LINE: Queueing Analysis Algorithms

User manual for Python

Last revision: September 10, 2026

Contents

Chapter 1

Introduction

1.1 What is LINE?

LINE is an open-source software package to analyze queueing models via analytical methods and simulation. The tool aims at simplifying the computation of performance and reliability metrics in models of systems such as software applications, business processes, computer networks, and others. LINE decomposes a high-level system model into one or more stochastic models, typically extended queueing networks, that are subsequently analyzed using either numerical algorithms or simulation.

LINE supports 170+ modeling features: node types (queues, delays, caches, fork-join, routers), 39 scheduling disciplines (FCFS, LCFS, PS, priority variants, polling), distributions (Erlang, hyperexponential, MAP, MMPP, phase-type, matrix-exponential), cache replacement, finite capacity, impatience (balking, reneging), retrial queues, batch arrivals, dependencies (load, class, joint), and layered queueing network constructs.

A key feature of LINE is that it decouples the model description from the solvers used for its solution. That is, LINE implements model-to-model transformations that automatically translate the model specification into the input format (or data structure) accepted by its solutions algorithms. LINE sends models for evaluation either to native algorithms of external solvers that include Java Modelling Tools (JMT; http://jmt.sf.net) and LQNS (http://www.sce.carleton.ca/rads/lqns/). Native model solvers are instead based on formalisms and algorithms based on:

- Continuous-time Markov chains (`CTMC`)
- Discrete-event simulation (`LDES`)
- Fluid/mean-field ordinary differential equations (`FLD`)
- Matrix analytic methods (`MAM`)
- Normalizing constant analysis and generating-function asymptotics (`NC`)
- Mean-value analysis (`MVA`)
- Stochastic Simulation Algorithms (`SSA`)

Each solver encodes a general solution paradigm and can implement both exact and approximate analysis methods. For example, the `MVA` solver implements both exact mean value analysis (MVA) and approximate mean value analysis (AMVA). `NC` is named for the normalizing constant, but its remit is wider: it is also LINE's default solver for *asymptotic expansions of generating functions*, and that is where such methods belong. Most of them expand the generating function of a normalizing constant, as `pana`, `le`, `ble`, `kt`, `bkt`, `lekt`, `bk`, `gleint` and the Norlund-Rice family do; `morrison` instead expands the generating function of the balance equations themselves, of a model that has no product form and hence no normalizing constant to expand. A method is hosted by `NC` because it is a transform expansion, not because it computes a constant. The offered methods typically differ for accuracy, computational cost, and the subset of model features they support. A special solver, called `AUTO`, is supplied that provides an automated selection of the solver to use for a given model.

LINE also includes two meta-solvers that can leverage other solvers to analyze composite models such as layered queueing networks or queueing networks in random environments. The native meta-solvers include:

- Random environment solver (`ENV`)
- Layered network meta-solver (`LN`).

Figure 1.1 summarizes how these components are layered internally. A high-level system model is first handled by a meta-solver, when composite, and then dispatched to one of the network solvers. Each solver exposes an analyzer that selects a concrete solution method, which in turn is built on top of a set of reusable numerical APIs.

LINE can be applied to models specified in the following formats:

- LINE *modeling language*. This is a domain-specific object-oriented language designed to resemble the abstractions available in JMT's queueing network simulator (JSIM). This requires the model direct coding of the system model.
- *Layered queueing network models (LQNS XML format).* LINE is able to solve a sub-class of layered queueing network models, either specified using the LINE modeling language or according to the XML metamodel of the LQNS solver.
- *JMT simulation models (JSIMg, JSIMw formats).* LINE is able to import and solve queueing network models specified using JSIMgraph and JSIMwiz. LINE models can be exported to, and visualized with, JSIMgraph and JSIMwiz.
- *JSON model serialization.* LINE also accepts models supplied in a portable JSON format that conforms to the `line-model.schema.json` specification, enabling model interchange across codebases and with external tools, as described in Section D.2.

1.2 Network solvers

Solvers analyze objects of class to return average, transient, distributions, or state probability metrics. A solver can implement one or more *methods*, which although featuring a similar overall solution strategy, they

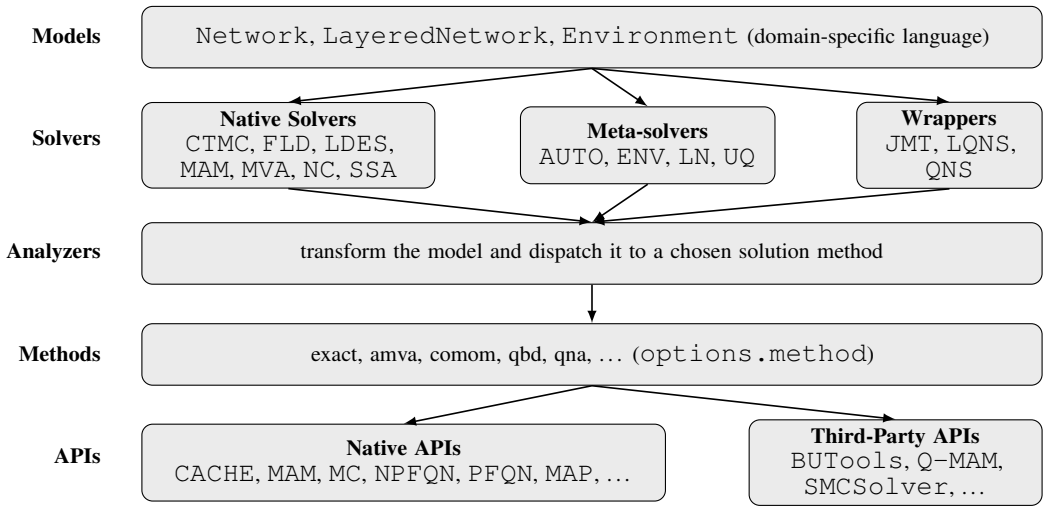


Figure 1.1: Layered internal architecture of LINE. A model is evaluated by a native solver, by a meta-solver that coordinates other solvers, or handed to an external tool through a solver wrapper; each solver runs an analyzer that selects a solution method, and methods are implemented on top of shared numerical APIs.

can differ significantly from each other in the way this strategy is actually implemented and on whether the final solution is exact or approximate.

A 'method' flag can be passed upon invoking a solver to specify the solution method that should be used. For example, the following invocations are identical:

```
MVA(model, method='exact').avg_table()
```

In what follows, we describe the general characteristics and supported model features for each solver available in LINE and their methods.

Available solvers

The following solvers are available within LINE 3.0.x:

- `AUTO`: This solver uses an algorithm to select the best solution method for the model under consideration, among those offered by the other solvers. Analytical solvers are always preferred to simulation-based solvers. This solver is implemented by the `AUTO` class.
- `CTMC`: This is a solver that returns the exact values of the performance metrics by explicit generation of the continuous-time Markov chain (CTMC) underpinning the model. As the CTMC typically incurs state-space explosion, this solver can successfully analyze only small models. The CTMC solver is the only method offered within LINE that can return an exact solution on all Markovian models, all other solvers are either approximate or are simulators. This solver is implemented by the `CTMC` class.

- `FLD`: This solver analyzes the model by means of an approximate fluid model, leveraging a representation of the queueing network as a system of ordinary differential equations (ODEs). The fluid model is approximate, but if the servers are all PS or INF, it can be shown to become exact in the limit where the number of users and the number of servers in each node grow to infinity [120]. This solver is implemented by the `FLD` class.
- `JMT`: This is a solver that uses a model-to-model transformation to export the LINE representation into a JMT simulation (JSIM) or analytical (JMVA) models [11]. The JSIM simulation solver can analyze also non-Markovian models, in particular those involving deterministic or Pareto distributions, or empirical traces. This solver is implemented by the `JMT` class.
- `LDES`: This is LINE's own discrete-event simulator, a native simulation solver built on the SSJ library. Like `JMT`, it analyzes the model by discrete-event simulation and supports non-Markovian distributions and a broad range of scheduling, routing, and blocking features, but it runs in-process (or through a standalone native binary) without requiring an external tool. This solver is implemented by the `LDES` class.
- `LQNS`: This is a wrapper for the `lqns` analytical solver and the `lqsim` simulator distributed with the LQNS suite, which solve layered queueing networks. The model is exported to the LQNS XML format, the back-end tool is run as an external process, and its output is parsed back into the LINE result tables; both commands must therefore be available on the system path. It accepts `LayeredNetwork` models only, and is the external counterpart of LINE's own layered solver, `LN`. This solver is implemented by the `LQNS` class.
- `MAM`: This is a matrix-analytic method solver, which relies on quasi-birth death (QBD) processes to analyze open queueing systems. This solver is implemented by the `MAM` class.
- `MVA`: This is a solver based on approximate and exact mean-value analysis. This solver is typically the fastest and offers very good accuracy in a number of situations, in particular models where stations have a single-server. Besides mean-value analysis, the `MVA` solver also hosts LINE's closed-form analytical decomposition methods for open networks, namely the Queueing Network Analyzer (`qna`) [161] and its index-of-dispersion counterpart, the Robust Queueing Network Analyzer (`rqna`) [162], which characterizes each flow by its full index of dispersion for counts to capture bursty, non-renewal traffic and bounds the mean workload at each queue by robust queueing. In the near-immediate feedback elimination of `rqna`, our implementation keeps the strictly higher-utilization stations as explicit queues and collapses only the equal- or lower-utilization ones, so that the network traffic rates and the variability injected by upstream bottlenecks are preserved. This solver is implemented by the `MVA` class.
- `NC`: This solver uses a combination of methods based on the normalizing constant of state probability to solve a model. The underpinning algorithm are particularly useful to compute marginal and joint state probabilities in queueing network models. This solver is implemented by the `NC` class.
- `SSA`: This is a Stochastic Simulation Algorithms based on the CTMC representation of the model. Contrary to the `JMT` simulator, which has online estimators for all the performance metrics, `SSA` estimates only the probability distribution of the system states, indirectly deriving the metrics after the simulation

is completed. Moreover, the SSA execution can more efficiently parallelized on multi-core machines. Moreover, it is possible to retrieve the evolution over time of each node state, including quantities that are not loggable in JMT, e.g., the active phase of a service or arrival distribution. This solver is implemented by the SSA class.

