

Package ‘quicopt’

October 14, 2026

Title Client for the 'Quicopt' Optimization Service

Version 0.1.1

Description Author an optimization model — including a model under uncertainty, with random variables, expectations, conditional value at risk and chance constraints — in a native R interface, and solve it with the 'Quicopt' service <<https://quicopt.com>>. Models are encoded to the service's wire format and solved remotely; no solver is installed locally. Random variables carry distributions or empirical scenario columns taken directly from a data frame, so observed history becomes a stochastic program in one call.

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>

Config/roxygen2/version 8.1.0

NeedsCompilation no

Author Tim Bode [aut, cre, cph]

Maintainer Tim Bode <t.bode@fz-juelich.de>

Repository CRAN

Date/Publication 2026-10-14 11:10:08 UTC

Contents

add	2
as_program	3
consets	3
constraint	4
CONTINUOUS	5
cvar	6
DEFAULT_BASE_URL	6
distribution	7
empirical	7
encode	8
encode_params	9
expectation	9
expressions	10
index_set	10
ir	11
job_status	12
minimize	13
model	13
num_var	14
parametric	15
prob	15
program	16
rand_var	17
set_distribution	18
set_empirical	19
set_scenarios	19
solve_model	20
stochastic	21
submit	22
wire	23
Index	24

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

Description

A comparison of model expressions is a constraint, not a logical: `add(m, x + y <= 5)` requires the row to hold, and `==` states an equality. A comparison of vector expressions adds one row per element, so `add(m, x <= cap)` with two length- n vectors is n rows.

Usage

`add(m, rel)`

Arguments

`m` A `model()`.

`rel` A comparison built with `<=`, `>=` or `==`.

Value

The model, invisibly.

as_program	<i>Lower a model to a program</i>
------------	-----------------------------------

Description

The program is the model as plain data — what `encode()` serializes and the service reads. `solve()` does this on the way out; call it directly to inspect what will be sent.

Usage

```
as_program(m)
```

Arguments

`m` A `model()`.

Value

A `program()`.

consets	<i>Constraint sets</i>
---------	------------------------

Description

A constraint is a set membership: the expression `f` must land in the set, so `x + 2*y <= 5` is written as `5 - (x + 2*y)` in `nonneg()` — one sign convention rather than two.

Usage

```
zero()
```

```
nonneg()
```

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

Arguments

- bin The binary `ir_var()` whose activity implies the inner set.
- inner The constraint set that holds when bin is active.

Value

A plain list, with no class attribute, naming the set a constrained expression must lie in; it is what `constraint()` takes as set. `zero()` returns `list(kind = "zero")`, meaning the expression equals 0. `nonneg()` returns `list(kind = "nonneg")`, meaning the expression is at least 0. `indicator()` returns a list with `kind = "indicator"` and the fields `bin` and `inner` as given, meaning `inner` is imposed only where `bin` is active.

constraint	<i>A constraint row</i>
------------	-------------------------

Description

A constraint row

Usage

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

Arguments

- f The constrained expression node.
- set The constraint set f must lie in (`zero()`, `nonneg()`, `indicator()`).
- over Quantifier bindings, a list of `list(idx, set_ref)` pairs (`list()` for a single scalar row).

Value

A plain list, with no class attribute, with the fields `f`, `set` and `over` as given. It states that `f` lies in `set`, once for every binding of the indices in `over`, and is an entry of the list a `program()` takes as constraints.

CONTINUOUS	<i>A variable declaration</i>
------------	-------------------------------

Description

A variable declaration

Usage

CONTINUOUS

INTEGER

BINARY

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

Arguments

name	The variable's name; solutions come back keyed by it.
axes	Index-set names the variable ranges over (character() for a scalar).
domain	CONTINUOUS, INTEGER or BINARY.
lower, upper	A number (-Inf/Inf for an open direction), or the name of a parameter table when the bound varies by index.
start	The initial point handed to the solver.

Value

var_decl() returns a plain list, with no class attribute, with the fields name, axes, domain (the integer domain code), lower, upper and start (numeric). It declares one variable of the model, and is an entry of the list a program() takes as vars.

CONTINUOUS, INTEGER and BINARY are not functions but integer constants (1L, 2L and 3L): the codes the service uses for a variable's domain, to be passed as domain.

cvar	<i>The conditional value at risk at level alpha</i>
------	---

Description

The mean of x over its worst $1 - \alpha$ fraction of scenarios — at $\alpha = 0.95$, the average of the worst 5%. Minimizing it optimizes the tail instead of the average, and is the usual way to ask for a solution that holds up in bad scenarios rather than merely on average.

