"))))))),
          nonneg())
```

constraint

A constraint, for building a program by hand

Description

States that the expression `f` lies in set (see `zero()` and the other constraint sets). With `over`, the constraint is repeated for every element of one or more index sets, like a constraint written "for all `i` in `S`". `add()` builds these for you.

Usage

```
constraint(f, set, over = list())
```

Arguments

- | | |
|-------------------|---|
| <code>f</code> | The expression node. |
| <code>set</code> | The set <code>f</code> must lie in: <code>zero()</code> , <code>nonneg()</code> or <code>indicator()</code> . |
| <code>over</code> | A list of <code>list(idx, set_ref)</code> pairs, each repeating the constraint for every element of the set <code>set_ref</code> (from <code>ir_set_ref()</code>), with the index named <code>idx</code> standing for the element in <code>f</code> . <code>list()</code> for a single constraint. |

Value

A plain list without a class, with the fields `f`, `set` and `over`. It is an element of the list a `program()` takes as constraints.

Examples

```
# x <= 4, written as 4 - x >= 0
constraint(ir_apply("-", list(ir_const(4), ir_var("x"))), nonneg())
```

CONTINUOUS	<i>A decision variable, for building a program by hand</i>
------------	--

Description

Declares one decision variable of a `program()`. `num_var()`, `int_var()` and `bin_var()` build these for you.

Usage

```
CONTINUOUS

INTEGER

BINARY

SEMICONTINUOUS

SEMIINTEGER

var_decl(
  name,
  axes = character(),
  domain = CONTINUOUS,
  lower = -Inf,
  upper = Inf,
  start = 0
)
```

Arguments

name	The variable's name, used for it in the answer.
axes	The names of the index sets the variable is indexed over; <code>character()</code> for a single variable.
domain	CONTINUOUS, INTEGER, BINARY, SEMICONTINUOUS or SEMIINTEGER.
lower, upper	A number (<code>-Inf</code> or <code>Inf</code> for no bound), or the name of a parameter table, for a bound that differs from index to index.
start	The value the search starts from.

Details

A variable of domain SEMICONTINUOUS is either 0 or a value between its bounds, and one of domain SEMIINTEGER is either 0 or a whole number between them (see the `semi` argument of `num_var()`). Both need finite bounds.

Value

`var_decl()` returns a plain list without a class, with the fields `name`, `axes`, `domain`, `lower`, `upper` and `start`. It is an element of the list a `program()` takes as `vars`.

CONTINUOUS, INTEGER, BINARY, SEMICONTINUOUS and SEMIINTEGER are not functions but constants: the whole numbers 1 to 5, which stand for the five kinds of variable in domain.

Examples

```
var_decl("tables", domain = INTEGER, lower = 0)
```

cvar

The average over the worst scenarios

Description

`cvar(x, alpha)` is the average of `x` over the worst $1 - \alpha$ share of the scenarios, the ones in which `x` is largest. With `alpha = 0.95`, it is the average over the worst 5%. The measure is known as the *conditional value at risk*. Minimizing it asks for a decision that keeps the bad scenarios as good as possible, rather than the average one.

Usage

```
cvar(x, alpha)
```

Arguments

<code>x</code>	A random expression (see <code>is_random()</code>).
<code>alpha</code>	A number strictly between 0 and 1. It cannot depend on a decision.

Details

`x` is read as a cost: large values are bad. For a profit, use the negative, `cvar(-profit, 0.95)`.

Value

An expression of the same length as `x`, no longer random.

Examples

```

m <- model()
stock <- num_var(m, "stock", lower = 0, upper = 200)
demand <- rand_var(m, "demand", normal(100, 15))
set_scenarios(m, 512, seed = 42)
cost <- 3 * stock + 10 * max(demand - stock, 0)
minimize(m, cvar(cost, 0.95))          # the average cost of the worst 5% of scenarios

```

DEFAULT_BASE_URL	<i>The address of the public Quicopt service</i>
------------------	--

Description

`solve()` and `submit()` send models to this address unless they are given another `base_url`.

Usage

```
DEFAULT_BASE_URL
```

Value

Not a function but a constant: a character string, the address of the public service.

Examples

```
DEFAULT_BASE_URL
```

distribution	<i>Distributions for random variables</i>
--------------	---

Description

These functions describe how a random variable is distributed, for use in `rand_var()`. Each takes the same parameters, in the same order, as R's corresponding random number function:

Usage

```

distribution(head, ...)

normal(mean, sd)

uniform(min, max)

exponential(rate)

bernoulli(prob)

```

Arguments

head	The name under which the service knows the distribution.
...	The distribution's parameters, each a number, a numeric vector, or an expression without random variables.
mean, sd	The mean and the standard deviation, as in <code>rnorm()</code> .
min, max	The smallest and the largest value, as in <code>runif()</code> .
rate	The rate, as in <code>rexp()</code> ; the mean is $1 / \text{rate}$.
prob	The probability of a 1, as in <code>rbinom(n, 1, prob)</code> .

Details

- `normal(mean, sd)`, like `rnorm()`: the mean and the standard deviation.
- `uniform(min, max)`, like `runif()`: the smallest and the largest value.
- `exponential(rate)`, like `rexp()`: the rate, so that the mean is $1 / \text{rate}$.
- `bernoulli(prob)`, like `rbinom(n, 1, prob)`: 1 with probability `prob`, and 0 otherwise.

For any other distribution, draw a sample in R and pass it to `empirical()`. `distribution(head, ...)` names a distribution that the service offers but this package has no function for yet; the service refuses a name it does not know.

Value

A distribution, to be passed to `rand_var()` or `set_distribution()`.

Parameters that depend on a decision

A parameter may be an expression of decision variables rather than a number. The distribution then depends on the decision: a demand whose mean falls as the price rises, or a breakdown that becomes less likely the more is spent on maintenance. A parameter cannot contain a random variable.

A parameter given as a number is checked right away: a rate that is not positive, a probability outside 0 to 1, or a min above max is an error. A parameter given as an expression cannot be checked in advance, so give the decision variables in it bounds that keep it in range.

Vector parameters

A parameter may be a vector. The distribution then describes that many random variables, drawn independently: `normal(c(6, 5, 4), 1)` is three normal variables with means 6, 5 and 4 and standard deviation 1. Each parameter has length 1 or the common length.

Examples

```
m <- model()
lead_time <- rand_var(m, "lead_time", uniform(2, 5))      # between 2 and 5 days
gap <- rand_var(m, "gap", exponential(1 / 30))           # 30 minutes on average
fails <- rand_var(m, "fails", bernoulli(0.02))           # 1 with probability 0.02

# the more is spent on maintenance, the less likely a breakdown
spend <- num_var(m, "spend", lower = 0, upper = 10)
breaks <- rand_var(m, "breaks", bernoulli(0.2 - 0.015 * spend))
```

empirical	<i>A random variable given by a sample</i>
-----------	--

Description

`empirical(x)` describes a random variable by a sample of its values, one per scenario, instead of by a distribution. Pass it to `rand_var()`, as in `rand_var(m, "demand", empirical(x))`. The length of the sample sets the number of scenarios.

Usage

```
empirical(data)
```

Arguments

data	The sample: a numeric vector without NA, one value per scenario.
------	--

Details

This is how to use a distribution the package has no function for: draw a sample in R, for example with `rlnorm()`, and pass it to `empirical()`.

The samples of a model are read side by side, the i -th value of each in scenario i , so values observed together stay together. `set_empirical()` does this for every column of a data frame.

A sample is fixed data. If it was drawn in R, R's `set.seed()` determines it, not the model's seed, and `resample()` does not draw it again.

Value

An object of class `quicopt_empirical`, for `rand_var()` or `set_distribution()`. It can also be an element of the sources of a `program()`.

Examples

```
set.seed(1)
m <- model()
demand <- rand_var(m, "demand", empirical(rlnorm(500, log(100), 0.3))) # lognormal
```

encode	<i>Encode a model as the bytes the service reads</i>
--------	--

Description

Turns a model or a program into the bytes that `solve()` sends. The same program always gives the same bytes, in whatever order its parts were built. `solve()` also accepts the bytes directly.

Usage

```
encode(prog)
```

Arguments

prog A `model()` or a `program()`.

Value

A raw vector.

Examples

```
m <- model()
x <- num_var(m, "x", lower = 0, upper = 4)
maximize(m, 3 * x)
bytes <- encode(m)
length(bytes)
identical(bytes, encode(as_program(m))) # TRUE
```

encode_params	<i>Encode parameter tables on their own</i>
---------------	---

Description

Encodes the parameter tables of a `program()` without the rest of it. The format has this message so that a program can be given new data without sending the whole program again; no function in this package sends it. Tables are written in a fixed order, so the same data always gives the same bytes.

Usage

```
encode_params(params)
```

Arguments

params Parameter tables, as for `program()`.

Value

A raw vector.

Examples

```
cost <- list(list(key = list(1L), value = 3), list(key = list(2L), value = 1.5))
encode_params(list(cost = cost))
```

evaluate	<i>Compute a quantity at a given solution</i>
----------	---

Description

A solve reports the values of the decision variables and of the objective. `evaluate()` computes any other quantity of the model at a given solution: the share of scenarios in which demand is met, the average shortfall, the cost of a plan written out by hand. It sends the model to the service with every decision variable fixed at the solution's value, so each call is one request.

Usage