- QNS: This is a dedicated wrapper solver for the qnsolver utility distributed within LQNS. This allows users to evaluate product-form models using the MVA algorithms implemented within LQNS. The available options specify the multiserver handling algorithm: Conway's multiserver approximation (conway) [45], Rolia's multiserver (rolia) [132], load-dependent MVA by Reiser-Lavenberg (reiser) [127], and Zhou-Woodside's multiserver approximation (zhou) [168]. This solver is implemented by the QNS class.

1.3 Automatic solver selection

The AUTO class (aliases: SolverAuto, SolverAUTO, LINE) provides interfaces to the core solution functions (e.g., avg, ...) that dynamically bind to one of the other solvers implemented in LINE (CTMC, NC, ...). It is often difficult to identify the best solver without some performance results on the model, for example to determine if it operates in light, moderate, or heavy-load regime.

The selection is therefore driven by structural features extracted from sn: the proportions of FCFS, processor-sharing and delay stations, the presence of class-switching nodes, the mean number of servers per queue, the number of jobs per chain, and the service process types. These features are normalized and combined into ratios that a decision rule maps to a solver.

When the selected solver does not implement the operation requested (for instance get_tran_avg, which not every solver provides), AUTO falls back along the list of feasible solvers ordered by increasing average runtime. Since the list ends with the simulators, a solution strategy always exists. Direct instantiation of a solver remains preferable once the appropriate one is known, as it avoids the feature-extraction step.

Selection methods. The constructor of AUTO (AUTO) accepts an options.method argument that fixes how the back-end solver is chosen. Categorical options trigger a single back-end consistent with the requested intent:

- options.method='default' or 'heur': heuristic auto-selection. LINE instantiates a list of feasible solvers ordered by typical execution time and dispatches the requested function to the first one that supports the model (and the operation, e.g. get_tran_avg). For Network models the candidate list is, in increasing average runtime, MVA, NC, MAM, FLD, SSA, CTMC, LDES; JMT is deliberately not among the candidates, because LDES subsumes its feature set and automatic selection therefore has nothing to gain from an external simulation process (JMT remains reachable through the explicit per-solver delegation below); for LayeredNetwork models it is LQNS followed by LN wrappers around NC, MVA, MAM, FLD; for Environment models it is ENV wrapping MVA, NC, or FLD.

- `options.method='sim'`: best simulation back-end. Routes to `LDES` (`Network`, with `SSA` taking precedence for event-level sampling), `LQNS` (`LayeredNetwork`), or `ENV` wrapping `MVA` (`Environment`).
- `options.method='exact'`: best exact analytical back-end. Routes to `NC` for closed `Network` models, to `CTMC` for non-closed ones, to `LN` wrapping `NC` for layered models, and to `ENV` wrapping `NC` for random environments.
- `options.method='fast'`: fast approximate back-end (`MVA` or its `LN`/`ENV` wrapping).
- `options.method='accurate'`: accurate approximate back-end (`FLD` or its `LN`/`ENV` wrapping).
- Per-solver delegations: passing the lowercase solver name forces the corresponding back-end with its default method. Recognised values are `mam`, `mva`, `nc`, `fluid`, `jmt`, `ssa`, `ctmc`, `ldes`, `env`, `ln`, `lqns`.

1.4 Choosing the codebase that solves the model

LINE exists as four codebases – MATLAB, Java, Python and C++ – that implement the same algorithms against the same internal representation. Two of them can also hand a model to another: the model is written once and the `lang` option decides which codebase actually solves it. This is what cross-checks a result without rewriting the model, and what makes a codebase's faster or more precise engine reachable from the one the model was built in.

The option is the `lang` keyword, accepted by every solver constructor:

```
MVA(model, lang='python')    # native Python (default)
MVA(model, lang='java')      # solved by the JAR, through a subprocess
MVA(model, lang='cpp')       # solved by the C++ port, through line-cli
```

Both non-native routes use the same transport: the model is serialized to `model.json`, the other codebase's command-line front end solves it, and its JSON answer is read back into the usual result container. Nothing about the model script changes.

`lang='cpp'` requires the `line-cli` binary, located through the `LINE_CLI_BINARY` environment variable, then `PATH`, then the in-tree build directories; build it with

```
cmake -S cpp -B cpp/build -DCMAKE_BUILD_TYPE=Release && cmake --build cpp/build
```

It serves `MVA`, `NC`, `CTMC`, `MAM`, `FLD`, `SSA`, `BA` and `AUTO`; `JMT`, `LDES` and `LQNS` wrap external tools the C++ port does not carry, so they refuse it. The one failure that falls back to native Python is a missing or unrunnable binary: a construct the C++ analyzer refuses, or any other error, is raised rather than silently answered by a different engine, since `lang` is a statement about what produced the numbers. The C++ API is documented in its own manual (`LINE-user-cpp.pdf`).

1.5 Obtaining the latest release

This document contains the user manual for the latest version of LINE, which can be obtained from:

http://line-solver.sf.net/

LINE 3.0.x has been tested using Python version 3.11 or later.

Alternatively, a development version of the codebase can be cloned from Github at https://github.com/imperial-qore/line-solver.

1.6 References

To cite the LINE solver, we recommend to reference:

- G. Casale. "Integrated Performance Evaluation of Extended Queueing Network Models with LINE", in *Proc. of WSC 2020*, ACM Press, Dec 2020.

The following are references for the individual tools used by LINE. Please cite them if your use of LINE focuses on features developed in these tools:

- *BuTools* [76]: G. Horváth and M. Telek. "BuTools 2: A Rich Toolbox for Markovian Performance Evaluation", in *Proc. of VALUETOOLS*, 2017.
- *SMCSolver* [15]: D. Bini, B. Meini, S. Steffé, J. F. Pérez, and B. Van Houdt. "SMCSolver and Q-MAM: Tools for Matrix-Analytic Methods", in *SIGMETRICS Performance Evaluation Review*, 39(4), 2012.
- *MAMSolver* [128]: A. Riska and E. Smirni. "MAMSolver: A Matrix Analytic Methods Tool", in *Proc. of the 12th Int. Conf. on Modelling Techniques and Tools (TOOLS)*, LNCS 2324:205–211, 2002.
- *KPC-Toolbox* [36]: G. Casale, E. Z. Zhang, and E. Smirni. "KPC-Toolbox: Best Recipes for Automatic Trace Fitting Using Markovian Arrival Processes", in *Performance Evaluation*, 67(9):873–896, 2010.
- *JMT* [11]: M. Bertoli, G. Casale, and G. Serazzi. "The JMT Simulator for Performance Evaluation of Non-Product-Form Queueing Networks", in *Proc. of the 40th Annual Simulation Symposium (ANSS)*, 2007.
- *LQNS* [61]: G. Franks, P. Maly, M. Woodside, D. C. Petriu, A. Hubbard, and M. Mroz. "Layered Queueing Network Solver and Simulator User Manual", Carleton University, 2013.
- *Q-MAM* [15]: D. Bini, B. Meini, S. Steffé, J. F. Pérez, and B. Van Houdt. "SMCSolver and Q-MAM: Tools for Matrix-Analytic Methods", in *SIGMETRICS Performance Evaluation Review*, 39(4), 2012.

1.7 Contact and credits

LINE is the collective work of many students and researchers hosted at the QORE Lab (https://qore.doc.ic.ac.uk/) at Imperial College London. Please refer to the AUTHORS files in the codebase for detailed credits to individual project contributors.

Project coordinator: Giuliano Casale, Department of Computing, Imperial College London, 180 Queen's Gate, SW7 2AZ, London, United Kingdom. Web: http://wp.doc.ic.ac.uk/gcasale/

1.8 Copyright and license

1.9 Acknowledgement

LINE has been partially funded by the European Commission grants FP7-318484 (MODAClouds), H2020-644869 (DICE), H2020-825040 (RADON), and by the EPSRC grant EP/M009211/1 (OptiMAM).

1.10 Use of AI assistance

During the development of LINE the authors used Anthropic's Claude models, namely Opus 4.0, Opus 4.5, Opus 4.8, Opus 5, Sonnet 4.5, Sonnet 5 and Fable 5, as well as OpenAI's Codex (GPT-5.1-Codex), to assist with implementation, porting across the MATLAB, Java, Python and C++ codebases, test authoring, refactoring and documentation drafting. The tool is provided as-is, without warranty of any kind, under the terms of its license. Algorithms taken from the scientific literature have been checked against the numerical examples reported in the corresponding papers, in addition to being cross-checked against simulation and against the reference implementation. The methods themselves, and their published references, remain the work of their respective authors; model assistance concerns the realization of those methods in code, not their derivation. No such model is an author or a copyright holder of any part of LINE. The last stable LINE release developed without any such assistance is 2.0.39.

Chapter 2

Getting started

2.1 Installation and support

Quick Start: You can get started with LINE as follows:

1. Obtain the latest release
   - Stable release (zip file): https://sf.net/projects/line-solver/files/latest/download

   Ensure that the files are decompressed in the installation folder.

2. From now on, you will need to run all the commands from the `python/` folder. Install the necessary Python libraries running

   ```
   pip install -r requirements.txt
   ```

3. LINE is now ready to use. For example, you can run a basic M/M/1 model using

   ```
   python3 mm1.py
   ```

4. Jupyter notebooks are also available under the `examples` and `gettingstarted/` folders.

   To run line within your Python program, import the `line_solver` module at the beginning of the file, e.g.,

   ```
   from line_solver import *
   ```

2.1.1 Software requirements

Certain features of LINE depend on external tools and libraries. The recommended dependencies are:

- Python version 3.11 or later.
- The packages within the `requirements.txt` file, which are as follows:

```
numpy pandas scipy matplotlib websockets nbformat nbconvert
```

- Jupyter or equivalent is required to visualize the `.ipynb` examples.