Usage

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

Arguments

x	A model expression.
α	The tail level, a plain number strictly between 0 and 1; it cannot depend on a decision.

Value

An expression of the same length.

DEFAULT_BASE_URL	<i>The public Quicopt endpoint</i>
------------------	------------------------------------

Description

`solve()` targets it unless another `base_url` is given.

Usage

```
DEFAULT_BASE_URL
```

Value

Not a function but a constant: a character string of length one, the URL of the public service that a request is sent to by default.

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

Description

`normal(mean, sd)` follows `rnorm()`'s parameterization: the mean and the standard deviation. `distribution(head, ...)` is the escape hatch for any distribution the service supports that has no named constructor here yet.

Usage

```
distribution(head, ...)
```

```
normal(mean, sd)
```

Arguments

<code>head</code>	The distribution's name in the service's catalog.
<code>...</code>	Its parameters, each a number or a deterministic expression.
<code>mean, sd</code>	The mean and standard deviation, as in <code>rnorm()</code> .

Details

A parameter may be a number or a deterministic model expression. An expression gives an *endogenous* distribution — one whose parameters depend on the decision, such as a demand whose mean rises with the price you set. A parameter may never contain a random variable: a distribution's parameters are data, not draws.

Value

A distribution, ready for `rand_var()` or `set_distribution()`.

empirical	<i>A random variable given as a fixed scenario column</i>
-----------	---

Description

Exactly scenarios values, one per scenario. Several empirical columns are read at the same scenario index, so columns observed jointly stay correlated — which is how a joint distribution is expressed.

Usage

```
empirical(data)
```

Arguments

data A numeric vector, one value per scenario.

Value

An object of class `quicopt_empirical`: a list with the fields `kind = "empirical"` and `data`, the column as a numeric vector. It declares one random variable by its observed values, and is an entry of the named list a `program()` takes as sources.

encode	<i>Encode a program to the bytes the service reads</i>
--------	--

Description

Two equal programs always encode to equal bytes, whichever order their tables happened to be built in. A model that declares no uncertainty encodes to exactly the bytes it would have before the stochastic layer existed, so declaring none costs an ordinary model nothing.

Usage

```
encode(prog)
```

Arguments

prog A `program()`, or a model built with `model()` (which is lowered first).

Details

Encoding is normally invisible: `solve_model()` does it for you, and what it sends is exactly these bytes. Reach for `encode` to send them yourself, store them, or check them.

Value

A raw vector: the encoded program.

encode_params	<i>Encode parameter tables alone, for rebinding data</i>
---------------	--

Description

Send the program once, then one of these per instance to re-solve the same structure on new data without re-sending the model. Tables are written in sorted order, so the same data always encodes to the same bytes.

Usage

```
encode_params(params)
```

Arguments

params	Named parameter tables, as in <code>program()</code> .
--------	--

Value

A raw vector: the encoded tables.

expectation	<i>The expected value over the scenarios</i>
-------------	--

Description

Minimizing an expectation optimizes the average case and says nothing about the bad ones; use `cvar()` when the bad ones are what matter.

Usage

```
expectation(x)
```

Arguments

x	A model expression.
---	---------------------

Details

x is any expression containing a random variable. The result is deterministic, and can be used anywhere a number can. Applied to a vector expression, it aggregates each element.

Value

An expression of the same length.

expressions

quicopt expressions — model arithmetic in plain R

Description

Arithmetic on a model's variables builds an expression rather than computing a number, and comparing two expressions builds a constraint rather than answering a logical. The operators are R's own — `+` `-` `*` `/` `^`, `sqrt`, `exp`, `log`, `sin`, `cos`, `abs`, `max`, `min`, `sum`, `prod` — dispatched through the `Ops`, `Math` and `Summary` group generics, so a model reads as ordinary R code.

Details

Expressions are vectors, like everything in R: a variable declared with `n = 10` has length 10, arithmetic is elementwise, `x[3]` indexes, and `sum(x)` folds. Lengths must match exactly or be 1 (a scalar broadcasts); anything else is an error — a model is no place for silent recycling.

An operator the service does not support raises at the point of use. So do the comparisons that have no constraint counterpart: `<` and `>` (use `<=` or `>=`, which for continuous quantities mean the same thing) and `!=`.