```
evaluate(m, solution, expr, seed = NULL, n_scenarios = NULL, ..., scenarios)
```

Arguments

<code>m</code>	A <code>model()</code> .
<code>solution</code>	A result from <code>solve()</code> , or a named numeric vector with a value for every decision variable, named as in a solution: "x", or "x[1]", "x[2]", ... for a vector variable. A model with a permutation needs a result, because only a result holds the arrangement.
<code>expr</code>	The quantity to compute: one expression that is not random.
<code>seed</code>	Left NULL, the model's own scenarios are used. Otherwise, the seed for new scenarios, different from the model's own.
<code>n_scenarios</code>	With seed: how many new scenarios to draw. Left NULL, as many as the model has. More scenarios give a more precise value.
<code>...</code>	Settings for the request, as for <code>solve_model()</code> : <code>base_url</code> , <code>api_key</code> , <code>project</code> , <code>config</code> , <code>gzip</code> , <code>timeout</code> and <code>transport</code> .
<code>scenarios</code>	The former name of <code>n_scenarios</code> , still accepted for now, with a warning.

Details

The quantity must be a single number, so a random expression is summarized first, for example with `prob()` or `expectation()`, as for an objective.

Without a seed, the value is computed on the model's own scenarios. With a seed, it is computed on new scenarios drawn from that seed, which shows how the solution does on scenarios it was not chosen for. `resample()` explains which random variables are drawn again.

Value

The value of `expr`, a number.

Examples

```
## Not run:
m <- model()
stock <- num_var(m, "stock", lower = 0, upper = 200)
demand <- rand_var(m, "demand", normal(100, 15))
set_scenarios(m, 512, seed = 42)
minimize(m, 3 * stock + 10 * expectation(max(demand - stock, 0)))
add(m, prob(demand <= stock) >= 0.9)
res <- solve(m)

met <- prob(demand <= stock)
evaluate(m, res, met) # on the model's scenarios
evaluate(m, res, met, seed = 7, n_scenarios = 1000) # on 1000 new ones
evaluate(m, c(stock = 110), met) # for a stock of 110

## End(Not run)
```

expectation

The average over the scenarios

Description

`expectation(x)` is the average of `x` over the model's scenarios: an estimate of its expected value. It turns a random expression into a single number, which can be used in the objective or in a constraint.

Usage

`expectation(x)`

Arguments

`x` A random expression (see [is_random\(\)](#)).

Details

Minimizing an average makes the typical scenario good, and says little about the bad ones; [cvar\(\)](#) looks at those instead. For a vector `x`, each element is averaged separately.

Value

An expression of the same length as `x`, no longer random.

Examples

```
m <- model()
stock <- num_var(m, "stock", lower = 0, upper = 200)
demand <- rand_var(m, "demand", normal(100, 15))
set_scenarios(m, 512, seed = 42)
shortfall <- max(demand - stock, 0)          # units short, in each scenario
minimize(m, 3 * stock + 10 * expectation(shortfall))
```

expressions

Expressions: arithmetic on variables

Description

The values of decision variables and random variables are not known when you write a model, so arithmetic on them cannot compute a number. Instead it builds an *expression*: a description of a quantity, which the service evaluates for every solution it considers. Expressions are written as ordinary R code, with these operators and functions:

Details

- `+`, `-`, `*`, `/` and `^`;
- `abs()`, `sqrt()`, `exp()`, `log()`, `sin()` and `cos()`;
- `sum()`, `prod()`, `max()`, `min()` and `mean()`.

Expressions need not be linear. Printing an expression shows what was built.

Expressions are vectors

A variable declared with `length = 10` is an expression of length 10, and expressions behave like numeric vectors in most ways: arithmetic works element by element, `x[3]` picks one element, `c()` combines expressions and numbers into a longer expression, and `length()` counts the elements.

There are two differences. First, the two sides of an arithmetic operation must have the same length, or one of them length 1. R would recycle the shorter vector, which in a model is almost always a mistake, so here it is an error. Second, `sum()`, `prod()`, `max()`, `min()` and `mean()` combine all the elements of all their arguments into one, as they do for numbers: `max(x, 0)` is the largest of all the elements of `x` and 0, not an element-by-element maximum, and there is no `pmax()`. The arguments can come in any order, so `max(0, x)` works too (see [summaries](#)). `mean(x)` is the mean over the elements of `x`; the mean over the scenarios is `expectation()`.

Comparisons

Comparing two expressions, as in `x <= y`, builds a comparison rather than returning TRUE or FALSE. What the comparison means depends on the function it is given to: `add()` makes it a constraint, `prob()` measures how often it holds across the scenarios, and `holds()` turns it into an expression that is 1 where it holds and 0 where it does not. `add()` and `prob()` accept `<=` and `>=`, and `add()` also `==`; `holds()` accepts all six comparisons.

What does not work

Functions not listed above, such as `round()`, `%%` and `log()` with a base, stop with an error. Base R functions that are not written for expressions do not work on them either: use `variance()` instead of `var()`, `holds()` instead of `ifelse()`, and `identical()` instead of `%in%` or `match()` to check whether two expressions are the same.

A model with `max()`, `min()` or `holds()` is solved by a search, so its status is "heuristic". When every variable of such a model is an integer or a binary variable, each needs finite bounds; a model with a continuous variable or with random variables has no such limit. A fixed fee for using a variable at all is the exception: written with `active()` on a variable declared with `semi = TRUE`, it keeps a linear model exact.

Examples

```
m <- model()
tables <- num_var(m, "tables", lower = 0)
chairs <- num_var(m, "chairs", lower = 0)
50 * tables + 20 * chairs           # an expression, not a number
tables + chairs <= 18                # a comparison, not TRUE or FALSE

x <- num_var(m, "x", lower = 0, upper = 1, length = 3)
2 * x                               # element by element
sum(x)                              # one expression
max(x, 0.5)                         # the largest of all four
```

holds	<i>A comparison as a 0/1 expression</i>
-------	---

Description

`holds(a <= b)` is 1 when the comparison is true and 0 when it is false. Unlike a constraint, it does not require anything: it is an expression like any other, so it can be added up, multiplied by a cost, or used in the objective. `sum(holds(x >= 1))`, for example, counts the elements of `x` that are at least 1. All six comparisons are accepted, including `<`, `>` and `!=`, which `add()` refuses.

Usage

```
holds(rel, tol = 0)
```

Arguments

- `rel` A comparison of expressions.
- `tol` How far the comparison may be off and still count as true; a number of at least 0.

Details

With random variables, `holds()` is evaluated in each scenario separately. `expectation(holds(demand <= stock))` is then the share of scenarios in which demand is met, the same number as `prob(demand <= stock)`. The difference is that `holds()` lets you combine the event with other quantities before averaging: for example, `expectation(200 * holds(demand > stock))` is the expected cost of a fixed charge of 200 whenever the stock runs out.

`tol` loosens the comparison. `holds(a == b, tol = 0.01)` is 1 when `a` and `b` are within 0.01 of each other, and `holds(a <= b, tol = 0.01)` is 1 when `a` is at most `b + 0.01`. Without it, `==` and `!=` compare exactly.

A model with `holds()` is solved by a search, so its status is "heuristic". The same holds for `max()` and `min()`. When every variable of such a model is an integer or a binary variable, each needs finite bounds; a model with a continuous variable or with random variables has no such limit.

Value

An expression with one 0/1 element per element of the comparison.

A fee due from a level

`holds()` writes a cost that is due once a total reaches a level, and a discount that is earned there. With deliveries `ship` from one warehouse to several customers, `60 * holds(sum(ship) >= 1)` is a fee of 60 that the warehouse costs once it ships anything, and `-30 * holds(sum(ship) >= 60)` a discount of 30 from 60 units on. `min(sum(active(ship)), 1)` is the same "anything shipped" written with `active()`: 1 once any delivery takes place. (`min()` and `max()` read the model expression from their first argument, so it comes before the number.)

Examples

```
m <- model()
stock <- num_var(m, "stock", lower = 0, upper = 200)
demand <- rand_var(m, "demand", normal(100, 15))
set_scenarios(m, 512, seed = 42)
stockout <- holds(demand > stock)           # 1 in the scenarios where the stock runs out
minimize(m, 3 * stock + 200 * expectation(stockout))

met <- holds(demand <= stock)
add(m, expectation(met) >= 0.9)           # the same as prob(demand <= stock) >= 0.9
```

Description

A named set of elements, for indexing variables (the axes of `var_decl()`), for reductions such as sums (`ir_reduce()`), and for repeating a constraint over its elements (the over of `constraint()`).

Usage

```
index_set(name, elements)
```

Arguments

name	The set's name.
elements	A list of whole numbers and strings.

Value

A plain list without a class, with the fields name and elements. It is an element of the list a [program\(\)](#) takes as sets.

Examples

```
index_set("items", list(1L, 2L, 3L))
```

ir	<i>Expression nodes, for building a program by hand</i>
----	---

Description

These functions build the pieces of a [program\(\)](#)'s expressions directly, as plain lists. A [model\(\)](#) builds them for you from ordinary R arithmetic, so you need them only to build a program by hand. Each node is a list whose kind field says what it is:

Usage

