LINE for R

User Manual for the `linesolver` package

LINE Development Team

November 28, 2026

Contents

1 What the R edition is **2**

2 Installation **2**
2.1 Pointing the install at an engine . . . 2
2.2 Building the engine from source . . . 3
2.3 What the build leaves out . . . 3
2.4 Solvers that run external programs . . . 4
2.5 Windows and two-BLAS hazards . . . 4

3 A first model **4**

4 Building models **5**
4.1 Nodes and classes . . . 5
4.2 Distributions . . . 5
4.3 Enumerations . . . 5
4.4 Topology . . . 6

5 The portable document **6**

6 Solving **6**
6.1 The whole vocabulary, not just the tables . . . 7
6.2 Knobs . . . 7

7 Reading results **7**

8 Beyond the average tables **8**

9 Random models **8**

10 Limitations **9**

11 Where to look next **9**

1 What the R edition is

LINE exists as four codebases at functional parity: MATLAB, Java, native Python and C++. The R edition is not a fifth. `linesolver` **declares models in R and analyses them with the C++ engine, in process**, through a C application binary interface.

That distinction decides most of what follows. R owns the model description and the portable document; it owns none of the numerics. There is no R implementation of mean value analysis to disagree with the other four, and no R solver to validate separately: a result obtained in R is the C++ engine's result, reached without spawning a process.

R does	declare nodes, classes, distributions, routing; read and write the `line-model` JSON interchange; call the engine; present results as data frames
R does not	compute visit ratios, chains, the station partition, or moment fitting; implement any solver; validate a model (the engine's reader refuses by name)

Table 1: The division of labour

This manual covers the R surface. For the modelling theory, the solver catalogue and the algorithm references, read any of the four shared manuals (`LINE-user-matlab.pdf` and its siblings); they describe the same engine and the same models.

2 Installation

The package ships as a source tarball, `linesolver_<version>.0.tar.gz`, carried as `r/linesolver_<version>.0.tar.gz` inside the R archive `line-<version>-r.zip` (and inside the integrated release), published with every LINE release on the project's SourceForge site (https://sourceforge.net/projects/line-solver/files/) and linked from the R card on the home page (https://line-solver.sourceforge.net). Unzip the archive and install the tarball it carries as it is: R installs a source package only from its `.tar.gz`. The fourth component of the version is always `0`: R needs at least three components, and the suffix keeps the R package aligned with the PyPI package of the same release. It is not on CRAN. Install it from the downloaded file:

```
install.packages("linesolver_<version>.0.tar.gz", repos = NULL, type = "
    source")
```

or, from a shell, `R CMD INSTALL linesolver_<version>.0.tar.gz`.

The package has a pure-R half and a compiled half, and **an installation without the engine is a reduced package, not a broken one**. Without it, models can be built, saved and loaded, which makes R a first-class *producer* of models the other four editions solve, on a machine with no LINE build at all. With it, they can also be solved.

```
library(linesolver)
capi_available() # TRUE when the engine is linked in
capi_status() # "available", or the reason it is not
line_version() # version, ABI, and the build's optional features
```

2.1 Pointing the install at an engine

`configure` looks for an engine in a fixed order and echoes every hit into the install log:

1. `LINE_CAPI_LIB`, an explicit path. **That file or an error**, never a fallback: naming a library says which engine you mean, and quietly linking a different one would hand you numbers from a build you did not ask for. Pair it with `LINE_CAPI_INCLUDE` when the headers are elsewhere.
2. `LINE_CAPI_HOME`, a tree containing one.
3. `LINE_DEV/common`, or a `common/` or `r/capi/build` directory reachable by walking up from the package. These are guesses and fall through.
4. `pkg-config --exists linecapi`.
5. A copy bundled by an earlier install of this package.
6. Building from the bundled `r/capi` sources.