One caveat comes with `==` building a constraint: `unique()` still works on these objects, but `%in%` and `match()` silently answer as if no two were equal — compare identity with `identical()` instead.

index_set

A named index set with concrete elements

Description

A named index set with concrete elements

Usage

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

Arguments

<code>name</code>	The set's name.
<code>elements</code>	A list of integers and strings.

Value

A plain list, with no class attribute, with the fields `name` and `elements` as given. It defines one index set of the model, and is an entry of the list a `program()` takes as `sets`.

ir *quicopt IR — a model as plain data*

Description

The form a model takes between the interface that wrote it and the service that solves it: variables, expressions and constraints, with no trace of how they were authored. `model()` and friends build one of these on the way out; `encode()` turns it into the bytes the service reads. The shape is the service's published contract — these constructors track it, they never fork it.

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

Arguments

value	A numeric constant.
name	The referenced name.
index	An index tuple (a list of integers and strings; <code>list()</code> for a scalar).
op	A catalog operator key, e.g. "+".
args	Enclosing bound indices the set is applied to (<code>list()</code> for a flat set).
idx	The bound dummy index name.
over	An <code>ir_set_ref()</code> the fold ranges across.
body	The folded expression.
cond	Keep a term only where cond is non-zero; NULL keeps every term.

Details

Nodes are plain lists tagged by a kind field. An index tuple is a plain list whose entries are integers (concrete coordinates) or strings (bound index names).

The constructors carry an `ir_` prefix rather than mirroring the Python client's bare names: `Reduce` is a base R function, and this package extends base names, it does not mask them.

Value

A plain list, with no class attribute, holding one node of a model's expression tree. `ir_const()`, `ir_param()`, `ir_var()`, `ir_apply()`, `ir_reduce()` and `ir_source_ref()` each return an expression node: its `kind` field ("const", "param", "var", "apply", "reduce" or "source") says which node it is, and the remaining fields are the arguments under their own names (value coerced to numeric). Such a node stands wherever an expression is expected: as an entry of another node's `args`, as the objective of a `program()`, or as the `f` of a `constraint()`. `ir_set_ref()` returns a list with the fields `name` and `args`: a reference to an index set rather than an expression, so it carries no `kind`, and it is what `ir_reduce()` takes as over.

job_status	<i>Poll a submitted job</i>
------------	-----------------------------

Description

`job_status` fetches the job's state (queued, running, done, failed) and its log tail. `job_result` fetches the finished solve, polling past the service's `not_done` answer until the worker finishes. `job_log` fetches the plain-text log, and `job_delete` removes the job and its stored result from the service.

Usage

```
job_status(job)

job_result(job, wait = TRUE, timeout = 120, poll = 0.5)

job_log(job)

job_delete(job)
```

Arguments

<code>job</code>	A <code>quicopt_job</code> from <code>submit()</code> .
<code>wait</code>	Poll until the job is done (TRUE), or fetch exactly once.
<code>timeout</code>	Maximum seconds to keep polling before giving up.
<code>poll</code>	Seconds between polls.

Value

`job_status` returns the service's job state as a list.
`job_result` returns the finished solve as a `quicopt_result`.
`job_log` returns the log as a single string.
`job_delete` returns NULL, invisibly.

minimize	<i>State what the model optimizes</i>
----------	---------------------------------------

Description

The objective is a single expression; fold a vector with `sum()` first. A model given no objective is a feasibility problem.

Usage

```
minimize(m, e)
```

```
maximize(m, e)
```

Arguments

m	A <code>model()</code> .
e	The objective expression.

Value

The model, invisibly.

model	<i>Create an empty optimization model</i>
-------	---

Description

A model owns decision variables, an objective, constraint rows, and — for a model under uncertainty — random variables and a scenario count. Declare variables with `num_var()`, `int_var()`, `bin_var()` and `rand_var()`, state the goal with `minimize()` or `maximize()`, add constraints with `add()`, and hand the model to `solve()`.

Usage

```
model()
```

Details

Every setter both mutates the model and returns it invisibly, so the imperative style and the pipe are the same functions:

```
m <- model()
x <- num_var(m, "x", 0, 4)           # handle style

m <- model() |>                      # pipe style; m$x retrieves the handle
  add_var("x", 0, 4)
```

Note the pipe mutates its input — the model is one shared object, not a value being copied along the chain.

Value

A model, an environment of class `quicopt_model`.