```
ir_const(value)

ir_param(name, index = list())

ir_var(name, index = list())

ir_apply(op, args)

ir_reduce(op, idx, over, body, cond = NULL)

ir_source_ref(name)

ir_struct_ref(name)

ir_table_ref(param, index)

ir_set_ref(name, args = list())
```

Arguments

value	A number.
name	The name of the variable, table, random variable, permutation, partition or index set referred to.
index	A position: a list of whole numbers and index names, <code>list()</code> for none. For <code>ir_table_ref()</code> , a list of one or two expression nodes.
op	The name of an operator the service knows, such as "+", "*" or "sqrt". For <code>ir_reduce()</code> , the operator that combines the terms, such as "+" for a sum.
args	For <code>ir_apply()</code> , a list of argument nodes. For <code>ir_set_ref()</code> , the indices the set depends on, <code>list()</code> for a set that depends on none.
idx	The name of the index that runs over the set, as used in body.
over	The index set to run over, from <code>ir_set_ref()</code> .
body	The expression to combine over the set.
cond	An expression node: only the terms in which it is not 0 are included. NULL includes every term.
param	The name of the parameter table.

Details

- `ir_const(value)`: a number.
- `ir_var(name, index)`: a decision variable.
- `ir_param(name, index)`: an entry of one of the program's parameter tables.
- `ir_source_ref(name)`: a random variable declared in the program's sources.
- `ir_apply(op, args)`: an operator, such as "+" or "sqrt", applied to a list of argument nodes.
- `ir_reduce(op, idx, over, body, cond)`: body combined over the elements of an index set, for example the sum of body over all i in a set S.
- `ir_set_ref(name, args)`: the index set a reduction runs over.
- `ir_struct_ref(name)`: a permutation or a partition declared in the program's structures. It may appear only as the second argument of the operators "item_at" and "slot_of" (a permutation) or "part_of" and "size_of" (a partition), which is what `item_at()`, `slot_of()`, `part_of()` and `size_of()` build.
- `ir_table_ref(param, index)`: an entry of a parameter table at positions given by expression nodes, which is what indexing a `lookup_table()` builds.

An index is a list of whole numbers, for fixed positions, and strings, for the names of indices bound by an enclosing reduction or constraint; it is `list()` for a variable or entry without an index.

Value

A plain list without a class. Every function except `ir_set_ref()` returns an expression node, which can stand wherever a program expects an expression: in another node's args, as the program's objective, or as the f of a `constraint()`. `ir_set_ref()` returns a reference to an index set instead, a list with the fields name and args and no kind, which is what `ir_reduce()` takes as over.

Examples

```
# 3 * x + 1
ir_apply("+", list(ir_apply("*", list(ir_const(3), ir_var("x"))), ir_const(1)))

# the sum of cost[i] * y[i] over the elements i of the set "items"
ir_reduce("+", "i", ir_set_ref("items"),
          ir_apply("*", list(ir_param("cost", list("i")), ir_var("y", list("i")))))
```

is_random

Does an expression vary across scenarios?

Description

An expression that contains a random variable has a different value in each scenario, and is called *random*. Summarizing it across the scenarios, for example with [expectation\(\)](#), gives a single number again: demand - stock is random, expectation(demand - stock) is not, and neither is 3 * stock.

Usage

```
is_random(x)
```

Arguments

x An expression, or a numeric vector (which is never random).

Details

An objective or a constraint must not be random, and a summary such as [expectation\(\)](#) needs a random expression to summarize. quicopt checks both rules itself and stops with an error when one is broken, so you need `is_random()` only in your own code, for example in a function that accepts both kinds of expression.

Value

A logical vector with one element per element of x.

Examples

```
m <- model()
stock <- num_var(m, "stock", lower = 0, upper = 200)
demand <- rand_var(m, "demand", normal(100, 15))
is_random(demand - stock)                      # TRUE
is_random(expectation(demand - stock))        # FALSE
is_random(c(stock, stock^2))                   # FALSE FALSE
```

item_at	<i>The item in a slot, and the slot of an item</i>
---------	--

Description

For a permutation P from `perm_var()`, `item_at(slot, P)` is the item in a slot, and `slot_of(item, P)` is the slot that an item is in. For a route whose items are stops and whose slots are the visits, `item_at(1, route)` is the first stop visited, and `slot_of(3, route)` is when stop 3 is visited.

Usage

```
item_at(slot, P)
```

```
slot_of(item, P)
```

Arguments

slot, item	Whole numbers from 1 to the size of the permutation.
P	A permutation from <code>perm_var()</code> .

Details

The result is an expression whose value the service chooses, like the value of a decision variable. To use it to read data, such as the distance between two stops, index a `lookup_table()` with it.

The first argument is a plain number, or a vector of numbers, and both functions return one element per number: `item_at(1:4, P)` is the items in the first four slots, an expression of length 4.

Value

An expression with one element per element of the first argument, each a whole number from 1 to the size of the permutation.

Examples

```
m <- model()
route <- perm_var(m, "route", 5)
item_at(1, route)           # the first stop visited
slot_of(3, route)           # when stop 3 is visited
item_at(1:4, route)         # the first four stops visited
```

job_status

Follow up a submitted job

Description

- `job_status()` returns the job's state, "queued", "running", "done" or "failed", together with the last lines of its log.
- `job_result()` returns the answer of the job, in the same form as `solve()`. By default it waits for the job to finish, checking every `poll` seconds for up to `timeout` seconds. Like `solve()`, it warns when the service granted less time than the job's `time_limit`.
- `job_log()` returns the job's log as text.
- `job_delete()` deletes the job and its stored answer from the service.

Usage

```
job_status(job)
```

```
job_result(job, wait = TRUE, timeout = NULL, poll = 0.5)
```

```
job_log(job)
```

```
job_delete(job)
```

Arguments

<code>job</code>	A job, as returned by <code>submit()</code> .
<code>wait</code>	TRUE waits until the job is finished. FALSE asks once, and is an error if the job is not finished yet.
<code>timeout</code>	How many seconds to wait at most: by default 120, or the job's <code>time_limit</code> plus 60 when that is longer.
<code>poll</code>	How many seconds to wait between two checks.

Value

`job_status()` returns the job's state as a list.

`job_result()` returns the answer, a list of class `quicopt_result` (see `solve()`).

`job_log()` returns the log as a single character string.

`job_delete()` returns NULL, invisibly.

Examples

```
## Not run:
job <- submit(m)
job_status(job)
res <- job_result(job)
job_delete(job)

## End(Not run)
```

lookup_table	<i>A table of data indexed by decisions</i>
--------------	---

Description

Sometimes a model needs a number from a table at a position that is not known yet: the distance between the first and the second stop of a route whose ordering the service is still choosing, or the price of an option that an integer variable picks. An ordinary R vector or matrix cannot be indexed this way. `lookup_table()` adds the data to the model under a name, and indexing the result with expressions, such as `item_at()` or an integer variable, builds an expression whose value is the entry at the positions the service chooses.

Usage

```
lookup_table(m, name, values)

## S3 method for class 'quicopt_table'
x[i, j]
```

Arguments

<code>m</code>	A <code>model()</code> .
<code>name</code>	The table's name, unique within the model.
<code>values</code>	A numeric vector or matrix without NA.
<code>x</code>	A lookup table.
<code>i, j</code>	Positions in the table: numbers, or expressions such as <code>item_at()</code> . <code>j</code> only for a table made from a matrix.

Details

A table made from a vector takes one index, and a table made from a matrix two. Each index may be a number, a vector of numbers, or an expression.

Unlike indexing an ordinary R matrix, the two indices are paired up element by element: `km[1:4, 2:5]` is a 4 x 4 block of a matrix `km`, but for a lookup table `dist`, `dist[1:4, 2:5]` has four elements, the entries `[1, 2]`, `[2, 3]`, `[3, 4]` and `[4, 5]`. This is what makes `dist[item_at(1:4, P), item_at(2:5, P)]` the four legs of a five-stop route.

The result is an ordinary expression: it can be multiplied by a cost, added up, or averaged with `expectation()`. An index that is not a whole number is rounded, and one outside the table is moved to the nearest end, so a continuous variable can be used as an index too.

A model with a lookup table is solved by a search, so its status is "heuristic". The whole table is sent with the model, so a very large table makes the request large.

Value

The table, to be indexed with `[`.
`x[i]` and `x[i, j]` return an expression with one element per position.

Examples

```
# the distances between five stops along a road
position <- c(0, 3, 1, 4, 2)
m <- model()
dist <- lookup_table(m, "dist", abs(outer(position, position, "-")))
route <- perm_var(m, "route", 5)
legs <- dist[item_at(1:4, route), item_at(2:5, route)] # the four legs of the route
minimize(m, sum(legs))

# the price of one of four options, picked by an integer variable
price <- lookup_table(m, "price", c(3, 1, 4, 1.5))
choice <- int_var(m, "choice", lower = 1, upper = 4)
price[choice]
```

minimize	<i>Set the objective</i>
----------	--------------------------

Description

The objective is the single number that the service makes as small (`minimize()`) or as large (`maximize()`) as possible. Calling either function again replaces the objective. An expression with several elements has to be combined into one first, for example with `sum()`. A model without an objective asks for any solution that meets the constraints.

Usage