Every hit that is not the source build is **proved by running it**: `configure` compiles and executes a three-line program calling `line_capi_abi_version()` and compares the answer with the header's. A library from a different revision links perfectly well and then misbehaves at the first solve, so the version check is not a formality.

```
LINE_CAPI_LIB=/opt/line/lib/libline_capi.a \
LINE_CAPI_INCLUDE=/opt/line/include \
   R CMD INSTALL linesolver_*.tar.gz
```

2.2 Building the engine from source

Caution

Expect about half an hour and **eighteen gigabytes of resident memory**. `line_cli.cpp` is one translation unit that includes every solver header, and at `-O2` it is the optimiser rather than the parser that costs. Measured on a 20-core host with g++ 11.4: **17.9 GB peak in `cc1plus`**, roughly 27 minutes of CPU, a 93 MB object. A machine with 16 GB will not finish it.

```
cd r/capi && ./make.sh # -O2, no -march=native
cd ../linesolver
LINE_CAPI_LIB=/tmp/line-mp-build-capi/libline_capi.a \
LINE_CAPI_INCLUDE=../capi/include R CMD INSTALL .
```

Set `LINE_CAPI_BUILD=no` to decline the build and install the R-only half instead. That is a supported configuration, not a degraded one.

Tip

Reinstalling after changing the engine decision needs `--preclean`. R reuses `src/shim.o`, and this package compiles that file two ways: with the engine, or without it. A stale object survives and the package then reports no engine while `configure` says it took the library.

2.3 What the build leaves out

LAPACK comes from R, but only after a probe: the engine calls `dgeev_`, `dgees_`, `dgesvd_`, `dtrexc_` and `dtrsyl_`, and R's bundled `-lRlapack` is a documented subset, so `configure` checks for all five rather than linking and hoping.

Feature	Default	If absent
LAPACK	taken from R	the eigenvalue, SVD, Schur and Sylvester entry points refuse **by name**
HiGHS	off	the dense simplex handles every width instead of handing wide models to the sparse solver, so a wide QRF bound is slow rather than refused, and nothing labels it
SymEngine	off	the symbolic CTMC solution is unavailable

Table 2: Optional engine features, reported by `line_version()$features`

2.4 Solvers that run external programs

Four solvers are subprocess clients and need something else on the machine. The package installs none of them and reports absence as a refusal naming the missing program: `LDES` needs `common/ldes`, `JMT` a JVM and `JMT.jar` (`LINE_JMT_DIR` overrides, and `LINE_JMT_DOWNLOAD=1` opts into the engine's verified download of that one jar), `LN` with the `lqns` back end needs `lqns`, and `LQNS` needs `lqns` on a layered model and `qnsolver` for its `qns.*` methods on a flat network.

2.5 Windows and two-BLAS hazards

There is no source path on Windows: Rtools' MinGW-w64 cannot link an MSVC-built DLL without a generated import library, and the exception machinery differs, so a build that appears to succeed can unwind through the wrong runtime. Use a prebuilt Rtools-built DLL, or the pure-R half.

If a prebuilt engine was linked against a system BLAS while R links its own, the dynamic linker binds each `dgeev_` to whichever loaded first. You then get one implementation chosen by load order with the other's thread pool still running. The source path avoids it by linking R's `$(LAPACK_LIBS)`. Either way, set `OPENBLAS_NUM_THREADS=1`.

3 A first model

An M/M/1 queue with arrival rate 1 and service rate 2, built and solved:

```
library(linesolver)

model <- Network$new("M/M/1")
source <- Source$new(model, "Source")
queue <- Queue$new(model, "Queue", SchedStrategy$FCFS)
sink <- Sink$new(model, "Sink")

cls <- OpenClass$new(model, "Class1")
source$set_arrival(cls, Exp(1))
queue$set_service(cls, Exp(2))

model$link(serial_routing(source, queue, sink))

MVA$new(model)$avg_table()
# Station JobClass QLen Util RespT ResidT ArvR Tput
# 1 Source Class1 0 0 0 0 0 1
# 2 Queue Class1 1 0.5 1 1 1 1
# # method=mm1 arith=double iter=1
```

Two things in that listing are worth naming. Nodes and classes **register themselves** with the model as they are constructed, so `Source$new(model, "Source")` mutates `model` and there is nothing to assign back; this is the idiom the other four editions use and the reason the classes are R6 rather than S4. And the footer reports that the engine recognised the model and resolved to its closed form `mm1` rather than running a general algorithm.

Caution

`Network` is a reference object and deliberately not cloneable. R6 cloning is *shallow*, so a copy would share this model's node and class objects, whose back-pointers still address the original: mutating one would silently change the other. Round-trip through `save_model()`/`load_model()` for an independent copy.