Partial Java ports of these libraries have been implemented or are shipped with LINE:

- KPC-Toolbox (https://github.com/kpctoolboxteam/kpc-toolbox): version 0.3.4 or later.
- MAMSolver (https://www.cs.wm.edu/MAMSolver/): tools for matrix-analytic methods.
- M3A (https://github.com/imperial-qore/M3A): version 1.0.0.
- BuTools (https://github.com/ghorvath78/butools): version 2.0 or later.
- Q-MAM/SMC (https://win.uantwerpen.be/~vanhoudt/tools/QBDfiles.zip).

Optional dependencies recommended to utilize all features available in this edition of LINE are as follows:

- Java Modelling Tools (http://jmt.sf.net): version 1.3.0 or later, required by the `JMT` solver. The latest version is automatically downloaded at the first call of the solver.
- LQNS (https://github.com/layeredqueuing/V6): version 6.2.28 or later. System paths need to be configured such that the `lqns` and `lqnsim` solvers need are available on the command line.
- Symbolic analysis backend (https://hub.docker.com/r/imperialqore/line-sage-rest): a SageMath computer algebra system behind a small REST service, used for symbolic steady-state analysis, exact parametric sensitivities and fluid Jacobians (§10.9.2). It requires Docker and is obtained with

  ```
  docker pull imperialqore/line-sage-rest:latest
  ```

  after which LINE starts and stops the service on its own. The image is not fetched automatically on first use; the environment check described below reports whether it is present. Without it, symbolic analysis falls back on `sympy`.

2.1.2 Checking the installation

Every distribution ships an environment check that reports which of the above dependencies are reachable. It is run as a console script installed with the package:

```
line-install
```

or equivalently as a function call:

```
from line_solver import line_install
line_install()
```

The check verifies that a Java runtime is available and that the symbolic backend image of §10.9.2 can be started. A missing optional dependency is reported as a warning and not as an error, since it disables a subset of the solvers and leaves the rest of LINE usable.

2.1.3 Getting help

For discussions, bug reports, new feature requests, please create a thread on the Sourceforge forums:

- General discussion: https://sf.net/p/line-solver/discussion/help/
- Bugs and issues: https://sf.net/p/line-solver/tickets/
- Feature requests: https://sf.net/p/line-solver/feature-requests/

2.2 Getting started examples

2.2.1 Tutorials

The sixteen step-by-step tutorials that introduce the modelling language, from a single M/M/1 queue to layered, cached, random-environment, and Petri-net models, are collected in the LINE primer for this language (`LINE-primer-python.pdf`, chapter *Tutorials*). The present manual documents each construct they use in full detail, and the LINE internals manual (`LINE-internals-python.pdf`) documents the solution algorithms.

2.2.2 Model gallery

LINE includes a collection of classic, commonly occurring, queueing models under the `gallery/` folder. They include single queueing systems (e.g., $M/M/1$, $M/H_2/1$, $D/M/1$, ...), tandem queueing systems, and basic queueing networks. For example, to instantiate and estimate the mean response time for a tandem network of $M/M/1$ queues we may run

```
print(MVA(gallery_mm1_tandem(),'method','exact').avg_table())
```

Obtaining the following pandas DataFrame

```
     Station JobClass  QLen  Util  RespT  ResidT  ArvR  Tput
0  mySource  myClass  0.0000   0.0  0.0000  0.0000   0.0   1.0
1    Queue1  myClass  8.9992   0.9  8.9992  8.9992   1.0   1.0
2    Queue2  myClass  8.9992   0.9  8.9992  8.9992   1.0   1.0
```

The examples in the gallery may also be used as templates to accelerate the definition of basic models. Tutorial 10 of the primer shows a gallery instantiation of a $M/E_2/1$ queue.

Table 2.1 lists the gallery models grouped by category. Each entry is a parameter-free factory available in MATLAB (`gallery_*.m`), in the Java JAR (`Gallery.gallery_*()`), and in Python

(`from line_solver import gallery_*`). Layered models return a `LayeredNetwork`, the random-environment model returns an `Environment`, and all others return a `Network`.

Table 2.1: Model gallery entries by category.

Category	Gallery functions
Single queues	`gallery_aphm1`, `gallery_coxm1`, `gallery_detm1`, `gallery_dm1`, `gallery_erlm1`, `gallery_gamm1`, `gallery_hypm1`, `gallery_mapm1`, `gallery_merl1`, `gallery_mhyp1`, `gallery_mm1`, `gallery_mmap1`, `gallery_mpar1`, `gallery_parm1`, `gallery_replayerm1`, `gallery_um1`
Processor sharing	`gallery_erlm1_ps`, `gallery_erlm1ps`, `gallery_mm1_ps`
Multiserver	`gallery_erldk`, `gallery_hyperlk`, `gallery_mapmk`, `gallery_mdk`, `gallery_merlk`, `gallery_mhypk`, `gallery_mmapk`, `gallery_mmk`
Multiclass / priority	`gallery_mm1_multiclass`, `gallery_mm1_prio`, `gallery_mm1_ps_multiclass`, `gallery_mmap1_multiclass`
Tandem / linear	`gallery_hyphyp1_linear`, `gallery_hyphyp1_tandem`, `gallery_merl1_linear`, `gallery_merl1_tandem`, `gallery_mhyp1_linear`, `gallery_mhyp1_tandem`, `gallery_mm1_linear`, `gallery_mm1_tandem`, `gallery_mm1_tandem_multiclass`
Feedback	`gallery_mm1_feedback`, `gallery_mm1_ps_feedback`
Reentrant lines	`gallery_erlerl1`, `gallery_erlerl1_reentrant`, `gallery_erlm1_reentrant`, `gallery_hyperl1_feedback`, `gallery_hyperl1_reentrant`, `gallery_hyphyp1_reentrant`, `gallery_hypm1_reentrant`, `gallery_lukumar_reentrant`, `gallery_merl1_reentrant`, `gallery_mhyp1_reentrant`, `gallery_mm1_ps_reentrant`, `gallery_mm1_reentrant`
Closed networks	`gallery_cqn`, `gallery_cqn_multiclass`, `gallery_repairmen`
Fork-join	`gallery_fj_closed`, `gallery_fj_open`
Caches	`gallery_cache_lru`, `gallery_cache_routing`
Finite capacity	`gallery_fcr`, `gallery_mm1k`
Layered (LQN)	`gallery_lqn_basic`, `gallery_lqn_workflows`, `gallery_multitier`, `gallery_multitier_storage`
Random environment	`gallery_renv_breakdown`
Random generators	`gallery_qn_random`, `gallery_lqn_random`

The `gallery_qn_random` and `gallery_lqn_random` entries wrap the `NetworkGenerator` and `LayeredNetworkGenerator` random model generators. Both accept an optional `seed` argument (default fixed) so that repeated calls return the same model; `gallery_qn_random` uses a deterministic (cyclic) topology to keep the generated network reproducible under the seed.

2.3 Controlling verbosity

Solver verbosity may be configured at program start using, e.g.:

```
GlobalConstants.set_verbose(VerboseLevel.DEBUG)
```

The three available verbosity levels are:

- `VerboseLevel.SILENT`: Suppresses all solver output messages
- `VerboseLevel.STD`: Shows standard solver output messages (default)

- `VerboseLevel.DEBUG`: Shows detailed solver output including debug information

At `STD` verbosity each solve emits a single banner line before its result table, in the form

```
<solver> analysis [method: <requested>/<resolved>; type: <accuracy>, <randomness>; lang: ...
<language>; env: <version>] completed in <t>s.
```

Example banner lines are:

```
MVA analysis [method: default/exact; type: exact, deterministic; lang: matlab; env: ...
2025a] completed in 0.001771s.
CTMC analysis [method: cftp; type: exact, randomized; lang: python; env: 3.13.7] ...
completed in 0.041220s.
NC analysis [method: default/le; type: approximate, deterministic; lang: java; env: ...
17.0.9] completed in 0.002104s.
SSA analysis [method: serial; type: approximate, randomized; lang: matlab; env: 2025a] ...
completed in 1.204813s.
BA analysis [method: gb.upper; type: upper bound, deterministic; lang: python; env: ...
3.13.7] completed in 0.000615s.
```

Chapter 3

Basic queueing networks

Throughout this chapter, we discuss the specification of `Network` models. The chapter covers what a queueing network needs in order to be specified and solved: its nodes, its job classes, the routes the jobs take, and the distributions of their service and inter-arrival times. Kept together, these are exactly the ingredients of a product-form model, class switching included. The features that take a model out of product form are collected in Chapter 5, and the formalisms that are not queueing networks in Chapter 6. LINE currently supports open, closed, and mixed networks with non-exponential service and arrivals, and state-dependent routing. All solvers support the computation of basic performance metrics, while some more advanced features are available only in specific solvers. Each `Network` model requires in input a description of the nodes, the network topology, and the characteristics of the jobs that circulate within the network. In output, LINE returns performance and reliability metrics. Technical background on these models can be found in books such as [18, 93] or in tutorials such as [6, 92].

Figure 3.1 shows a small queueing network of the kind LINE analyzes, and names on the diagram the metrics it returns. A fixed population of N users sits at terminals: each user thinks for Z seconds, then submits a request. The request queues at the CPU, is then served by one of two disks, and on completion returns to its user, who begins thinking again. In the vocabulary used throughout this chapter, the users, the CPU and the disks are *nodes*; the CPU and the disks are *stations*, since a request spends time there waiting and in service, and the terminals are a *delay* station, where a user is never queued behind another. The requests circulating in the network are the *jobs*, and all jobs that follow the same route and service requirements form a *class*. The loop drawn by the return path makes the model *closed*: no job enters or leaves, so exactly N jobs are present at all times. Had the requests instead been generated by an external `Source` and left through a `Sink`, the model would be *open*.