```
minimize(m, e)

maximize(m, e)
```

Arguments

- m A `model()`.
- e The expression to minimize or maximize.

Details

In a model with random variables, the objective has to be one number, not one per scenario: summarize it first, for example with `expectation()` (the `stochastic` help page lists all the summaries).

Value

The model, invisibly.

Examples

```
m <- model()
tables <- num_var(m, "tables", lower = 0)
chairs <- num_var(m, "chairs", lower = 0)
maximize(m, 50 * tables + 20 * chairs) # the profit
```

model

Create an empty model

Description

A model collects everything the service needs to find the best decision: the decisions to be made, the objective, the constraints and, when some of the data is uncertain, the random variables and the number of scenarios. `model()` creates an empty one, and these functions fill it:

Usage

```
model()
```

Details

- decisions: `num_var()`, `int_var()`, `bin_var()`, `perm_var()` for an ordering and `part_var()` for a division into groups;
- subproblems: `block()`;
- uncertain data: `rand_var()` and `set_empirical()`;
- the objective: `minimize()` or `maximize()`;
- constraints: `add()`.

`solve()` then sends the model to the service. `m$name` retrieves a variable declared in `m` by its name, and printing a model shows a short summary of it.

Each of the functions above changes the model in place and returns it invisibly, so a model can also be written as a pipe:

```
m <- model() |>
  add_var("x", lower = 0, upper = 4) # m$x retrieves the variable
```

Unlike most R objects, a model is not copied when it is passed to a function. After `m2 <- m |> add(m$x <= 3)`, `m` has the constraint too, and `m2` and `m` are the same model.

Value

An empty model, an object of class quicopt_model.

Examples

```
m <- model()
tables <- num_var(m, "tables", lower = 0)
chairs <- num_var(m, "chairs", lower = 0)
maximize(m, 50 * tables + 20 * chairs)
add(m, tables + chairs <= 18)
m
```

num_var	<i>Declare decision variables</i>
---------	-----------------------------------

Description

A decision variable is a quantity the service chooses. `num_var()` declares one that can take any value between its bounds, `int_var()` one that must be a whole number, and `bin_var()` a yes-or-no decision, which is either 0 or 1.

Usage

```
num_var(
  m,
  name,
  lower = -Inf,
  upper = Inf,
  length = 1,
  start = 0,
  semi = FALSE,
  n
)

int_var(
  m,
  name,
  lower = -Inf,
  upper = Inf,
  length = 1,
  start = 0,
  semi = FALSE,
  n
)

bin_var(m, name, length = 1, start = 0, n)
```

```

add_var(
  m,
  name,
  lower = -Inf,
  upper = Inf,
  length = 1,
  start = 0,
  domain = c("num", "int", "bin"),
  semi = FALSE,
  n
)

```

Arguments

m	A <code>model()</code> .
name	The variable's name, used for it in the answer. It must be unique within the model.
lower, upper	The smallest and the largest value allowed. <code>-Inf</code> and <code>Inf</code> , the defaults, leave that side open.
length	How many variables to declare under this name.
start	The value the service's search starts from (see <code>set_start()</code>).
semi	Whether the variable may also be 0 when 0 lies outside its bounds (see above).
n	The former name of length, still accepted for now, with a warning.
domain	For <code>add_var()</code> : "num", "int" or "bin", to declare the variable as <code>num_var()</code> , <code>int_var()</code> or <code>bin_var()</code> would.

Details

Each returns the variable, for use in expressions such as `50 * tables + 20 * chairs`; `m$tables` retrieves it from the model too. `add_var()` declares a variable in the same way but returns the model, for use in a pipe (see `model()`).

With `semi = TRUE`, the variable is either 0 or between its bounds, never in between: a delivery of either nothing or 10 to 25 units is `int_var(m, "delivery", lower = 10, upper = 25, semi = TRUE)`. Solvers for mixed-integer programs call such a variable *semi-continuous*, or *semi-integer* when it must be a whole number. The bounds must then be finite. When 0 already lies between them, `semi` changes nothing. `active()` is 1 when such a variable is not 0, which is how a fixed fee per delivery is written: `fee * active(delivery)`.

With `length` greater than 1, one call declares that many variables under one name, and they behave like an R vector: `x[3]` is the third, arithmetic works element by element, and `sum(x)` adds them up. `lower`, `upper` and `start` may then be vectors with one value per element. In the answer the elements are named `"x[1]"`, `"x[2]"`, and so on; a single variable is named `"x"`.

Value

The variable, an expression with `length` elements.

`add_var()` returns the model, invisibly.

Examples

```
m <- model()
tables <- int_var(m, "tables", lower = 0)
take <- bin_var(m, "take", length = 6) # six yes-or-no decisions
share <- num_var(m, "share", lower = 0, upper = 1, length = 3)
sum(share)
take[2]
delivery <- int_var(m, "delivery", lower = 10, upper = 25, semi = TRUE) # none, or 10 to 25
```

parametric	<i>A random variable with a distribution, for building a program by hand</i>
------------	--

Description

Declares one random variable of a `program()` by its distribution. `head` names the distribution, such as "normal", and `params` holds its parameters as expression nodes, in the same order as for `normal()` and the other distribution functions. A parameter may contain decision variables, but no random variable. `rand_var()` builds these for you.

Usage

```
parametric(head, params)
```

Arguments

<code>head</code>	The name under which the service knows the distribution.
<code>params</code>	A list of expression nodes, one per parameter.

Value

A plain list without a class, with the fields `kind = "parametric"`, `head` and `params`. It is an element of the named list a `program()` takes as sources, and the element's name is the name `ir_source_ref()` refers to.

Examples

```
# demand ~ normal(100, 15)
sources <- list(demand = parametric("normal", list(ir_const(100), ir_const(15))))
```

part_of

*The part of an item, and the size of a part***Description**

For a partition P from `part_var()`, `part_of(item, P)` is the part an item is in, and `size_of(part, P)` is how many items a part holds. For people divided into teams, `part_of(3, teams)` is the team of person 3, and `size_of(1, teams)` is the number of people in team 1.

Usage

```
part_of(item, P)
```

```
size_of(part, P)
```

Arguments

item	Items: whole numbers from 1 to the number of items.
P	A partition from <code>part_var()</code> .
part	Parts: whole numbers from 1 to the number of parts.

Details

The result is an expression whose value the service chooses, like the value of a decision variable. To read data that depends on an item's part, index a `lookup_table()` with `part_of()`.

The first argument is a plain number, or a vector of numbers, and both functions return one element per number: `part_of(1:5, P)` is the parts of the first five items, an expression of length 5.

Value

An expression with one element per element of the first argument: for `part_of()` each a whole number from 1 to the number of parts, for `size_of()` each a whole number from 0 to the number of items.

Examples

```
m <- model()
teams <- part_var(m, "teams", 5, 2)
part_of(3, teams)           # the team of person 3
size_of(1, teams)          # how many people are in team 1
part_of(1:5, teams)        # everyone's team
```

part_var	<i>Divide items into groups as a decision</i>
----------	---

Description

Some decisions put things into groups: people into teams, jobs onto machines, parcels into vans. `part_var()` declares such a decision, a *partition*: each of `n_items` items goes into exactly one of `n_parts` groups, called *parts*. Items are numbered from 1 to `n_items` and parts from 1 to `n_parts`. A part may stay empty.

Usage

```
part_var(m, name, n_items, n_parts, start = NULL)
```

```
add_part_var(m, name, n_items, n_parts, start = NULL)
```

Arguments

<code>m</code>	A <code>model()</code> .
<code>name</code>	The partition's name, unique within the model.
<code>n_items</code>	How many items; at least 1.
<code>n_parts</code>	How many parts; at least 2.
<code>start</code>	The division the service's search starts from: <code>start[i]</code> is the part of item <code>i</code> , a whole number from 1 to <code>n_parts</code> . Left <code>NULL</code> , the service chooses one.

Details

`part_of()` gives the part an item is in, and `size_of()` how many items a part holds. Both are expressions whose values the service chooses, like the value of a decision variable. A requirement on the sizes, such as teams of two and three, is an ordinary constraint: `add(m, size_of(1, teams) == 2)`. Data that depends on an item's part, such as what a person costs in each team, is read from a `lookup_table()`: `cost[1:5, part_of(1:5, teams)]`. Whether an item is in a given part is `holds(part_of(i, teams) == t)`.

Value

The partition, for use with `part_of()` and `size_of()`.

`add_part_var()` returns the model, invisibly.

The answer

For each partition, the result of `solve()` holds `res$structures$<name>` with two integer vectors: `part_of`, the part of each item, and `size_of`, the number of items in each part. `set_start()`, `evaluate()` and `resample()` take the division from a result too.

A model with a partition is solved by a search, so its status is "heuristic".

`add_part_var()` declares the partition in the same way but returns the model, for use in a pipe; `m$name` then retrieves the partition.

Examples