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

Description

num_var declares a continuous variable, int_var an integer one, and bin_var a binary one. With n greater than 1 the declaration is a vector variable: x[3] indexes it, sum(x) folds it, arithmetic is elementwise, and per-element bounds are given as vectors of length n. Solutions come back keyed "x" for a scalar and "x[1]", "x[2]", ... for a vector.

Usage

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

```
int_var(m, name, lower = -Inf, upper = Inf, n = 1, start = 0)
```

```
bin_var(m, name, n = 1)
```

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

Arguments

m	A <code>model()</code> .
name	The variable's name, unique within the model.
lower, upper	Bounds; <code>-Inf</code> / <code>Inf</code> leave a direction unbounded. A vector of length n sets per-element bounds.
n	How many elements the variable has.
start	The initial point handed to the solver.
domain	For <code>add_var</code> : "num", "int" or "bin".

Details

The handle is returned and is also retrievable from the model as `m$x`; the `add_var` variant returns the model instead, for pipes.

Value

The variable handle (an expression of length `n`).

`add_var` returns the model, invisibly.

parametric	<i>A random variable drawn from a distribution</i>
------------	--

Description

`head` is a catalog operator ("normal", ...) and each parameter is an ordinary deterministic expression node — so a distribution whose mean is itself a decision needs nothing the grammar does not already have.

Usage

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

Arguments

<code>head</code>	The distribution's catalog name.
<code>params</code>	A list of parameter nodes.

Value

A plain list, with no class attribute, with the fields `kind = "parametric"`, `head` and `params` as given. It declares one random variable by its distribution, and is an entry of the named list a [program\(\)](#) takes as sources; the entry's name is the name [ir_source_ref\(\)](#) refers to.

prob	<i>The probability that a comparison holds</i>
------	--

Description

The fraction of scenarios in which it does. This is what a chance constraint is built from:

Usage

```
prob(rel)
```

Arguments

rel A comparison built with $\leq$ or $\geq$.

Details

```
add(m, prob(demand - x <= 0) >= 0.9)
```

which reads as *demand is met in at least 90% of scenarios*. The line holds two comparisons, both meaningful: the one inside prob is the event being measured, the outer one is the service level demanded of it.

rel is a comparison, $a \leq b$ or $a \geq b$, with at least one side containing a random variable. An equality is refused: for a continuous quantity its probability is zero. Elementwise over vector comparisons.

Value

An expression: a probability between 0 and 1 per compared element.

program

A complete optimization model as plain data

Description

The tables keyed by index tuples are lists of entries rather than named lists, because an index tuple is not a string: params maps a table name to a list of `list(key = <index tuple>, value = <number>)` entries, indexed_sets maps a name to `list(key = ..., value = <element list>)` fibres, and fix is a list of `list(var = , index = , value = )` pins. Entry order does not matter; encoding sorts them canonically.