3.1 Default performance metrics

The default metrics supported by all solvers are as follows:

- *Mean queue-length* (`QLen`). This is the mean number of jobs residing at a node when this is observed at a

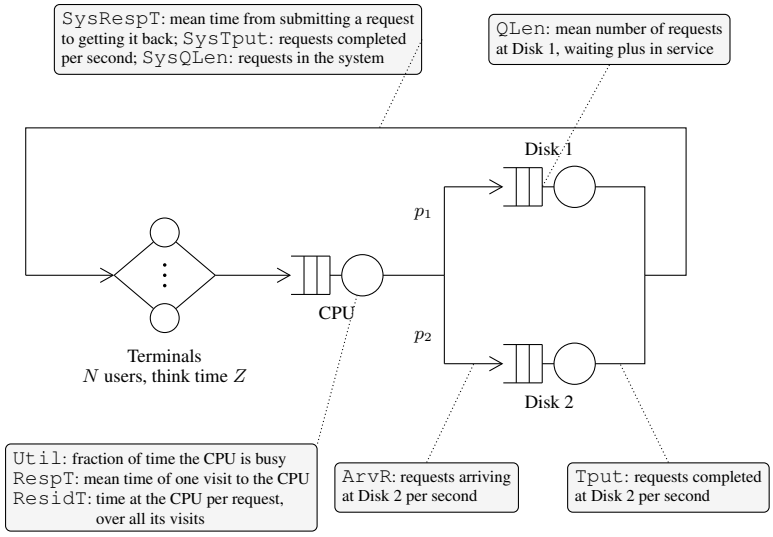


Figure 3.1: An extended queueing network: N users at terminals submit requests that are served by a CPU and by one of two disks before returning to their user. The annotations name the metrics LINE reports, per station and class for `QLen`, `Util`, `RespT`, `ResidT`, `ArvR` and `Tput`, and per chain for the system metrics.

random instant of time.

- *Mean utilization* (`Util`). For nodes that serve jobs, this is the mean fraction of time the node is busy processing jobs. In both single-server and multi-server nodes, this is a number normalized between 0 and 1, corresponding to 0% and 100%. In infinite-server nodes, the utilization is set by convention equal to the mean queue-length, therefore taking the interpretation of the mean number of jobs in execution at the station.
- *Mean response time* (`RespT`). This is the mean time a job spends traversing a node within a network. If the node is visited multiple times, the response time is the time spent for a single visit to the node.
- *Mean residence time* (`ResidT`). This is the total time a job accumulates, on average, to traverse a node within a network. If the node is visited multiple times, the residence time is the time accumulated overall visits to the node prior to returning to the reference station or arriving to a sink.
- *Mean throughput* (`Tput`). This is the mean departure rate of jobs completed at a resource per time unit. Typically, this matches the mean arrival rate, unless the node switches the class of the jobs in which case the arrival rate of a class may not match its departure rate.
- *Mean arrival rate* (`ArvR`). This is the actual rate at which jobs of a given class arrive at a station, measured in jobs per unit time. The arrival rate differs from throughput primarily in networks with class-switching mechanisms, where jobs may arrive at a node in one class and depart in a different class. In such cases, the arrival rate for a class reflects the incoming flow before any class transformation occurs, while the

throughput represents the outgoing flow after transformation. For nodes without class-switching, the arrival rate and throughput are identical under steady-state conditions due to flow balance.

The metrics above characterize individual nodes. Response times, throughputs and queue lengths are also reported end-to-end, in which case they are called *system* metrics and are indexed by chain rather than by station-class pair:

- *System response time* (`SysRespT`): time a job spends in the network, measured from its arrival at the `Source` for open chains, or from its departure from the reference station for closed chains, until it reaches the `Sink` or returns to the reference station.
- *System throughput* (`SysTput`): rate at which such end-to-end visits complete.
- *System queue-length* (`SysQLen`): total number of jobs present in the network, summed over all stations and classes.

These are obtained with the `avg_sys` method, which returns a table, or with `get_avg_sys`, which returns the same quantities as arrays.

3.2 Network object definition

3.2.1 Creating a network and its nodes

A queueing network can be described in LINE using the `Network` class constructor with a unique string identifying the model name:

```
model = Network('myModel')
```

The returned object of the `Network` class offers functions to instantiate and manage resource *nodes* (stations, delays, caches, ...) visited by jobs of several types (*classes*).

A node is a resource in the network that can be visited by a job. A node must have a unique name and can either be *stateful* or *stateless*, the latter meaning that the node does not require state variables to determine its state or actions. If jobs visiting a stateful node can be required to spend time in it, the node is also said to be a *station*. A list of nodes available in `Network` models is given in Table 3.1.

We now provide more details on each of the nodes available in `Network` models.

Queue node. A `Queue` specifies a queueing station from its name and scheduling strategy, e.g.

```
queue = Queue(model, 'Queue1', SchedStrategy.FCFS)
```

specifies a first-come, first-served queue. It is alternatively possible to instantiate a queue using the `QueueingStation` constructor, which is merely an alias for `Queue`.

Queueing stations have by default a single server. The `set_number_of_servers` method can be used to instantiate multi-server stations. For heterogeneous servers with different capabilities, use `ServerType` to define server types and `HeteroSchedPolicy` to control which server types handle which job classes.

Table 3.1: Nodes available in `Network` models.

Node	Description
`Cache`	A node to switch job classes based on hits/misses in its cache
`ClassSwitch`	A node to switch job classes based on a static probability matrix
`Delay`	A station where jobs spend time without queueing
`Fork`	A node that forks jobs into tasks
`Join`	A node that joins sibling tasks into the original job
`Logger`	A node that logs passage of jobs
`Place`	A node that holds tokens in a stochastic Petri net
`Queue`	A node where jobs queue and receive service
`Router`	A node that routes jobs to other nodes
`Sink`	Exit point for jobs in open classes
`Source`	Entry point for jobs in open classes
`Transition`	A timed or immediate node that fires tokens between places in a stochastic Petri net

Valid scheduling strategies are specified within the `SchedStrategy` static class and include: If a strategy requires class weights, these can be specified directly as an argument to the `set_service` function or using the `set_strategy_param` function, see later the description of DPS scheduling for an example.

Delay node. Delay stations, also called infinite server stations, may be instantiated either as objects of `Queue` class, with the `SchedStrategy.INF` scheduling strategy, or using the following specialized constructor

```
delay = Delay(model, 'ThinkTime')
```

As for queues, for readability it is possible to instantiate delay nodes using the `DelayStation` class which is an alias for the `Delay` class.

Source and Sink nodes. As seen in the M/M/1 getting started example, these nodes are mandatory elements for the specification of open classes. Their constructor only requires a specification of the unique name associated with the nodes:

```
source = Source(model, 'Source')
sink = Sink(model, 'Sink')
```

The `Source` node supports state-dependent arrival rates via the load-dependent scaling mechanism. Instead of specifying a fixed inter-arrival distribution, the effective arrival rate can vary as a function of the total number of jobs in the system. This is achieved by assigning a load-dependent rate function to the Source station using the `lldscaling` mechanism, in the same way load-dependent service rates are specified for queue nodes (see Section 5.2.1). When a load-dependent arrival rate is defined, the `CTMC` and `SSA` solvers automatically account for state-dependent rates during analysis, enabling accurate modeling of systems where arrival rates diminish as congestion increases (e.g., finite source populations or flow-controlled networks).

ClassSwitch node. This is a stateless node to change the class of a transiting job based on a static probabilistic policy. For example, it is possible to specify that all jobs belonging to class 1 should become of class 2 with probability 1.0, or that a transiting job of class 2 should become of class 1 with probability 0.3. This example is instantiated as follows

```
cs = ClassSwitch(model, 'ClassSwitchPoint', [[0.0, 1.0], [0.3, 0.7]])
```

Router node. This node is able to route jobs according to a specified `RoutingStrategy`, which can either be probabilistic or not (e.g., round-robin). Upon entering a `Router`, a job neither waits nor receives service; it is instead directly forwarded to the next node according to the specified routing strategy. A `Router` can be instantiated as follows:

```
router = Router(model, 'RouterNode')
```

An example of use of this node is given in the round-robin load-balancing tutorial of the LINE primer. Routing strategies need to be specified for each class using the `set_routing` method and valid choices are as follows

- Random routing (`RoutingStrategy.RAND`)
- Round robin (`RoutingStrategy.RROBIN`)
- Weighted round robin (`RoutingStrategy.WRROBIN`)
- Probabilistic routing (`RoutingStrategy.PROB`)
- Join-the-shortest-queue (`RoutingStrategy.JSQ`)
- Shortest queue of d (`RoutingStrategy.SQ`)

The state-dependent strategies (`RROBIN`, `WRROBIN`, `JSQ`, `SQ`) are supported by both `SSA` and `LDES`. `SQ` samples d candidate destinations with replacement and dispatches to the shortest among them. For `WRROBIN`, weights are supplied via additional arguments to `set_routing`; the dispatcher then cycles through a list in which each destination appears as many times as its integer weight. For example, assume that `oclass` is a class of jobs. In order to route jobs in this class with equal probabilities to every outgoing link we set

```
router.set_routing(oclass, RoutingStrategy.RAND)
```

It should be noted that `set_routing` is also available for all other nodes such as queueing stations, delays, etc. Therefore, the added value of the `Router` node is the ability to represent certain system elements that centralize the routing logic, such as load balancers.

3.2.2 Scheduling parameters

Upon setting service distributions at a station, one may also specify scheduling parameters such as weights as additional arguments to the `set_service` function. For example, if the node implements discriminatory processor sharing (`SchedStrategy.DPS`), the command

```
queue.set_service(class2, Cox2.fit_mean_and_scv(0.2,10), 5.0)
```

assigns a weight 5.0 to jobs in class 2. The default weight of a class is 1.0.

3.2.3 Job classes

Jobs travel within the network placing service demands at the stations. The demand placed by a job at a station depends on the class of the job. Jobs in *open classes* arrive from the external world and, upon completing the visit, leave the network. Jobs in *closed classes* start within the network and are forbidden to ever leave it, perpetually cycling among the nodes.