```

# Six players of strengths 7, 5, 4, 3, 2 and 1 form a team of four and a
# team of two, as evenly matched as possible.
strength <- c(7, 5, 4, 3, 2, 1) # of players 1 to 6
m <- model()
teams <- part_var(m, "teams", 6, 2) # 6 players into teams 1 and 2
in_team <- function(t) holds(part_of(1:6, teams) == t) # 1 for each player in team t
add(m, size_of(1, teams) == 4) # team 1 has four players
minimize(m, max(sum(strength * in_team(1)), sum(strength * in_team(2))))
## Not run:
res <- solve(m)
res$objective # 11
res$structures$teams$part_of # 2 1 2 1 1 1: players 1 and 3 form team 2
res$structures$teams$size_of # 4 2

## End(Not run)

```

partition_decl

A partition, for building a program by hand

Description

Declares one partition of a `program()`: size items, each in one of parts parts (see `part_var()`). `part_var()` builds these for you.

Usage

```
partition_decl(size, parts, start = integer(), fixed = FALSE)
```

Arguments

size	How many items, at least 1.
parts	How many parts, at least 2.
start	The assignment the search starts from: <code>start[i]</code> is the part of item <code>i</code> , a whole number from 1 to parts. <code>integer()</code> leaves the start to the service.
fixed	Whether the partition is fixed at start, which must then be given. This is how <code>evaluate()</code> and <code>resample()</code> keep the assignment of a solution.

Value

A plain list without a class, with the fields `kind = "partition"`, `size`, `parts`, `start` and `fixed`. It is an element of the named list a `program()` takes as structures, and the element's name is the name `ir_struct_ref()` refers to.

Examples

```

# five people in two teams, people 1 to 3 starting in team 1
structures <- list(teams = partition_decl(5, 2, start = c(1, 1, 1, 2, 2)))

```

perm_var	<i>Declare an ordering as a decision</i>
----------	--

Description

Some decisions are an arrangement: an ordering, such as the stops of a courier's route or the jobs on a machine, or a one-to-one assignment, such as which department moves into which office. `perm_var()` declares such a decision, a *permutation*.

Usage

```
perm_var(m, name, n_items, start = NULL)
```

```
add_perm_var(m, name, n_items, start = NULL)
```

Arguments

<code>m</code>	A <code>model()</code> .
<code>name</code>	The permutation's name, unique within the model.
<code>n_items</code>	How many items, and so how many slots; at least 2.
<code>start</code>	The arrangement the service's search starts from: <code>start[i]</code> is the slot of item <code>i</code> , so <code>start</code> contains each of the numbers 1 to <code>n_items</code> once. Left <code>NULL</code> , item <code>i</code> starts in slot <code>i</code> .

Value

The permutation, for use with `item_at()`, `slot_of()` and `precede()`.

`add_perm_var()` returns the model, invisibly.

Items and slots

A permutation arranges things, called *items*, in numbered places, called *slots*, with exactly one item in each slot and one slot for each item. Items and slots are both numbered from 1 to `n_items`. What they stand for depends on the problem:

- for a route, the items are the stops and the slots are the visits: slot 1 is the first stop visited, slot 2 the second, and so on;
- for a machine, the items are the jobs and the slots are the positions in the queue;
- for an office plan, the items are the departments and the slots are the offices.

`item_at()` gives the item in a slot, and `slot_of()` the slot of an item. Both are expressions whose values the service chooses, like the value of a decision variable. Data that depends on the arrangement, such as the distance between consecutive stops, is read from a `lookup_table()`: `dist[item_at(1:4, route), item_at(2:5, route)]` is the length of each of the four legs of a five-stop route.

`precede()` requires one item to be in an earlier slot than another, such as a pickup before its delivery.

The answer

For each permutation, the result of `solve()` holds both directions under `res$structures$<name>: item_at`, the item in each slot (for a route, the stops in the order they are visited), and `slot_of`, the slot of each item. `set_start()`, `evaluate()` and `resample()` take the arrangement from a result too.

A model with a permutation is solved by a search, so its status is "heuristic". Permutations can be combined with random variables, for example a route whose travel times are uncertain, minimized in `expectation()`. `vignette("permutations", package = "quicopt")` works through two examples.

`add_perm_var()` declares the permutation in the same way but returns the model, for use in a pipe; `m$name` then retrieves the permutation.

Examples

```
# Five stops along a road. Find the shortest route through all of them
# that visits stop 4 before stop 1.
position <- c(0, 3, 1, 4, 2)                # km along the road, stops 1 to 5
m <- model()
dist <- lookup_table(m, "dist", abs(outer(position, position, "-")))
route <- perm_var(m, "route", 5)
precede(route, 4, 1)                        # stop 4 is visited before stop 1
minimize(m, sum(dist[item_at(1:4, route), item_at(2:5, route)]))
## Not run:
res <- solve(m)
res$structures$route$item_at                # the stops in the order visited

## End(Not run)
```

permutation_decl	<i>A permutation, for building a program by hand</i>
------------------	--

Description

Declares one permutation of a `program()`: size items in size slots, one item per slot (see `perm_var()`). `perm_var()` builds these for you.

Usage

```
permutation_decl(size, start = integer(), fixed = FALSE, precede = list())
```

Arguments

size	How many items, at least 2.
start	The arrangement the search starts from: <code>start[i]</code> is the slot of item <code>i</code> , so <code>start</code> contains each of the numbers 1 to size once. <code>integer()</code> starts with item <code>i</code> in slot <code>i</code> .

fixed	Whether the permutation is fixed at start, which must then be given. This is how <code>evaluate()</code> and <code>resample()</code> keep the arrangement of a solution.
precede	A list of pairs <code>c(before, after)</code> , each requiring item before to be in an earlier slot than item after.

Value

A plain list without a class, with the fields `kind = "permutation"`, `size`, `start`, `fixed` and `precede`. It is an element of the named list a `program()` takes as structures, and the element's name is the name `ir_struct_ref()` refers to.

Examples

```
# five stops, stop 4 before stop 1
structures <- list(route = permutation_decl(5, precede = list(c(4, 1))))
```

precede	<i>Require one item to come before another</i>
---------	--

Description

`precede(P, before, after)` requires item before to be in an earlier slot of the permutation `P` than item after: a pickup before its delivery, or a job before the job that needs its output. The service only considers arrangements that meet the requirement, so every solution meets it.

Usage

```
precede(P, before, after)
```

Arguments

<code>P</code>	A permutation from <code>perm_var()</code> .
<code>before, after</code>	Items: whole numbers from 1 to the size of the permutation.

Details

`precede()` may be called several times for one permutation. Requirements that contradict each other, such as 1 before 2 and 2 before 1, are an error.

Value

The permutation, invisibly.

Examples

```
m <- model()
route <- perm_var(m, "route", 5)
precede(route, 4, 1)           # stop 4 is visited before stop 1
precede(route, 2, 5)           # and stop 2 before stop 5
```

prob

The probability that a comparison holds

Description

`prob(a <= b)` is the share of the scenarios in which `a <= b` holds: an estimate of its probability. It is a single number, so it can be used in a constraint. A requirement on a probability is called a *chance constraint*:

Usage

```
prob(rel)
```

Arguments

`rel` A comparison of expressions, written with `<=` or `>=`.

Details

```
add(m, prob(demand <= stock) >= 0.9)
```

reads as "demand is met in at least 90% of the scenarios". The line holds two comparisons, which do different jobs: the inner one, `demand <= stock`, is the event checked in each scenario, and the outer one, `>= 0.9`, is the requirement on how often it happens. The `margin` argument of `add()` allows for the sampling error of the estimate.

The comparison is written with `<=` or `>=`, and at least one side must contain a random variable. `==` is not accepted, since the probability that a quantity which can take any value equals one particular value is 0. `<` and `>` are not accepted either, since for such a quantity they mean the same as `<=` and `>=`. For vector expressions, each element gets its own probability.

`holds()` turns the same comparison into a 0/1 value in each scenario, and `expectation(holds(a <= b))` is the same number as `prob(a <= b)`.

Value

An expression with one probability, between 0 and 1, per element of the comparison.

Examples

```
m <- model()
stock <- num_var(m, "stock", lower = 0, upper = 200)
demand <- rand_var(m, "demand", normal(100, 15))
set_scenarios(m, 512, seed = 42)
add(m, prob(demand <= stock) >= 0.9)                      # demand met in at least 90% of scenarios
```

program

A model as plain data

Description

A program holds a complete model as plain R lists, in the form the service reads: `encode()` turns it into bytes, and `solve()` and `submit()` accept it directly. `as_program()` converts a `model()` into a program. Building one by hand, with this function and the ones it links to, is for parts of the format that a `model()` does not offer, such as index sets and parameter tables.

Usage