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()
)
```

Arguments

sets	A list of <code>index_set()</code> s.
indexed_sets	Dependent sets carried as data (see above).
params	Named parameter tables (see above).
vars	A list of <code>var_decl()</code> s.
objective	The objective expression node.
sense	"min" or "max".
constraints	A list of <code>constraint()</code> s.
fix	Per-index variable pins (see above).
scenarios	How many scenarios are drawn (at least 1).
scenario_seed	The seed they are drawn from (at least 1).
sources	Named <code>parametric()</code> / <code>empirical()</code> declarations.

Details

A model under uncertainty adds three more: the random variables it draws (sources, a named list of `parametric()` / `empirical()` declarations), how many scenarios are drawn and the seed they are drawn from. The last two are model data — they pin the sampled instance, so the same program always sees the same draws. Left at their defaults they say nothing, and the encoded bytes are those of a deterministic model.

Value

An object of class `quicopt_program`: a list with one field per argument, under the argument's name (scenarios and scenario_seed coerced to numeric). It is the complete model in the form the service reads, with nothing left to resolve: `encode()` turns it into bytes, and `solve_model()` and `submit()` accept it directly.

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

Description

The variable is not a decision: the solver is handed its value rather than choosing it, and every use of it means the same sample within a scenario. Two independent random variables are two declarations under two names.

Usage

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

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

Arguments

m	A <code>model()</code> .
name	The random variable's name, unique within the model.
dist	A <code>distribution()</code> such as <code>normal(100, 15)</code> , or an <code>empirical()</code> column holding one observed value per scenario. May be left NULL and supplied later with <code>set_distribution()</code> .

Details

A random variable takes no bounds and no domain — its distribution already says what values it takes.

Value

The random variable's handle (an expression of length 1).

`add_rand_var` returns the model, invisibly.

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

Description

Give a random variable its distribution

Usage

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

Arguments

m	A <code>model()</code> .
v	The random variable's handle, from <code>rand_var()</code> .
dist	A <code>distribution()</code> or an <code>empirical()</code> column.

Value

The model, invisibly.

set_empirical	<i>Turn observed history into a model's uncertainty</i>
---------------	---

Description

Every chosen column of a data frame becomes an `empirical()` random variable named after the column, and the number of rows becomes the model's scenario count. All columns are read at the same scenario index, so rows observed jointly stay jointly distributed — correlation in the data survives into the model.

Usage

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

Arguments

<code>m</code>	A <code>model()</code> .
<code>data</code>	A data frame of jointly observed rows.
<code>cols</code>	Which columns to use (default: all of them).

Details

A non-numeric column is an error, not a skip: a silently dropped column would leave a model that solves fine and answers the wrong question. Select with `cols` when the frame carries more than its uncertainty.

Value

The model, invisibly. The handles are retrievable as `m$<column>`.

set_scenarios	<i>Set how many scenarios are drawn, and from which seed</i>
---------------	--

Description

More scenarios estimate the true problem more closely and cost more to solve. Both settings belong to the model, not to the solve, so the same model always faces the same sample and two solves of it are comparable. Left unset, a model is solved over one scenario — unless an `empirical()` column sets the count by its own length.

Usage

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

Arguments

m	A <code>model()</code> .
n	How many scenarios to draw.
seed	The draw seed; left NULL, the current one is kept.

Details

The scenarios are drawn by the service from this seed; R's `set.seed()` plays no role. `n` and `seed` are both at least 1.

Value

The model, invisibly.

solve_model	<i>Solve a model with the Quicopt service</i>
-------------	---

Description

Encodes the model, sends it, and returns the parsed result. The first keyless call mints an API key, remembered for the rest of the R session; pass `api_key` to authenticate with a key you already hold (used as-is, never remembered).

Usage

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

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

Arguments

m	A <code>model()</code> , a <code>program()</code> , or already-encoded bytes.
base_url	The service to talk to; defaults to <code>DEFAULT_BASE_URL</code> .
api_key	A key you hold, or NULL to mint and reuse a free-tier key.
project	A project tag for per-project invoicing, or NULL.

<code>config</code>	Named list of extra query parameters; a <code>source_language</code> here overrides the automatic tag.
<code>gzip</code>	Compress the request body.
<code>timeout</code>	Seconds to wait for the service.
<code>transport</code>	The HTTP layer, replaceable for testing: a function(<code>req</code>) taking <code>list(method, url, headers, body, timeout)</code> and returning <code>list(status, headers, body)</code> .
<code>a</code>	The model (the argument is named <code>a</code> to match the <code>solve</code> generic).
<code>b</code>	Unused; supplying it is an error.
<code>...</code>	Passed on to <code>solve_model()</code> .

Details

`solve(m)` and `solve_model(m)` are the same function — the former extends the `base::solve` generic (as `solve.qr` does), the latter is the unambiguous name for pipes and for reading. For a long-running model, or a first call against a cold worker, `submit()` queues the same request asynchronously instead of blocking.

Value

The service's answer as a `quicopt_result`: a list with `status`, `objective`, `feasible`, `solution` (a named numeric vector), `model_class` (the class the service read the model as, e.g. "milp"), and the ready-to-print `display`. Printing the result prints `display`.

stochastic

Optimization under uncertainty

Description

Part of a model's data is often unknown when the decision has to be made: demand, prices, yields, arrival times. Declare that data as random variables carrying distributions, and the model is solved over a sample of scenarios drawn from them.

Details

Two rules describe the whole surface:

- A variable declared with `rand_var()` is a random variable, not a decision variable. Every use of it references the same sample.
- An expression containing a random variable is itself random, and cannot serve as an objective or a constraint until an aggregator reduces it over the scenarios: `expectation()` for the mean, `cvar()` for the tail, `prob()` for a chance constraint.

`set_scenarios()` sets how many scenarios are drawn and from which seed. Both belong to the model, so repeated solves see the same sample. The drawing happens in the service, from the model's own seed — R's `set.seed()` plays no role here.

Examples