4 Building models

4.1 Nodes and classes

Method spelling is snake_case throughout, one canonical form: `avg_table()`, `set_service()`, `set_arrival()`, `link()`, `set_number_of_servers()`. Python supports three spellings of each getter through an aliasing layer; that is a liability to port and the C++ facade already chose the short form for the same reason.

4.2 Distributions

The 42 distribution constructors (40 distributions and the two parameter priors `Prior` and `PriorContinuous`) are plain functions rather than classes, so the idiom is `Exp(1)` and not `Exp$new(1)`. Each returns a small object that is already the wire shape.

```
Exp(2) # rate
Erlang(2, 3) # rate, phases
HyperExp(c(0.3, 0.7), c(1, 5))
Coxian(c(1, 2), c(0.4))
Det(1.5); Gamma(2, 3); Pareto(1.5, 1)
Prior(list(Exp(1), Exp(2)), c(0.4, 0.6)) # uncertain service, for UQ
```

Caution

Several references declare service by its *mean* rather than its rate. `Exp.fitMean(0.9)` in MATLAB is `Exp(1/0.9)` here. Writing `Exp(0.9)` instead is a rate of 0.9 against an arrival rate of 1, which is an unstable model rather than a slightly wrong one.

4.3 Enumerations

Enumerations cross the wire as **names, never ordinals**, because the numeric values differ across the editions. The R value *is* the wire string, so there is no lookup table that can drift: `SchedStrategy$FCFS` is `"FCFS"`.

A typo is **refused** rather than returning `NULL`:

```
SchedStrategy$FCFSS
```

```
# Error: SchedStrategy has no member 'FCFSS'; valid names are: INF, FCFS,
    ...
```

That refusal is load-bearing. A bare list lookup returning `NULL` would build a queue with no scheduling key at all, and the model would solve as something else with nothing in the output naming the cause. The tables are `SchedStrategy` (44 members), `RoutingStrategy` (9), `DropRule` (5), `ReplacementStrategy` (7) and `ImpatienceType` (3).

4.4 Topology

`serial_routing(a, b, c, ...)` chains nodes in order and closes the cycle when the last node is not a `Sink`. Anything else uses a routing matrix:

```
P <- model$init_routing_matrix()
P$set(cls, delay, delay, 0.7)
P$set(cls, delay, queue, 0.3)
P$set(cls, queue, delay, 1.0)
model$link(P)
```

`P$set` takes either `(class, from, to, p)` or `(fromClass, toClass, from, to, p)`, matching the other editions.

5 The portable document

`save_model()` and `load_model()` read and write `line-model`, the JSON interchange shared with the MATLAB, Java, Python and C++ editions and specified by `doc/line-model.schema.json`.

```
save_model(model, "mm1.json")
txt <- save_model(model) # or as a string
again <- load_model("mm1.json")
```

This is the half that needs no engine, and it is what makes R useful on a machine with no LINE build: a document written here is solved by any of the other four editions.

Caution

Numbers cross as their **shortest round-tripping decimal**, the convention the schema states and the other writers use. The emitter is handwritten for exactly this reason: `jsonlite::toJSON(digits = NA)` is documented as maximum precision and is in fact 15 significant digits, which turns `1/3` into a different double and `0.1 + 0.2` into `0.3`. It is exact for round numbers, which is why it passes any test written with them.

`load_model()` reads a `Network`. A `LayeredNetwork` or an `Environment` document is refused by name rather than partially read; both remain solvable through the engine once built elsewhere.

6 Solving

Fifteen solvers, each the same class with a different engine name: `MVA`, `NC`, `CTMC`, `FLD`, `MAM`, `SSA`, `BA`, `AUTO`, `LDES`, `JMT`, `LN`, `ENV`, `AG`, `LQNS`, `UQ`. Constructing one runs nothing; the first request for a result runs the engine and caches it. `reset()` discards the cache.

```
s <- MVA$new(model) # or MVA$new(model, "exact")
s$avg_table() # per station and class
s$avg_sys_table() # per chain
s$avg_node_table() # includes non-station nodes
s$method_used() # what it actually resolved to
s$get_avg()$QLen # metrics as (station x class) matrices
```

6.1 The whole vocabulary, not just the tables

`avg_table()` and its siblings are conveniences. `analysis(name, ...)` reaches **every** analysis the engine serves, because the binding passes the command-line argument vector through verbatim rather than modelling it as a structure. There is nothing to widen when the engine gains a flag.

```
CTMC$new(model)$analysis("gen", cutoff = 5, raw = TRUE) # generator
MVA$new(model)$analysis("cdf-respt", raw = TRUE) # response-time CDF
```

Pass `raw = TRUE` for anything that is not a table. A generator, a state space or a set of CDF curves has no data-frame shape, and flattening one would destroy the structure the caller needs.