```
program(
  sets = list(),
  indexed_sets = list(),
  params = list(),
  vars = list(),
  objective = NULL,
  sense = "min",
  constraints = list(),
  fix = list(),
  scenarios = 1,
  scenario_seed = 1,
  sources = list(),
  structures = list(),
  blocks = list()
)
```

Arguments

sets	A list of <code>index_set()</code> s.
indexed_sets	Sets that differ from position to position (see above).
params	Parameter tables (see above).
vars	A list of <code>var_decl()</code> s.
objective	The objective, an expression node (see <code>ir</code>).
sense	"min" or "max".
constraints	A list of <code>constraint()</code> s.
fix	Variables fixed at a value (see above).
scenarios	How many scenarios to draw, at least 1.
scenario_seed	The seed to draw them from, at least 1.
sources	The random variables: a named list of <code>parametric()</code> and <code>empirical()</code> declarations.
structures	The permutations and partitions: a named list of <code>permutation_decl()</code> and <code>partition_decl()</code> declarations.
blocks	The blocks (see <code>block()</code>): a named list, each element a list of <code>ir_var()</code> nodes, the block's variables.

Details

Data indexed by positions is given as lists of entries, because a position (a list of numbers and strings) cannot be a name:

- `params` maps the name of a parameter table to a list of entries `list(key = <position>, value = <number>);`
- `indexed_sets` maps a name to a list of entries `list(key = <position>, value = <list of elements>)`, a set that differs from position to position;
- `fix` is a list of entries `list(var = , index = , value = )`, each fixing one variable at a value.

The order of the entries does not matter.

A model with random variables also needs `sources`, which declares them, and `scenarios` and `scenario_seed`, which say how many scenarios are drawn and from which seed. A model with permutations or partitions needs `structures`, and one with blocks `blocks`.

Value

An object of class `quicopt_program`: a list with one element per argument, under the argument's name.

<code>rand_var</code>	<i>Declare a random variable</i>
-----------------------	----------------------------------

Description

A random variable stands for a quantity that is not known when the decision is made, such as tomorrow's demand. It can be used in expressions like a decision variable, but the service does not choose its value: in each scenario, the value is drawn from the variable's distribution. Within one scenario, every use of the variable has the same value. Two random variables declared separately are drawn independently.

Usage

```
rand_var(m, name, dist = NULL, length = NULL, n)
```

```
add_rand_var(m, name, dist = NULL, length = NULL, n)
```

Arguments

<code>m</code>	A <code>model()</code> .
<code>name</code>	The random variable's name, unique within the model.
<code>dist</code>	A distribution such as <code>normal(100, 15)</code> (see <code>distribution()</code>), or an <code>empirical()</code> sample with one value per scenario. It may be left out and given later with <code>set_distribution()</code> .
<code>length</code>	How many random variables to declare under this name. Left <code>NULL</code> , the number follows from the length of the distribution's parameters.
<code>n</code>	The former name of <code>length</code> , still accepted for now, with a warning.

Details

A random variable has no bounds; its distribution says which values it can take.

A distribution with vector parameters declares several random variables under one name, which behave like an R vector: `rand_var(m, "hours", normal(c(6, 5, 4), 1))` declares `hours[1]`, `hours[2]` and `hours[3]`. With scalar parameters and length given, it declares that many variables with the same distribution. Either way, the elements are drawn independently of each other. Random variables that move together, such as demand and price, are best declared from observed data with `set_empirical()`.

`add_rand_var()` declares the variable in the same way but returns the model, for use in a pipe; `m$name` then retrieves the variable.

Value

The random variable, an expression with one element per random variable declared.

`add_rand_var()` returns the model, invisibly.

Examples

```
m <- model()
demand <- rand_var(m, "demand", normal(100, 15))
hours <- rand_var(m, "hours", normal(c(6, 5, 4), 1)) # three, one per job
delay <- rand_var(m, "delay", uniform(1, 1.5), length = 4) # four with the same distribution
```

resample

Check a solution on new scenarios

Description

A solution is chosen to do well on the model's scenarios, so the objective and the probabilities reported for it are measured on the very scenarios that shaped it. On new scenarios it usually does a little worse, and a chance constraint that was only just met is missed about half the time. `resample()` draws new scenarios from seed, keeps the solution fixed, and computes the model's objective and constraints on them.

Usage

```
resample(m, solution, seed, n_scenarios = NULL, ..., scenarios)
```

Arguments

<code>m</code>	A <code>model()</code> .
<code>solution</code>	A result from <code>solve()</code> , or a named numeric vector with a value for every decision variable (see <code>evaluate()</code>).
<code>seed</code>	The seed for the new scenarios, different from the model's own.
<code>n_scenarios</code>	How many new scenarios to draw. Left NULL, as many as the model has. More scenarios give a more precise check.

... Settings for the request, as for `solve_model()`.

scenarios The former name of `n_scenarios`, still accepted for now, with a warning.

Details

Only random variables with a distribution are drawn again. A random variable made from an `empirical()` sample or by `set_empirical()` is data and stays as it is. A model whose random variables are all of that kind has nothing to draw again, which is an error, and while a model has any of them, the number of scenarios cannot change. Each call is one request to the service.

Value

A result in the same form as from `solve()`, for the fixed solution on the new scenarios. `objective` is the objective on them, `feasible` says whether every constraint still holds, and `solver_data$max_violation` is by how much the worst constraint is missed; for a chance constraint, as a probability.

Examples

```
## Not run:
res <- solve(m)                                # m as in ?evaluate
chk <- resample(m, res, seed = 7, n_scenarios = 1000)
chk$feasible                                  # does every constraint still hold?
chk$solver_data$max_violation                 # if not, by how much
chk$objective                                 # the objective on the new scenarios

## End(Not run)
```

scenario_max	<i>The largest, the smallest and a quantile over the scenarios</i>
--------------	--

Description

The value a quantity takes in one particular scenario:

Usage

```
scenario_max(x)

scenario_min(x)

scenario_quantile(x, prob)
```

Arguments

`x` A random expression (see `is_random()`).

`prob` A number above 0 and at most 1. It cannot depend on a decision.

Details

- `scenario_max(x)` is the largest value of `x` among the scenarios. Minimizing it makes the worst scenario as good as possible.
- `scenario_min(x)` is the smallest. Maximizing it does the same for a quantity where large is good, such as a profit.
- `scenario_quantile(x, prob)` is the value that the share `prob` of the scenarios stays at or below. With `n_scenarios` scenarios it is the `ceiling(prob * n_scenarios)`-th smallest value, which is what `quantile(x, prob, type = 1)` returns for a sample. `scenario_quantile(x, 1)` is `scenario_max(x)`. For a cost it is also known as the *value at risk*; `cvar()` at the same level is the average of the values above it.

These are different from `max()`, `min()` and `quantile()`. `max(a, b)` compares two expressions *within* each scenario, and the result is still random. `scenario_max(x)` compares the values of one expression *across* the scenarios, and the result is a single number.

A single scenario decides the largest or smallest value, so it changes more from one sample of scenarios to the next than an average does, and a larger sample usually contains a more extreme scenario. Check a solution found with these on new scenarios, with `resample()`.

Value

An expression of the same length as `x`, no longer random.

Examples

```
m <- model()
stock <- num_var(m, "stock", lower = 0, upper = 200)
demand <- rand_var(m, "demand", normal(100, 15))
set_scenarios(m, 512, seed = 42)
cost <- 3 * stock + 10 * max(demand - stock, 0)
minimize(m, scenario_max(cost))           # the cost of the worst scenario
add(m, scenario_quantile(cost, 0.95) <= 500) # at most 500 in 95% of scenarios
```

set_distribution	<i>Give a random variable its distribution</i>
------------------	--

Description

Sets or replaces the distribution of a random variable declared with `rand_var()`, for example one declared without a distribution.

Usage

```
set_distribution(m, v, dist)
```

Arguments

m	A <code>model()</code> .
v	The random variable, as returned by <code>rand_var()</code> .
dist	A distribution such as <code>normal(100, 15)</code> , or an <code>empirical()</code> sample, of the same length as v.

Value

The model, invisibly.

Examples

```
m <- model()
demand <- rand_var(m, "demand")
set_distribution(m, demand, normal(100, 15))
```

set_empirical	<i>Use observed data as a model's uncertainty</i>
---------------	---

Description

Turns the columns of a data frame into random variables: each column becomes one, named after the column, and each row becomes one scenario. The columns are read row by row, so values observed together stay together, and any correlation between the columns carries over into the model. No distribution has to be chosen or fitted.

Usage

```
set_empirical(m, data, cols = NULL)
```

Arguments

m	A <code>model()</code> .
data	A data frame with one row per observation.
cols	The names of the columns to use. Left NULL, all of them.

Details

The random variables are not returned; retrieve them by name, as `m$demand`. The number of scenarios is set to the number of rows.

Every column used must be numeric; a column that is not is an error rather than being skipped. Use `cols` to select the columns that describe the uncertainty when the data frame holds others too.

Value

The model, invisibly.

Examples

```
history <- data.frame(demand = c(96, 104, 121, 88, 110),
                      price  = c(12.1, 11.8, 11.2, 12.5, 11.6))
m <- model()
stock <- num_var(m, "stock", lower = 0, upper = 200)
set_empirical(m, history)
maximize(m, expectation(m$price * min(m$demand, stock)) - 3 * stock)
```

set_scenarios

Set how many scenarios are drawn, and from which seed

Description

The service draws `n_scenarios` scenarios, each a possible outcome of the model's random variables. More scenarios describe the uncertainty more accurately, and take longer to solve. A model whose scenarios are never set is solved over a single scenario, unless an `empirical()` sample sets the number by its length.

Usage