```
## Not run:
# Order x units at 3 apiece against a demand learned later, pay 10 per unit
# of shortfall, and meet demand in at least 90% of scenarios:
m <- model()
x <- num_var(m, "x", 0, 200)
demand <- rand_var(m, "demand", normal(100, 15))
set_scenarios(m, 512, seed = 42)
minimize(m, 3 * x + 10 * expectation(max(demand - x, 0)))
add(m, prob(demand - x <= 0) >= 0.9)
solve(m)$solution

## End(Not run)
```

submit

Submit a model for asynchronous solving

Description

Queues the model and returns immediately with a job handle; the solve runs on the service while your session goes on. Await it with [job_result\(\)](#), peek with [job_status\(\)](#) or [job_log\(\)](#), and discard it with [job_delete\(\)](#). The arguments are those of [solve_model\(\)](#); the handle keeps the connection settings, so the polling calls need none of them repeated.

Usage

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

Arguments

m	A model() , a program() , or already-encoded bytes.
base_url	The service to talk to; defaults to DEFAULT_BASE_URL .
api_key	A key you hold, or NULL to mint and reuse a free-tier key.
project	A project tag for per-project invoicing, or NULL.
config	Named list of extra query parameters; a <code>source_language</code> here overrides the automatic tag.
gzip	Compress the request body.

timeout	Seconds to wait for the service.
transport	The HTTP layer, replaceable for testing: a function(req) taking list(method, url, headers, body, timeout) and returning list(status, headers, body).

Value

A quicopt_job handle.

wire	<i>quicopt wire — a program's bytes</i>
------	---

Description

`encode()` turns a `program()` into the bytes the service reads. Encoding is deterministic: the same model always produces the same bytes, and they are the bytes the service produces for that model too — checked byte for byte against committed goldens in tests/.

Index

add, 2
add(), 13
add_rand_var (rand_var), 17
add_var (num_var), 14
as_program, 3

bin_var (num_var), 14
bin_var(), 13
BINARY, 5
BINARY (CONTINUOUS), 5

consets, 3
constraint, 4
constraint(), 4, 12, 17
CONTINUOUS, 5, 5
cvar, 6
cvar(), 9, 21

DEFAULT_BASE_URL, 6, 20, 22
distribution, 7
distribution(), 18

empirical, 7
empirical(), 17–19
encode, 8
encode(), 3, 11, 17, 23
encode_params, 9
expectation, 9
expectation(), 21
expressions, 10

index_set, 10
index_set(), 17
indicator (consets), 3
indicator(), 4
int_var (num_var), 14
int_var(), 13
INTEGER, 5
INTEGER (CONTINUOUS), 5
ir, 11
ir_apply (ir), 11

ir_const (ir), 11
ir_param (ir), 11
ir_reduce (ir), 11
ir_reduce(), 12
ir_set_ref (ir), 11
ir_set_ref(), 11
ir_source_ref (ir), 11
ir_source_ref(), 15
ir_var (ir), 11
ir_var(), 4

job_delete (job_status), 12
job_delete(), 22
job_log (job_status), 12
job_log(), 22
job_result (job_status), 12
job_result(), 22
job_status, 12
job_status(), 22

maximize (minimize), 13
maximize(), 13
minimize, 13
minimize(), 13
model, 13
model(), 3, 8, 11, 13, 14, 18–20, 22

nonneg (consets), 3
nonneg(), 3, 4
normal (distribution), 7
num_var, 14
num_var(), 13

parametric, 15
parametric(), 17
prob, 15
prob(), 21
program, 16
program(), 3–5, 8–10, 12, 15, 20, 22, 23

rand_var, 17

`rand_var()`, [7](#), [13](#), [18](#), [21](#)

`set_distribution`, [18](#)
`set_distribution()`, [7](#), [18](#)
`set_empirical`, [19](#)
`set_scenarios`, [19](#)
`set_scenarios()`, [21](#)
`solve()`, [3](#), [6](#), [13](#)
`solve.quicopt_model (solve_model)`, [20](#)
`solve_model`, [20](#)
`solve_model()`, [8](#), [17](#), [21](#), [22](#)
`stochastic`, [21](#)
`submit`, [22](#)
`submit()`, [12](#), [17](#), [21](#)

`var_decl (CONTINUOUS)`, [5](#)
`var_decl()`, [17](#)

`wire`, [23](#)

`zero (consets)`, [3](#)
`zero()`, [4](#)