6.2 Knobs

Anything after the method becomes an engine flag: drop the dashes, and write dashes as underscores.

```
SSA$new(model, samples = 1e5, d = 23000) # --samples 100000 -d 23000
MVA$new(model, iter_max = 50, tol = 1e-9) # --iter_max 50 --tol 1e-09
```

Numbers are rendered so the engine can read them: `1e5` arrives as `100000` and not as `1e+05`, which its integer parsers would read as far as the `e`.

`timeout` bounds the wall-clock time of the solve, in seconds, and defaults to `Inf`, which sends no flag at all. When the budget runs out, the engine returns the partial solution it holds, `solver_meta(x)$timed_out` is `TRUE`, and R warns. A solve with no partial solution to return, such as an exact enumeration, fails instead.

```
x <- SSA$new(model, samples = 1e7, timeout = 30)$avg_table()
solver_meta(x)$timed_out # TRUE when cut short
```

7 Reading results

A result is a `data.frame` carrying the solve's provenance as attributes (`solver_meta()`): the resolved method, the arithmetic, the iteration count and any warning. Four things about it are deliberate and none is obvious.

Rows can be missing. The engine drops any (station, class) row whose six metrics are all exactly zero, so the table is **not a complete grid** and its dimensions do not imply the model's. Use `get_avg()` when you want matrices: it allocates the full grid from the model and scatters by name. `tget(tbl, "QLen", "Queue", "Class1")` reads one cell and returns zero for an absent pair, which is what an absent pair means.

NA is not zero. A `null` in a result column means the solver did not compute that metric, which `ENV` and `LN` with `lqns` legitimately do. It stays `NA`. The matrices fill a missing row with zero instead, because there the missing row *was* zero. The asymmetry looks like a defect and is not.

Printing is five significant digits per cell. This matches what the other four editions print. It is deliberately not `print(df, digits = 5)`, which computes one format for a whole column and lets a single wide value widen every cell beside it.

A nonconverged solve warns. The same model warns under Python, and silence here would let a reader conclude the R answer was the better one.

8 Beyond the average tables

```
find_solver(model) # every solver and method, AND why each refusal
model_index(model) # nodes, classes, stations, the index maps
validate_model(model) # the reader's refusal, without a solve
line_version() # version, ABI, build features
api_list() # the engine's callable algorithms
api_invoke("pfqn_ca", list(L = ..., N = c(2, 1)))
```

`find_solver()` is the one worth knowing about: it returns `solver`, `method`, `runnable`, `method_class`, `metrics` and **`reason`**. On a small open model it reports around 156 combinations with the reason each refusal was made, which is how you find out that a method needs a feature your model does not declare.

`api_list()` returns one row per function the engine registers, with an `access` column: `api` for those `api_invoke()` can call, `library` for those it refuses (the reason is in `reason`, typically an argument that is itself a function), and `model` for the `sn` and layered domains, which are reached as a solver analysis instead. `api_list()` shows the `api` rows and `api_list(NULL)` all of them.

`model_index()` answers "which node is a station", and it asks the engine rather than deciding in R. That is a LINE semantic; a second implementation of it would be a second thing to drift.

Caution

`api_invoke()` arguments are **row-major**, as JSON nested arrays are throughout this project, while R's `matrix()` is column-major. The binding converts an R matrix to that layout on the way in and back on the way out, so pass R matrices as they are. Do not transpose them yourself: the error would be silent for every symmetric matrix and wrong for every other one.