Open classes

The constructor for an open class only requires the class name and the creation of special nodes called `Source` and `Sink`

```
source = Source(model, 'Source')
sink = Sink(model, 'Sink')
```

Sources are special stations holding an infinite pool of jobs and representing the external world. Sinks are nodes that route a departing job back into this infinite pool, i.e., into the source. Note that a network can include at most a single `Source` and a single `Sink`.

Once source and sink are instantiated in the model, it is possible to instantiate open classes using

```
class1 = OpenClass(model, 'Class1')
```

LINE does not require explicitly associating source and sink with the open classes in their constructors, as this is done automatically. However, the LINE language requires explicitly creating these nodes since the routing topology needs to indicate the arrival and departure points of jobs in open classes. However, if the network does not include open classes, the user will not need to instantiate a `Source` and a `Sink`.

Closed classes

To create a closed class, we need instead to indicate the number of jobs that start in that class (e.g., 5 jobs) and the *reference station* for that class (e.g., `queue`), i.e.:

```
class2 = ClosedClass(model, 'Class2', 5, queue)
```

The reference station indicates a point in the network used to calculate certain performance indexes, called *system performance indexes*. The end-to-end response time for a job in an open class to traverse the system is an example of a system performance index (system response time). The reference station of an open class is always automatically set by LINE to be the . Conversely, the reference station needs to be indicated explicitly

in the constructor for closed classes since the point at which a class job completes execution depends on the semantics of the model.

LINE also supports a special class of jobs, called *self-looping jobs*, which perpetually loop at the reference station, remaining in their class. The following example shows the syntax to specify a self-looping job, which is identical to closed classes but there is no need later to specify routing information.

```
model = Network('model')
delay = Delay(model, 'Delay')
queue = Queue(model, 'Queue1', SchedStrategy.FCFS)
cclass = ClosedClass(model, 'Class1', 10, delay, 0)
slclass = SelfLoopingClass(model, 'SLC', 1, queue, 0)
delay.set_service(cclass, Exp(1.0))
queue.set_service(cclass, Exp(1.5))
queue.set_service(slclass, Exp(1.5))
P = model.init_routing_matrix()
P[0] = [[0.7,0.3],[1.0,0.0]]
model.link(P)
```

Note that any routing information specified for the self-looping class will be ignored.

Mixed models

LINE also accepts models where a user has instantiated both open and closed classes. The only requirement is that, if two classes communicate by means of a class-switching mechanism, then the two classes must either be all closed or all open. In other words, classes in the same chain must either be both closed or open. Furthermore, for all closed classes in the same chain, it is required for the reference station to be the same.

The `FCFSPRPRIO` (FCFS preemptive-resume with priority) strategy combines priority-based scheduling with preemptive-resume semantics. When a higher-priority job arrives at a queue using this strategy, it preempts the currently running lower-priority job. The preempted job retains its accumulated service progress and resumes from where it was interrupted once all higher-priority jobs have been served. This is suitable for modeling systems such as interrupt-driven processors or prioritized packet queues. The `FCFSPRPRIO` strategy is supported by the CTMC, LDES, JMT, and SSA solvers.

The `SRPT` (Shortest Remaining Processing Time) strategy is a preemptive, non-priority discipline that always serves the job with the least remaining service requirement. When a new job arrives with a shorter remaining time than the job currently in service, the server preempts the current job and begins serving the new arrival. SRPT is known to minimize mean response time in single-server queues. In LINE, SRPT is supported by the LDES and JMT solvers. The priority variant `SRPTPRIO` applies SRPT within priority groups: jobs are first partitioned by priority level, and within each level the SRPT discipline is used to select the next job.

```
queue_pr = Queue(model, 'PreemptQueue', SchedStrategy.FCFSPRPRIO)
queue_srpt = Queue(model, 'SRPTQueue', SchedStrategy.SRPT)
```

The FSP (Fair Sojourn Protocol) strategy [63] is a preemptive single-server discipline that ranks jobs by their *virtual processor-sharing finish time*: the time at which each job would complete under a hypothetical processor-sharing shadow running over the same set of jobs. The server serves the job whose virtual finish time is smallest.

```
queue_fsp = Queue(model, 'FSPQueue', SchedStrategy.FSP)
```

Class switching

In LINE, jobs can switch their class while they travel between nodes (including self-loops on the same node). For example, this feature can be used to model queueing properties such as re-entrant lines in which a job visiting a station a second time may require a different average service demand than at its first visit.

A chain defines the set of reachable classes for a job that starts in the given class r and over time changes class. Since class switching in LINE does not allow a closed class to become open, and vice-versa, chains can themselves be classified into *open chains* and *closed chains*, depending on the classes that compose them.

Jobs in open classes can only switch to another open class. Similarly, jobs in closed classes can only switch to a closed class. Thus, class switching from open to closed classes (or vice-versa) is forbidden. More details about class-switching are given in Section 3.2.5.

Reference station

Before we have shown that the specification of classes requires choosing a reference station. In LINE, reference stations are properties of chains, thus if two closed classes belong to the same chain they must have the same reference station. This avoids ambiguities in the definition of the completion point for jobs in a chain.

For example, the system throughput for a chain is defined as the sum of the arrival rates at the reference station for all classes in that chain. That is, the solver counts a return to the reference station as a completion of the visit to the system. In the case of open chains, the reference station is always the Source and the system throughput corresponds to the rate at which jobs arrive at the sink Sink, which may be seen as the arrival rate seen by the infinite pool of jobs in the external world. If there is no class switching, each chain contains a single class, thus per-chain and per-class performance indexes will be identical.

Reference class

Occasionally, it is possible to encounter situations where a job needs to change class while remaining inside the same station. In this case, LINE modifies the network automatically to introduce a class-switching node for the job to route out of the station and immediately return to it in the new class.

One complication of the approach is that, by departing the node and returning to it, the job visits the station one additional time, affecting the visit count to the station and therefore performance metrics such as the residence time. To cope with this issue, LINE offers a method for the class objects, called

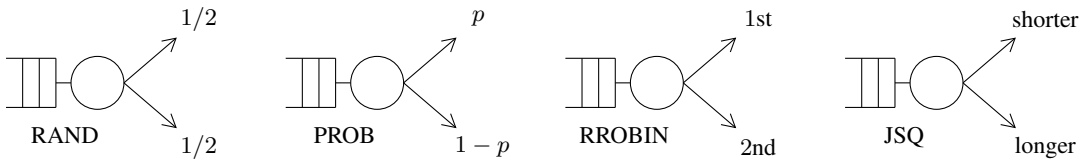


Figure 3.2: Routing strategies at a node with two outgoing links.

`set_reference_class`, that allows users to specify whether the visit of that class to the reference station should be considered upon computing the residence times across the network for the chain to which the class belongs. By default, all classes traversing the reference station are used in the visit count calculation.

3.2.4 Routing strategies

Figure 3.2 compares the strategies on a node with two outgoing links.

Probabilistic routing

Jobs travel between nodes according to the network topology and a routing strategy. Typically a queueing network will use a probabilistic routing strategy (`RoutingStrategy.PROB`), which requires specifying routing probabilities among the nodes. The simplest way to specify a large routing topology is to define the routing probability matrix for each class, followed by a call to the `link` function. This function will automatically add certain nodes to the network to ensure the correct class switching for jobs moving between stations (`ClassSwitch` elements).

In the running case, we may instantiate a routing topology as follows:

```
P = model.init_routing_matrix()
P.set(class1, class1, source, queue, 1.0)
P.set(class1, class1, queue, queue, 0.3) # self-loop with probability 0.3
P.set(class1, class1, queue, delay, 0.7)
P.set(class1, class1, delay, sink, 1.0)
P.set(class2, class2, delay, queue, 1.0) # note: closed class jobs start at delay
P.set(class2, class2, queue, delay, 1.0)
model.link(P)
```

When used as arguments to a cell array or matrix, class, and node objects will be replaced by a corresponding numerical index. Normally, the indexing of classes and nodes matches the order in which they are instantiated in the model and one can therefore specify the routing matrices using this property. In this case, we would have

```
P = model.init_routing_matrix()
P.set(class1, [[0,1,0,0], [0,0.3,0.7,0], [0,0,0,1], [0,0,0,0]])
P.set(class2, [[0,0,0,0], [0,0,1,0], [0,1,0,0], [0,0,0,0]])
```

```
model.link(P)
```

Where needed, the `get_class_index` and `get_node_index` functions return the numerical index associated with a node name, for example `model.get_node_index('Delay')`. Class and node names in a network *must be unique*. The list of names already assigned to nodes in the network can be obtained with the `get_class_names`, `get_station_names`, and `get_node_names` functions of the `Network` class.

It is also important to note that the routing matrix in the last example is specified between *nodes*, instead of between just stations or stateful nodes, which means that for example elements such as the `Sink` need to be explicitly considered in the routing matrix. The only exception is that `ClassSwitch` elements do not need to be explicitly instantiated in the routing matrix, provided that one uses the `link` function to instantiate the topology. Note that the routing matrix assigned to a model can be printed on the screen in a human-readable format using the `print_routing_matrix` function, e.g.,

```
model.print_routing_matrix()
```

prints

```
Delay [Class1] => Queue1 [Class1] : Pr=1.0
Delay [Class2] => Queue1 [Class2] : Pr=0.001
Queue1 [Class1] => Queue1 [Class1] : Pr=0.3
Queue1 [Class1] => Source [Class1] : Pr=0.7
Queue1 [Class2] => Source [Class2] : Pr=1.0
Source [Class1] => Sink [Class1] : Pr=1.0
Source [Class2] => Queue1 [Class2] : Pr=1.0
Sink [Class2] => Source [Class2] : Pr=1.0
```

Routing probabilities for Source and Sink nodes

In the presence of open classes, and in mixed models with both open and closed classes, one needs only to specify the routing probabilities *out* of the source. The probabilities out of the sink can all be set to zero for all classes and destinations (including self-loops). The solver will take care of adjusting these inputs to create a valid routing table.