```
set_scenarios(m, n_scenarios, seed = NULL)
```

Arguments

<code>m</code>	A <code>model()</code> .
<code>n_scenarios</code>	How many scenarios to draw, at least 1.
<code>seed</code>	The seed for the draws, at least 1. Left NULL, the model keeps its current seed.

Details

The number of scenarios and the seed belong to the model, so solving the same model again uses the same scenarios, and two solves of it can be compared. The scenarios are drawn by the service, so R's `set.seed()` has no effect on them.

The service limits the number of scenarios. A model above the limit is refused when it is solved, with a message that states the limit.

Value

The model, invisibly.

Examples

```
m <- model()
demand <- rand_var(m, "demand", normal(100, 15))
set_scenarios(m, 512, seed = 42)
```

set_start	<i>Start the next search from a known solution</i>
-----------	--

Description

The service's search starts from each variable's start value, which is 0 unless set otherwise. When a model is solved again after a small change, such as a tightened constraint or more scenarios, starting from the previous answer usually gets to a good solution sooner than starting from scratch. `set_start()` sets the start values from a result, or from a named vector of values. Variables it is not given keep their start value.

Usage

```
set_start(m, values)
```

Arguments

<code>m</code>	A <code>model()</code> .
<code>values</code>	A result from <code>solve()</code> , or a named numeric vector, with names as in a solution: "x" for a single variable, "x[1]", "x[2]", ... for the elements of a vector variable.

Details

The service rounds the start of a whole-number variable, and moves a start that lies outside the variable's bounds to the nearest bound. A result also holds the ordering found for each permutation (see `perm_var()`) and the groups found for each partition (see `part_var()`), and these become their starts.

Value

The model, invisibly.

Examples

```
## Not run:
m <- model()
stock <- num_var(m, "stock", lower = 0, upper = 200)
demand <- rand_var(m, "demand", normal(100, 15))
set_scenarios(m, 512, seed = 42)
minimize(m, 3 * stock + 10 * expectation(max(demand - stock, 0)))
res <- solve(m)

set_scenarios(m, 1000, seed = 42) # the same model, more scenarios
set_start(m, res)                 # start where the last search ended
solve(m)

## End(Not run)
```

solve_model

Solve a model

Description

solve() sends a model to the Quicopt service, waits for the answer, and returns it as a list (see the Value section). solve_model() is the same function under a name that cannot be confused with base R's solve(), which can read better in a pipe.

Usage

```
solve_model(
  m,
  base_url = DEFAULT_BASE_URL,
  api_key = NULL,
  project = NULL,
  config = NULL,
  gzip = FALSE,
  timeout = 60,
  time_limit = NULL,
  transport = NULL
)

## S3 method for class 'quicopt_model'
solve(a, b, ...)
```

Arguments

m	A model() . A program() , or the bytes from encode() , also work.
base_url	The address of the service.
api_key	Your API key, or NULL to use the session's free key (see the API keys section).
project	A project name, for billing by project, or NULL.
config	A named list of further settings, sent to the service as query parameters.
gzip	Whether to compress the model before sending it; worth it for a large model.
timeout	How many seconds to wait for the answer.
time_limit	How many seconds the solve may run, or NULL to leave that to the service (see the Time limit section).
transport	For tests: a function that is called instead of sending the request. It takes a list with the elements method, url, headers, body and timeout, and returns a list with the elements status, headers and body.
a	The model. The argument is called a because base R's solve() calls it that.
b	Not used; giving it is an error.
...	Further arguments for solve_model(), by name.

Details

`solve()` waits for up to `timeout` seconds. For a model that takes longer, `submit()` sends it without waiting.

Value

A list of class `quicopt_result`, with these elements:

- `status`: what kind of answer it is (see the Status section).
- `feasible`: whether the solution meets every constraint.
- `objective`: the value of the objective at the solution.
- `solution`: a named numeric vector with the value of each decision variable, in the order they were declared.
- `structures`: for a model with permutations or partitions, one element per permutation, holding the integer vectors `item_at` and `slot_of` (see `perm_var()`), and one per partition, holding `part_of` and `size_of` (see `part_var()`); otherwise `NULL`.
- `model_class`: the kind of model the service recognized, such as "lp", "milp" or "stochastic".
- `solver_data`: further details from the service, such as `max_violation`, by how much the worst constraint is missed.
- `display`: a summary prepared by the service, which is what printing the result shows.

Time limit

The service decides how long a solve may run. `time_limit` sets that time instead, in seconds. A solver that proves its answer sooner still stops sooner; a search, which cannot prove it is done, may stop sooner too, once it stops finding better answers. The service grants at most its maximum (120 seconds on the free tier). Asked for more, it runs for its maximum, and a warning says that the solve got less time than requested. `res$solver_data$time_budget_s` is the time the service granted.

`solve()` waits for the answer at least `time_limit` plus a minute, even when `timeout` is shorter.

API keys

The service needs an API key. The first time a model is solved in an R session without one, the service issues a free key, and `quicopt` keeps it in memory until the session ends; it is never written to disk. To use a key of your own, pass it as `api_key`. It is then used for that call only, and not kept.

Status

`status` says what kind of answer the result holds:

- "optimal": the service has proved that no better solution exists.
- "heuristic": the best solution a search found, without that proof. Models with random variables, permutations, partitions or lookup tables are solved this way, and so are models with `holds()`, `max()` or `min()`. If such a result has `feasible = FALSE`, the search found no solution that meets every constraint, which does not prove that none exists.

Other values say why no solution is available.

Examples

```
## Not run:
m <- model()
tables <- int_var(m, "tables", lower = 0)
chairs <- int_var(m, "chairs", lower = 0)
maximize(m, 50 * tables + 20 * chairs)
add(m, 3 * tables + chairs <= 41)
add(m, tables + chairs <= 18)

res <- solve(m)
res$status
res$solution
res

res <- solve(m, time_limit = 30)
res$solver_data$time_budget_s # 30: the seconds the service granted

## End(Not run)
```

stochastic

Optimization under uncertainty

Description

Some decisions have to be made before all the data is known: how much stock to order before demand is known, which jobs to accept before knowing how long they will take. quicopt models such a decision by describing each uncertain quantity as a *random variable*. The service draws many possible outcomes of the random variables, called *scenarios*, and finds the decision that does best across them.

Building blocks

- `rand_var()` declares a random variable with a distribution such as `normal()`; `set_empirical()` declares random variables from the columns of a data frame instead, one row per scenario.
- `set_scenarios()` sets how many scenarios the service draws, and the seed it draws them from.
- Any expression that contains a random variable has one value per scenario; it is called *random* (see `is_random()`). The objective and the constraints must each be one number, so a random expression is summarized across the scenarios before it is used there.

Summaries across the scenarios

- `expectation()`: the average.
- `prob()`: the share of scenarios in which a comparison holds, for a requirement such as "demand is met on 90% of days".
- `cvar()`: the average over the worst scenarios.

- `scenario_max()`, `scenario_min()`, `scenario_quantile()`: the largest value, the smallest, and a quantile.
- `variance()` and `std_dev()`: how much the value varies.

`holds()` turns a comparison into a 0/1 value in each scenario, which can be combined with other quantities before it is summarized.

Checking a solution

A solution is chosen to do well on the model's scenarios, so it tends to do a little worse on others, and the objective the service reports is optimistic. `evaluate()` and `resample()` compute the figures on new scenarios, and the margin argument of `add()` raises the target of a requirement on a probability to allow for this.

`vignette("stochastic", package = "quicopt")` introduces all of this with a worked example.

Examples

```
## Not run:
# Order stock at 3 per unit before demand is known, pay 10 for each unit
# of demand that cannot be met, and meet demand in at least 90% of scenarios.
m <- model()
stock <- num_var(m, "stock", lower = 0, upper = 200)
demand <- rand_var(m, "demand", normal(100, 15))
set_scenarios(m, 512, seed = 42)
minimize(m, 3 * stock + 10 * expectation(max(demand - stock, 0)))
add(m, prob(demand <= stock) >= 0.9)
res <- solve(m)
res$solution

# Does the requirement still hold on 1000 new scenarios?
resample(m, res, seed = 7, n_scenarios = 1000)$feasible

## End(Not run)
```

submit

Solve a model without waiting

Description

`submit()` sends a model to the service like `solve()` does, but returns at once with a *job*, while the service solves the model in the background and your R session can go on. `job_result()` collects the answer when it is ready, and `job_status()` shows how far the job has got.

Usage