9 Random models

`NetworkGenerator` and `LayeredNetworkGenerator` draw random models to a specification. They are the R spelling of the generators the MATLAB, Java and Python editions carry, and **nothing is drawn in R**: the specification is assembled and validated here, the draw happens in the engine. A seed therefore names one model rather than one per binding.

```
g <- NetworkGenerator$new(seed = 23000)
g$set_varying_service_rates(TRUE)$set_topology("cyclic")
```

```
m <- g$generate(queues = 3, delays = 1, open_classes = 1, closed_classes =
    1)
MVA$new(m)$avg_table()
```

`$generate()` returns an ordinary `Network`, so it saves, validates and solves like any other. The draw is rejection-sampled: every model it returns has each closed chain visiting its own reference station, which is the shape the solvers accept.

```
h <- LayeredNetworkGenerator$new(seed = 23000)
h$set_host_demand_range(c(0.1, 2))$set_task_inf_probability(0.3)
lqn <- h$generate(clients = 2, levels = 2, tasks = 5, processors = 2)
lqn$save("model.lqnx")
```

A layered draw returns a `LayeredNetwork`, a holder for the `.lqnx` document with `$lqnx()`, `$name()` and `$save()`. Hand the saved file to any edition, or to the engine with `-i lqnx`. `LN`, `LQNS` and `LDES` also take it directly, `LN$new(lqn)$avg_table()`, and return one row per layered element (columns `Node`, `NodeType` and the five metrics, with `NA` where an element does not define a metric). `get_avg()` is refused there, since a layered model has no station list to scatter by.

`$spec()` shows what will be sent; every setter chains and returns the generator invisibly. Both generators need the engine and refuse by name without it, exactly as the solvers do.

Caution

Only the layered generator reproduces across editions. A seeded `LayeredNetworkGenerator` draws the same model here as `jline.gen.LayeredNetworkGenerator` does in Java. The flat `NetworkGenerator` does not: the Java reference builds its topology with a second, unseeded random source, so its own topology is not a function of the declared seed. A seeded `NetworkGenerator` reproduces *within* the R and C++ editions and agrees with the others only in distribution.

10 Limitations

- **A single long solve cannot be interrupted from R.** `R_CheckUserInterrupt()` unwinds by `longjmp`, which cannot cross C++ frames without skipping their destructors, and the engine's solver loops take no callback. The binding polls between the analyses of a comma list and at each document emission, and no further. Bound a long run with `samples`, `iter_max` or `timespan` instead of relying on being able to stop it.
- `load_model()` reads a `Network`; `LayeredNetwork` and `Environment` documents are refused by name.
- `Gamma()` here is the LINE distribution and masks `stats::Gamma()`, the GLM family. The name is kept for consistency with the other four editions and the mask is announced at attach.
- The package is not on CRAN. The obstacle is the engine, not the package: one translation unit peaks at 17.9 GB and roughly 27 minutes of CPU, against a check budget measured in minutes. The source tarball is 87 KB and `R CMD check` is clean.

11 Where to look next

`Network$new(name)`	the model container
`Source`, `Sink`	open-class arrival and departure
`Queue`, `Delay`	the stations; `Queue` takes a scheduling strategy
`set_patience()`, `set_balking()`, `set_retrial()`	impatience and retrials at a station
`model$add_region()`	a finite capacity region over one or several stations
`Router`, `ClassSwitch`, `Logger`	non-station nodes
`Fork`, `Join`	fork-join; `Join$new(model, name, fork)` pairs the two, and `set_quorum()` makes a partial join
`Cache`	a cache of `n_items` items; `set_read()`, `set_hit_class()`, `set_miss_class()` and `set_retrieval()` for delayed hits
`Place`, `Transition`	stochastic Petri nets; a `Place` becomes a fluid place with `set_fluid()`, and a `Transition` mode then carries `set_fluid_rate()`, `set_fluid_guard()`, `set_fluid_impulse()`, `set_fluid_rate_dependence()` and `set_fluid_flow_dependence()`
`OpenClass`, `ClosedClass`, `SelfLoopingClass`, `SignalClass`	job classes
`RoutingMatrix`	topology when `serial_routing` is not enough
`LayeredNetwork`	wraps an existing `.lqnx` document for `LN` and `LQNS`; R has no layered builder
`NetworkGenerator`, `LayeredNetworkGenerator`	random model generators

Table 3: The R6 classes

`vignette("getting-started")`	the first model, end to end
`vignette("installation")`	the engine, its features, the hazards
`vignette("solvers")`	solvers, analyses, reading results
`linesolver-manual.pdf`	the generated reference for every function
`LINE-user-matlab.pdf` and siblings	the modelling theory and the solver catalogue, which are the same engine
`r/README.md`	notes for anyone editing the binding itself