Simplified definition of tandem and cyclic topologies

Tandem networks are open queueing networks with a serial topology. LINE provides functions that ease the definition of tandem networks of stations with exponential service times. For example, Tutorial 1 of the primer, on the M/M/1 queue, illustrates a simplified way to specify a serial routing topology, i.e.,

```
model.link(Network.serial_routing(source,queue,sink))
```

In a similar fashion, we can also rapidly instantiate a tandem network consisting of stations with PS and INF scheduling as follows

```
lam = [10,20]
D = [[11,12], [21,22]] # D(i,r) - class-r demand at station i (PS)
Z = [[91,92], [93,94]] # Z(i,r)  - class-r demand at station i (INF)
modelPsInf = Network.tandem_ps_inf(lam,D,Z)
```

The above snippet instantiates an open network with two queueing stations (PS), two delay stations (INF), and exponential distributions with the given inter-arrival rates and mean service times. The `Network.tandem_ps`, `Network.tandem_fcfs`, and `Network.tandem_fcfs_inf` functions provide static constructors for networks with other combinations of scheduling policies, namely only PS, only FCFS, or FCFS and INF.

A tandem network with closed classes is instead called a cyclic network. Similar to tandem networks, LINE offers a set of static constructors:

- `Network.cyclic_ps`: cyclic network of PS queues
- `Network.cyclic_ps_inf`: cyclic network of PS queues and delay stations
- `Network.cyclic_fcfs`: cyclic network of FCFS queues
- `Network.cyclic_fcfs_inf`: cyclic network of FCFS queues and delay stations

These functions only require replacing the arrival rate vector `A` by a vector `N` specifying the job populations for each of the closed classes, e.g.,

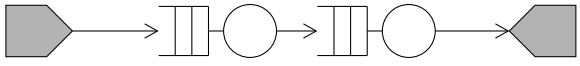
```
# Create cyclic PS network
N = [2, 1]  # job populations
D = [[11, 12], [21, 22]]  # service demands
model_ps_inf = Network.cyclic_ps(N, D)
```

Figure 3.3 shows what each constructor builds.

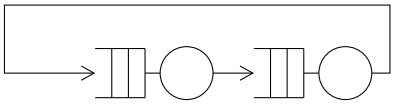
Simplified definition of cluster topologies

A cluster is a network in which jobs flow from an arrival point through a dispatcher (modelled by a `Router`) that distributes them across M parallel servers, after which they leave the system. LINE provides static constructors that build the open topology `Source` $\rightarrow$ `Dispatcher` $\rightarrow$ `Server[1..`M`]` $\rightarrow$ `Sink` and its closed counterpart `Think (Delay)` $\rightarrow$ `Dispatcher` $\rightarrow$ `Server[1..`M`]` $\rightarrow$ `Think`, as well as a mixed topology in which both families share the dispatcher and the servers. Service times at the servers are exponential and the dispatcher routes jobs according to `RoutingStrategy.RAND` by default.

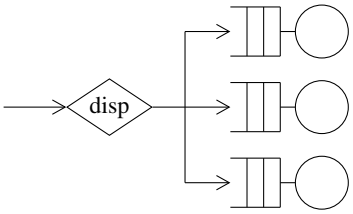
- `Network.cluster_ps`: open cluster with PS queues
- `Network.cluster_fcfs`: open cluster with FCFS queues, optionally multi-server
- `Network.cluster`: open cluster with user-supplied per-server scheduling strategies
- `Network.cluster_closed`: closed cluster with a per-class think delay
- `Network.cluster_mixed`: mixed cluster in which open and closed classes share the dispatcher and the servers, the open ones entering from the source and leaving at the sink and the closed ones cycling



(a) `tandem`



(b) `cyclic`



(c) `cluster`

Figure 3.3: The topologies produced by the simplified constructors.

through the think delay. Classes are ordered open first, so the demand matrix has $R_o + R_c$ columns For example, an open PS cluster and a closed FCFS cluster can be created as follows:

```
lam = [10, 20]
D = [[11, 12], [21, 22]]
model_open = Network.cluster_ps(lam, D)

N = [2, 1]
Z = [5, 5]
S = [2, 1]
strategy = [SchedStrategy.FCFS, SchedStrategy.FCFS]
model_closed = Network.cluster_closed(N, Z, D, strategy, S)
```

For richer configuration, LINE also exposes a chainable `line_solver.gen.Cluster` builder with setters `setDispatching`, `setScheduling`, `setStationServers`, `setClosed`, and `setMixed`, together with helpers `compareDispatching`, `compareScheduling`, `sweepArrivalRate`, and `sweepNumStations` for parametric studies. Worked examples are available under `examples/basic/cluster/` (e.g. `cl_basic`, `cl_closed`, `cl_compare`, `cl_multiclass`, `cl_sweep`).

3.2.5 Class switching

Depending on the specified probabilities, a job will be able to switch its class only among a subset of the available classes. Each subset is called a *chain*. Chains are computed in LINE as the weakly connected components of the routing probability matrix of the network when this is seen as an undirected graph. The function `model.get_chains()` produces the list of chains for the model, inclusive of a list of their composing classes.

The definition of class switching in a model is integrated in the specification of the routing between stations as described in the next subsection.

Probabilistic class switching

In models with class switching and probabilistic routing at all nodes, a routing matrix is required for each possible pair of source and target classes. For instance, suppose that in the previous example the job in the closed class `class2` switches into a new closed class (`class3`) while visiting the `queue` node. We can specify this routing strategy as follows:

```
# Set up routing matrix with class switching
P.set(class1, class1, queue, queue, 0.3)  # self-loop
P.set(class1, class1, queue, delay, 0.7)
P.set(class1, class1, delay, sink, 1.0)
P.set(class2, class3, delay, queue, 1.0)  # class switching
P.set(class3, class2, queue, delay, 1.0)
model.link(P)
```

Importantly, LINE assumes that a job switches class an instant *after* leaving a station, thus the performance metrics of a class at the node refer to the class that jobs had upon arrival to that node.

3.2.6 Service and inter-arrival time processes

A number of statistical distributions are available to specify job service times at the stations and inter-arrival times from the station. The class `Markovian` offers distributions that are analytically tractable, which are defined using absorbing Markov chains consisting of one or more states (*phases*) and called phase-type distributions. They include as special cases the distributions shown in Table 3.4.

Fitting a distribution

The `fit_mean_and_scv` function is available for all distributions that inherit from the `Markovian` class. This function provides exact or approximate matching of the first two moments, depending on the theoretical constraints imposed by the distribution. For example, an Erlang distribution with SCV=0.75 does not exist, because in a n-phase Erlang it must be SCV=$1/n$. In a case like this, `Erlang.fit_mean_and_scv(1,0.75)` will return the closest approximation, e.g., a 2-phase Erlang (SCV=0.5) with unit mean. The Erlang distribution also offers a function `fit_mean_and_order`(μ, n), which instantiates a n-phase Erlang with given mean μ.

In distributions that are uniquely determined by more than two moments, `fit_mean_and_scv` chooses a particular assignment of the residual degrees of freedom other than mean and SCV. For example, `HyperExp` depends on three parameters, therefore it is insufficient to specify mean and SCV to identify the distribution. Thus, `HyperExp.fit_mean_and_scv` automatically chooses to return a probability of selecting phase 1 equal to 0.99. Compared to other choices, this particular assignment corresponds to a higher probability mass

in the tail of the distribution. `HyperExp.fit_mean_and_scv_balanced` instead assigns p in a two-phase hyper-exponential distribution so that $p/\mu_1 = (1-p)/\mu_2$.

Some distributions admit no closed-form two-moment inverse and are fitted by an approximation, in which case the mean is matched exactly and the SCV only approximately. This is the case for `Weibull`, whose shape parameter is obtained from the Justus et al. (1976) relation $k = c^{-1.086}$ with $c = \sqrt{\mathrm{SCV}}$, the scale then being chosen so that the mean is exact. The relation was published for $1 \le k \le 10$, i.e. $\mathrm{SCV} \le 1$, where the realized SCV is within about 3% of the request; outside that range the error grows quickly, reaching 12% at $\mathrm{SCV} = 2$ and 48% at $\mathrm{SCV} = 4$. Use a phase-type distribution instead when the second moment must be matched exactly at high variability.

Three-moment fitting for APH and PH. The acyclic phase-type (`APH`) and general phase-type (`PH`) classes additionally expose static factory methods that fit a distribution from the first three moments, providing finer control over higher-order behaviour such as burstiness and skewness of the service time. For both `APH` and `PH`, three equivalent entry points are available:

- `APH.fit_central(mean,scv,skew)` fits the distribution from the first three standard moments (mean, squared coefficient of variation and skewness).
- accepts the first three central moments (mean, variance and skewness).In Python `APH.fit_central(mean,scv,skew)` takes the SCV rather than the variance, so it is the standardized-moment entry point of the first item; the central-moment convention of MATLAB and Java is available on `PH` and `Cox2`, whose `fit_central(mean,var,skew)` does take the variance.
- `APH.fit_raw_moments(m1,m2,m3)` accepts the first three raw moments $E[X^k]$ for $k = 1, 2, 3$.

Each variant returns an `APH` (resp. `PH`) instance whose parameters reproduce the requested moments, falling back to an exponential or immediate distribution when the requested mean is below `GlobalConstants.FineTol`. For example,

```
aph = APH.fit(1.0, 4.0, 8.0)
queue.set_service(class1, aph)
```

The `Coxian` class supports the same three-moment family of methods through its inherited APH interface. Both `APH` and `PH` also offer `fit_mean_and_scv(MEAN,SCV)` for two-moment fitting.

Inspecting and sampling a distribution

To verify that the fitted distribution has the expected mean and SCV it is possible to use the `get_mean` and `get_scv` functions, e.g.,

```
dist = Exp(1)
print(dist.get_mean())
print(dist.get_scv())
```

Moreover, the `sample` function can be used to generate values from the obtained distribution, e.g. we can generate 3 samples as

```
print(dist.sample(3))
```

The `eval_cdf` and `eval_cdf_interval` functions return the cumulative distribution function at the specified point or within a range, e.g.,

```
print(dist.eval_cdf_interval(2, 5))
print(dist.eval_cdf(5) - dist.eval_cdf(2))
```

For more advanced uses, the distributions of the `Markovian` class also offer the possibility to obtain the standard (D_0, D_1) representation used in the theory of Markovian arrival processes by means of the `get_representation` function [18].