```
submit(
  m,
  base_url = DEFAULT_BASE_URL,
  api_key = NULL,
```

```

    project = NULL,
    config = NULL,
    gzip = FALSE,
    timeout = 60,
    time_limit = NULL,
    transport = NULL
  )

```

Arguments

<code>m</code>	A <code>model()</code> . A <code>program()</code> , or the bytes from <code>encode()</code> , also work.
<code>base_url</code>	The address of the service.
<code>api_key</code>	Your API key, or NULL to use the session's free key (see the API keys section).
<code>project</code>	A project name, for billing by project, or NULL.
<code>config</code>	A named list of further settings, sent to the service as query parameters.
<code>gzip</code>	Whether to compress the model before sending it; worth it for a large model.
<code>timeout</code>	How many seconds to wait for the answer.
<code>time_limit</code>	How many seconds the solve may run, or NULL to leave that to the service (see the Time limit section).
<code>transport</code>	For tests: a function that is called instead of sending the request. It takes a list with the elements <code>method</code> , <code>url</code> , <code>headers</code> , <code>body</code> and <code>timeout</code> , and returns a list with the elements <code>status</code> , <code>headers</code> and <code>body</code> .

Details

The job keeps the address, the key and the other settings it was submitted with, so the functions that follow it up need only the job. Given a `time_limit`, `job_result()` waits at least that long plus a minute.

Value

A job, of class `quicopt_job`, for `job_result()`, `job_status()`, `job_log()` and `job_delete()`.

Examples

```

## Not run:
job <- submit(m)
job_status(job)$status
res <- job_result(job)      # waits until the job is finished

## End(Not run)

```

summaries

*Sums, products, maxima and minima***Description**

`sum()`, `prod()`, `max()` and `min()` work on model expressions as they do on numbers: each combines all the elements of all its arguments into one value, so `max(x, 0)` is the largest of all the elements of `x` and 0. The arguments can be expressions, numbers or both, in any order: `max(0, demand - stock)` and `max(demand - stock, 0)` mean the same.

Usage

```
sum(..., na.rm = FALSE)
```

```
prod(..., na.rm = FALSE)
```

```
max(..., na.rm = FALSE)
```

```
min(..., na.rm = FALSE)
```

Arguments

<code>...</code>	Expressions, numbers, or both.
<code>na.rm</code>	Passed to base R when no argument is an expression, and ignored otherwise, since an expression has no missing values.

Details

`quicopt` provides its own versions of these four functions, because base R would read `max(0, x)` with its numeric `max()` and stop with an error. Called without any expression, they are base R's functions, so loading the package changes nothing about how they work on numbers. R reports this when the package is attached, as these functions being masked from base.

The status of a model with `max()` or `min()` is "heuristic": a good answer, not a proven optimum.

Value

An expression with one element when any argument is an expression; otherwise what base R returns.

Examples

```
m <- model()
stock <- num_var(m, "stock", lower = 0, upper = 100)
demand <- c(60, 80, 120)
max(0, demand - stock)      # the largest shortfall of the three
sum(2, stock)               # 2 + stock
max(3, 7)                   # 7, as in base R
```

variance

*The variance and the standard deviation over the scenarios***Description**

How much a quantity varies from scenario to scenario, as opposed to what it is on average. `expectation(cost) + 2 * std_dev(cost)` is an objective that trades a low average cost against a steady one, and `add(m, variance(cost) <= 100)` limits the variation.

Usage

```
variance(x, sample = FALSE)
```

```
std_dev(x, sample = FALSE)
```

Arguments

<code>x</code>	A random expression (see is_random()).
<code>sample</code>	FALSE, the default, divides by the number of scenarios; TRUE divides by one less, as <code>var()</code> and <code>sd()</code> do.

Details

By default the scenarios count as the whole distribution, each with weight $1 / n_{\text{scenarios}}$, so `variance(x)` is $\text{expectation}(x^2) - \text{expectation}(x)^2$. R's `var()` and `sd()` divide by $n_{\text{scenarios}} - 1$ instead, because they estimate the variance of a population from a sample of it; `sample = TRUE` does the same. The two differ by a factor $n_{\text{scenarios}} / (n_{\text{scenarios}} - 1)$, which changes the value reported but not which decision is best.

Unlike [cvar\(\)](#), these measure variation in both directions: a scenario far better than average increases them as much as one far worse.

Value

An expression of the same length as `x`, no longer random.

Examples

```
m <- model()
stock <- num_var(m, "stock", lower = 0, upper = 200)
demand <- rand_var(m, "demand", normal(100, 15))
set_scenarios(m, 512, seed = 42)
cost <- 3 * stock + 10 * max(demand - stock, 0)
minimize(m, expectation(cost) + 2 * std_dev(cost))
```

`wire`*The bytes sent to the service*

Description

`solve()` sends a model to the service as a sequence of bytes, in a format defined by the service. `encode()` produces these bytes from a model or a `program()`, and `encode_params()` the bytes for parameter tables on their own. Encoding gives the same bytes for the same model every time.

Details

You need these functions only to store the bytes, compare them, or send them yourself; `solve()` encodes a model for you.

Index

[.quicopt_table (lookup_table), 25

active, 3

active(), 18, 19, 29

add, 4

add(), 8, 17, 18, 27, 37, 49

add_part_var (part_var), 32

add_perm_var (perm_var), 34

add_rand_var (rand_var), 39

add_var (num_var), 28

as_program, 5

as_program(), 38

bernoulli (distribution), 11

bin_var (num_var), 28

bin_var(), 5, 9, 27

BINARY (CONTINUOUS), 9

block, 6

block(), 27, 38

consets, 7

constraint, 8

constraint(), 8, 19, 21, 38

CONTINUOUS, 9

cvar, 10

cvar(), 16, 42, 48, 52

DEFAULT_BASE_URL, 11

distribution, 11

distribution(), 39

empirical, 13

empirical(), 12, 38, 39, 41, 43, 44

encode, 14

encode(), 5, 38, 46, 50, 53

encode_params, 14

encode_params(), 53

evaluate, 15

evaluate(), 32, 33, 35, 36, 40, 49

expectation, 16

expectation(), 4, 15, 17, 22, 26, 27, 35, 48

exponential (distribution), 11

expressions, 17

holds, 18

holds(), 4, 5, 17, 18, 37, 47, 49

index_set, 19

index_set(), 38

indicator (consets), 7

indicator(), 8

int_var (num_var), 28

int_var(), 9, 27

INTEGER (CONTINUOUS), 9

ir, 20, 38

ir_apply (ir), 20

ir_const (ir), 20

ir_param (ir), 20

ir_reduce (ir), 20

ir_reduce(), 19

ir_set_ref (ir), 20

ir_set_ref(), 8

ir_source_ref (ir), 20

ir_source_ref(), 30

ir_struct_ref (ir), 20

ir_struct_ref(), 33, 36

ir_table_ref (ir), 20

ir_var (ir), 20

ir_var(), 7, 38

is_random, 22

is_random(), 10, 16, 41, 48, 52

item_at, 23

item_at(), 21, 25, 34

job_delete (job_status), 24

job_delete(), 50

job_log (job_status), 24

job_log(), 50

job_result (job_status), 24

job_result(), 49, 50

job_status, 24

job_status(), 49, 50
 lookup_table, 25
 lookup_table(), 21, 23, 31, 32, 34
 max (summaries), 51
 maximize (minimize), 26
 maximize(), 27
 min (summaries), 51
 minimize, 26
 minimize(), 27
 model, 27
 model(), 4–6, 14, 15, 20, 25, 26, 29, 32, 34, 38–40, 43–46, 50
 nonneg (consets), 7
 nonneg(), 8
 normal (distribution), 11
 normal(), 30, 48
 num_var, 28
 num_var(), 3, 9, 10, 27
 parametric, 30
 parametric(), 38
 part_of, 31
 part_of(), 21, 32
 part_var, 32
 part_var(), 27, 31, 33, 45, 47
 partition_decl, 33
 partition_decl(), 38
 perm_var, 34
 perm_var(), 23, 27, 35, 36, 45, 47
 permutation_decl, 35
 permutation_decl(), 38
 precede, 36
 precede(), 34
 prob, 37
 prob(), 4, 15, 17, 48
 prod (summaries), 51
 program, 38
 program(), 5, 7–10, 13, 14, 20, 30, 33, 35, 36, 46, 50, 53
 rand_var, 39
 rand_var(), 11–13, 27, 30, 42, 43, 48
 resample, 40
 resample(), 4, 13, 15, 32, 33, 35, 36, 42, 49
 scenario_max, 41
 scenario_max(), 49
 scenario_min (scenario_max), 41
 scenario_min(), 49
 scenario_quantile (scenario_max), 41
 scenario_quantile(), 49
 SEMICONTINUOUS (CONTINUOUS), 9
 SEMIINTEGER (CONTINUOUS), 9
 set_distribution, 42
 set_distribution(), 12, 13, 39
 set_empirical, 43
 set_empirical(), 13, 27, 40, 41, 48
 set_scenarios, 44
 set_scenarios(), 4, 48
 set_start, 45
 set_start(), 29, 32, 35
 size_of (part_of), 31
 size_of(), 21, 32
 slot_of (item_at), 23
 slot_of(), 21, 34
 solve(), 5, 11, 14, 15, 24, 27, 32, 35, 38, 40, 41, 45, 49, 53
 solve.quicopt_model (solve_model), 46
 solve_model, 46
 solve_model(), 15, 41
 std_dev (variance), 52
 std_dev(), 49
 stochastic, 27, 48
 submit, 49
 submit(), 11, 24, 38, 47
 sum (summaries), 51
 summaries, 17, 51
 uniform (distribution), 11
 var_decl (CONTINUOUS), 9
 var_decl(), 19, 38
 variance, 52
 variance(), 18, 49
 wire, 53
 zero (consets), 7
 zero(), 8