3.3 Debugging and visualization

JSIMgraph is the graphical simulation environment of the JMT suite. LINE can export models to this environment for visualization purposes using the command

```
model.jsimg_view()
```

An example is shown in Figure 3.4. Using a related function, `jsimw_view`, it is also possible to export the model to the JSIMwiz environment, which offers a wizard-based interface.

Another way to debug a LINE model is to transform it into a graph object, i.e.,

```
# Display graph edges
G = model.get_graph()
print(f'Nodes: {G.nodes()}')
print(f'Edges: {G.edges(data=True)}')
```

3.3.1 TikZ export

The network topology can also be exported as LaTeX source, so that a model diagram can be included directly in a paper or a report. The TikZ export pipeline is backed by the JAR and is intentionally not available in the native Python implementation, which offers the `get_graph`/`plot` rendering described above instead. Its entry points and options are documented in the Java manual (https://line-solver.sf.net/doc/LINE-user-java.pdf).

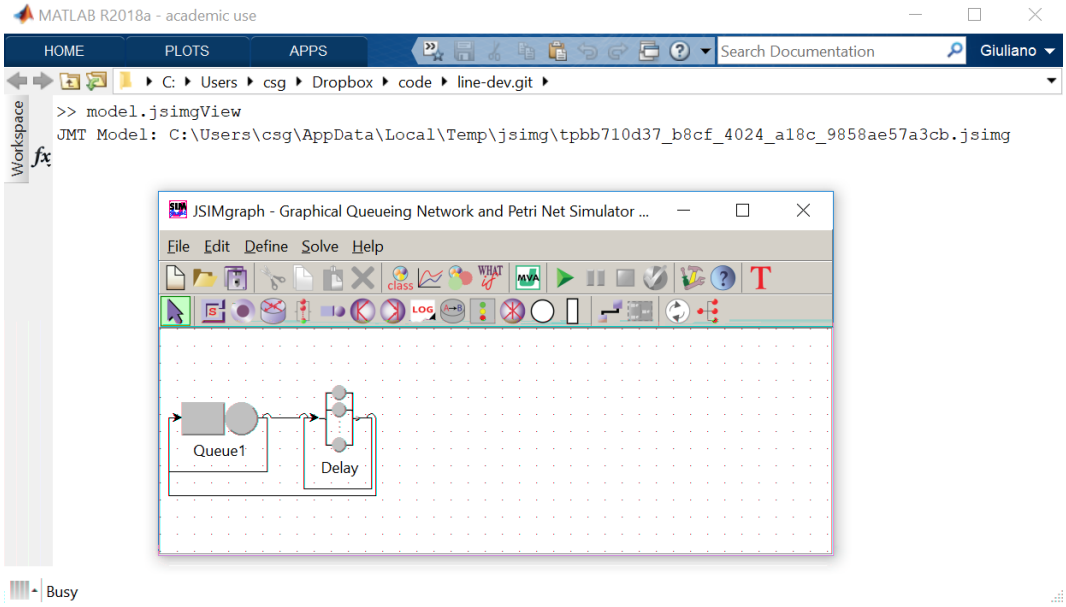


Figure 3.4: `jsimgView` function

Table 3.2: Scheduling strategies available in LINE

Strategy	**Code**	**Description**
Discriminatory processor-sharing	`DPS`	Processor sharing with class-dependent weights
Earliest due date	`EDD`	Job with earliest deadline is served first (non-preemptive)
Earliest deadline first	`EDF`	Preemptive variant of `EDD`: an arriving job with an earlier deadline preempts the in-service job
Foreground-background (Least attained service)	`FB` / `LAS`	Preemptive; serves the job with least attained service. `FB` and `LAS` are synonyms
First-come, first-served	`FCFS`	Jobs are served in order of arrival (non-preemptive)
FCFS preemptive-independent	`FCFSPI`	FCFS where a higher-priority arrival preempts the in-service job and the preempted job restarts service from scratch
FCFS preemptive-resume	`FCFSPR`	FCFS where a higher-priority arrival preempts the in-service job and the preempted job resumes from where it stopped
Fair sojourn protocol	`FSP`	Preemptive; serves the job with smallest virtual processor-sharing finish time [63]. Supported by `LDES`
Generalized processor-sharing	`GPS`	Processor sharing with configurable weights per class
Infinite-server	`INF`	All jobs receive service simultaneously (delay station)
Last-come first-serve	`LCFS`	Most recently arrived job is served first (non-preemptive)
LCFS preemptive-independent	`LCFSPI`	LCFS where an arriving job preempts the in-service job and the preempted job restarts service from scratch
LCFS preemptive-resume	`LCFSPR`	LCFS where an arriving job preempts the in-service job and the preempted job resumes from where it stopped
Longest expected processing time	`LEPT`	Job with longest expected processing time is served first
Longest job first	`LJF`	Job with longest service requirement is served first
Limited processor sharing	`LPS`	Processor sharing with limit on concurrent jobs
Longest remaining processing time	`LRPT`	Preemptive scheduling that always serves the job with the largest remaining service requirement; the dual of SRPT
Polling	`POLLING`	Server cycles through queues in round-robin fashion
Processor-sharing	`PS`	Server capacity is shared equally among all present jobs
Preemptive shortest job first	`PSJF`	Preemptive scheduling that prioritises jobs by their original service requirement (not the residual one), avoiding the constant churn of SRPT but still favouring short jobs
Shortest expected processing time	`SEPT`	Job with shortest expected processing time is served first
Shortest elapsed time first	`SETF`	Non-preemptive variant of `FB`/`LAS`: at each completion the next job in service is the one with least attained service, but it is not preempted by later-arriving jobs
Service in random order	`SIRO`	Jobs are selected randomly for service
Shortest job first	`SJF`	Job with shortest service requirement is served first

Table 3.3: Priority scheduling strategies available in LINE

Strategy	Code	Description
FCFS with priority	FCFSPRIO	FCFS within priority classes, non-preemptive. Also named HOL (head-of-line), the two being aliases
FCFS preemptive-resume with priority	FCFSPRPRIO	FCFS with preemptive-resume priority
FCFS preemptive-independent with priority	FCFSPIPRIO	FCFS with preemptive-independent priority
LCFS with priority	LCFSPRIO	LCFS within priority classes, non-preemptive
LCFS preemptive-resume with priority	LCFSPRPRIO	LCFS with preemptive-resume priority
LCFS preemptive-independent with priority	LCFSPIPRIO	LCFS with preemptive-independent priority
Processor-sharing with priority	PSPRIO	PS within priority classes
Discriminatory PS with priority	DPSPRIO	DPS within priority classes
Generalized PS with priority	GPSPRIO	GPS within priority classes
SRPT with priority	SRPTPRIO	Shortest remaining processing time within priority classes

Table 3.4: Phase-type distributions available in LINE

Distribution	**Description**
Exponential	`Exp`(λ), where λ is the rate of the exponential
n-phase Erlang	`Erlang`(α, n), where α is the rate of each of the n exponential phases
2-phase hyper-exponential	`HyperExp`$(p, \lambda_1, \lambda_2)$, that returns an exponential with rate λ_1 with probability p, and an exponential with rate λ_2 otherwise
n-phase hyper-exponential	`HyperExp`(p, λ), that builds a n-phase hyper-exponential from a rate vector $\lambda = [\lambda_1, \ldots, \lambda_n]$ and phase selection probabilities $p = [p_1, \ldots, p_n]$
2-phase Coxian	`Coxian`(μ_1, μ_2, ϕ_1), which assigns phases μ_1 and μ_2 to the two rates, and completion probability from phase 1 equal to ϕ_1 (the probability from phase 2 is $\phi_2 = 1.0$)
n-phase Coxian	`Coxian`(μ, ϕ), which builds an arbitrary Coxian distribution from a vector $\mu = [\mu_1, \ldots, \mu_n]$ of n rates and a completion probability vector $\phi = [\phi_1, \ldots, \phi_n]$ with $\phi_n = 1.0$
n-phase acyclic phase-type	`APH`(α, T), which defines an acyclic phase-type distribution with initial probability vector $\alpha = [\alpha_1, \ldots, \alpha_n]$ and transient generator T
n-phase matrix exponential	`ME`(α, A), which defines a matrix exponential distribution with initial vector $\alpha = [\alpha_1, \ldots, \alpha_n]$ (possibly non-normalized) and matrix parameter A. This generalizes phase-type distributions by allowing non-stochastic initial vectors [17]
n-phase concentrated matrix exponential	`CME`(m, n), a subclass of `ME` defining the matrix exponential distribution of odd order $n = 2k + 1$ with mean m whose squared coefficient of variation is minimal for that order [77]. Its SCV decays as $O(1/k^2)$, hence well below the bound $1/n$ attainable by a phase-type distribution of the same order, which makes it an inexpensive representation of a near-deterministic timing. Available orders are returned by `CME.getSupportedOrders()`, and `CME.fitMeanAndSCV`(m, c^2) returns the lowest order meeting a target SCV
Rational arrival process	`RAP`(H_0, H_1), which defines a correlated arrival process with hidden transition matrix H_0 and visible transition matrix H_1, where $H_0 + H_1$ forms a generator matrix. This generalizes the Markovian arrival process (MAP) [4]
Markovian arrival process	`MAP`(D_0, D_1), which defines a point process with phase-dependent arrival rates via matrices D_0 (hidden transitions) and D_1 (arrivals)
Markov-modulated Poisson process	`MMPP2`$(\lambda_1, \lambda_2, \sigma_1, \sigma_2)$, a two-phase MMPP with arrival rates λ_i and switching rates σ_i
Batch Markovian arrival process	`BMAP`(D), which generalizes MAP to allow batch arrivals via a set of arrival matrices $D = \{D_0, D_1, \ldots, D_k\}$
General phase-type	`PH`(α, T), which defines a phase-type distribution with initial probability vector α and transient generator T, allowing cyclic phase transitions
Markov-modulated deterministic process	`MMDP`(Q, R), which defines a fluid process with deterministic flow rates modulated by a background CTMC with generator Q and diagonal rate matrix R. This is the deterministic analogue of MMPP and is supported by the Markovian fluid queue solver (`FLD` with method `mfq`)
2-state Markov-modulated deterministic process	`MMDP2`$(r_0, r_1, \sigma_0, \sigma_1)$, a two-state MMDP with deterministic rates r_0 and r_1 in the two phases, and switching rates σ_0 (from state 0 to 1) and σ_1 (from state 1 to 0)
Marked Markovian arrival process	`MarkedMAP`(D, K), which defines a Markovian arrival process with K marking (color) types represented via the M3A format $D = \{D_0, D_1, D_{11}, \ldots, D_{1K}\}$. Supports multi-class arrival flows with correlated inter-arrival times and class assignment
Marked Markov-modulated Poisson process	`MarkedMMPP`(D, K), a restriction of `MarkedMAP` to diagonal D_{1k} matrices, i.e., a MMPP with K arrival classes. Represents a Poisson arrival stream partitioned into K distinguishable job types modulated by a shared background Markov chain
Discrete-time Markovian arrival process	`DMAP`(D_0, D_1), the discrete-time counterpart of `MAP`. Matrices D_0 (no-arrival transitions) and D_1 (arrival transitions) are sub-stochastic with $D_0 + D_1$ having row sums equal to 1. Useful for modelling slotted or frame-based arrival processes

Chapter 4

Basic analysis methods

The metrics defined here are the ones a queueing model is usually built to obtain: the mean number of jobs at a station, the fraction of time its servers are busy, the time a job spends there, and the rate at which work flows through it. Every solver of Section 1.2 returns them through the same handles, so the calls below are independent of the solver in use. The analyses that go beyond averages, and the ones that need a state or a sample path to be named, are deferred to Chapter 7.

4.1 Performance metrics

As discussed earlier, LINE supports a set of steady-state and transient performance metrics. Table 4.1 summarizes the definition of the associated random variables. For each metric, one or more analysis types may be available, which are extensively discussed in the next sections.

4.2 Steady-state analysis

4.2.1 Station average performance

LINE decouples network specification from its solution, allowing to evaluate the same model with multiple solvers. Model analysis is carried out in LINE according to the following general steps:

Step 1: Definition of the model. This proceeds as explained in the previous chapters.

Step 2: Instantiation of the solver(s). A solver is an instance of the `Solver` class. LINE offers multiple solvers, which can be configured through a set of common and individual solver options. For example,

```
solver = JMT(model)
```

returns a handle to a simulation-based solver based on JMT, configured with default options.

Table 4.1: Performance metrics

Metric	Acronym	Description
Queue-length	`QLen`	Number of jobs of class r (or chain-c) residing at a node i
Utilization	`Util`	Utilization of class-r (or chain-c) jobs at node i, scaled in [0,1] for multi-server nodes, equal to `QLen` at infinite server nodes
Response time	`RespT`	Time that a class-r (or chain-c) jobs spends for a single visit at node i
Residence time	`ResidT`	Cumulative time that a class-r (or chain-c) jobs spends across all visits at node i
Arrival rate	`ArvR`	Arrival rate of class-r (or chain-c) jobs at node i
Throughput	`Tput`	Throughput of class-r (or chain-c) jobs at node i
System Response time	`SysRespT`	For an open chain c, this is the time from leaving the source to arriving at the sink for *any* class in the chain. For a closed chain c, this is the interval of time between two successive visits to the reference station in any two *completing classes* within the chain.
System Throughput	`SysTput`	For an open chain c, this is the departure rate towards the sink for *any* class in the chain. For a closed chain c, this is the rate of arrival of *completing classes* in the chain at the reference station.
System queue length	`SysQLen`	Total mean number of jobs in chain c across all stations. For a closed chain, equal to the total number of jobs in the chain. For an open chain, related to `SysRespT` and `SysTput` through Little's law: $\texttt{SysQLen}(c) = \texttt{SysTput}(c) \cdot \texttt{SysRespT}(c)$.
Fork-join queue length	`FJQLen`	Number of jobs present within a fork-join scope, counting from the moment they are dispatched at the fork until synchronization is completed at the join. Measured per fork-join pair.
Fork-join response time	`FJRespT`	Time from fork dispatch to completion of synchronization at the join, measuring the end-to-end latency of the parallel section across all spawned branches.
Marginal queue-length distribution	`ProbMarg`	Probability mass function $P(n_{i,r} = k)$ of the number of class-r jobs at station i, marginalized over all other classes and stations. Returned by `get_prob_marg`.
Joint total queue-length law	`ProbSysMarg`	Joint probability $P(n_1 = k_1, \ldots, n_M = k_M)$ that every station holds a given TOTAL number of jobs, all classes summed out. Exact for closed product-form networks with load-independent single-server queues and infinite servers, and returned by `get_prob_sys_marg` under the `NC` solver only.

Step 3: Solution. Finally, this step solves the network and retrieves the concrete values for the performance indexes of interest. This may be done as follows, e.g.,

```
# QN(i,r): mean queue-length of class r at station i
QN = solver.avg_qlen()
# UN(i,r): utilization of class r at station i
UN = solver.avg_util()
# RN(i,r): mean response time of class r at station i (per visit)
RN = solver.avg_respt()
# TN(i,r): mean throughput of class r at station i
TN = solver.avg_tput()
# AN(i,r): mean arrival rate of class r at station i
AN = solver.avg_arv_r()
```

```
# WN(i,r): mean residence time of class r at station i (summed on visits)
WN = solver.avg_resid_t()
```

Alternatively, all the above metrics may be obtained in a single method call as

```
results = solver.avg()
QN = results.QN
UN = results.UN
RN = results.RN
TN = results.TN
AN = results.AN
WN = results.WN
```

In the methods above, LINE assigns station and class indexes (e.g., i, r) in order of creation in order of creation of the corresponding station and class objects. However, large models may be easier to debug by checking results using class and station names, as opposed to indexes. This can be done either by requesting LINE to build a table with the result

```
avg_table = solver.avg_table()
print(avg_table)
```

which however tends to be a rather slow data structure to use in case of repeated invocations of the solver, or by indexing the matrices returned by `avg` using the model objects. That is, if the first instantiated node is `queue` with name `MyQueue` and the second instantiated class is `cclass` with name `MyClass`, then the following commands are equivalent

```
QN[0, 1]  # 0-based indexing
QN[queue, cclass]  # object-based indexing
QN[model.get_station_index('MyQueue'), model.get_class_index('MyClass')]
```

Similar methods are defined to obtain aggregate performance metrics at chain level at each station, namely `avg_q_len_chain` for queue-lengths, `avg_util_chain` for utilizations, `avg_resp_t_chain` for response times, `avg_tput_chain` for throughputs, and the `avg_chain` method to obtain all the previous metrics.

Tabular result variants. For ease of inspection, the average results can also be retrieved as labelled tables that report station, node, class, and chain identifiers alongside the numerical values. Seven tabular variants are provided, each tailored to a different aggregation level:

- `get_avg_table`: one row per (*station*, *class*) pair, listing queue-length, utilization, response time, residence time, arrival rate, and throughput. This is the default view and matches the output of `avg`.
- `get_avg_chain_table`: one row per (*station*, *chain*) pair, with the class-level entries of a chain aggregated into the chain-level metric. Use it when the chain decomposition is more meaningful than the per-class breakdown.

- `get_avg_node_table`: one row per (*node*, *class*) pair. Unlike the station-level table it also covers non-station nodes such as `Source`, `Sink`, `Router`, `ClassSwitch`, `Fork` and `Join`.
- `get_avg_sys_table`: one row per chain, reporting the system response time `SysRespT` and system throughput `SysTput` at the reference station of the chain. It is the natural complement of `avg_sys`.
- `get_avg_cache_table`: one row per (*node*, *class*) of each `Cache` node, reporting the true-hit, delayed-hit and miss probabilities, which sum to one, plus per-list rows where an exact algorithm resolves the per-level occupancy. The `ListCost` column reports the mean storage cost held by the list, `NaN` unless item sizes were set. A delayed hit is a request served by a fetch that is already in flight; `CTMC` splits it exactly, while the simulation solvers fold it into the hit class. The `Item` column names the item a class reads in a cache network, where each item is carried by a dedicated class, and is zero otherwise (see 6.2.4).
- `get_avg_orbit_table`: one row per (*station*, *class*) pair of each retrial station, reporting the mean orbit length in the `Orbit` column. It separates the orbiting population from `QLen`, which also counts the jobs in service, and is empty for models without retrial stations (see 5.3.5).
- `get_avg_item_table`: one row per (*node*, *item*, *list*) of each `Cache` node, reporting the list capacity `ListCap`, the item storage cost `Size`, the steady-state probability `Prob` that the item resides in that list, and the expected cost `Cost`=`Size`·`Prob` it contributes there, which summed over items reproduces the list's `ListCost`. `Size` and `Cost` are `NaN` unless item sizes were set. Two further columns give the delayed-hit backlog, `DelayedHitQLen` and `DelayedHitQLenFull`, which `CTMC` computes exactly and other solvers leave as `NaN`.

Method aliases. MATLAB and Java expose progressively shorter aliases for the methods used most often during interactive analysis, all forwarding to the same canonical implementation. Table 4.2 lists the most common ones. The native Python interface (`python/`) instead exposes only the descriptive `snake_case` name (e.g., `avg_table`, `avg_qlen`), without the shorter forms. The same short-alias convention (`get`-prefix dropped, then `Table` shortened to `T`, then reduced to initials) extends to the per-chain and per-node-chain variants of Section 4 (e.g., `get_avg_qlen_chain` → `avg_qlen_chain`) and to the ensemble-solver methods used by `LN`, `UQ`, and `ENV` (e.g., `get_ensemble_avg_tables` → `ensemble_avg_tables`).

4.2.2 The utilization convention

Throughout LINE, `Util` is the mean fraction of a station's service capacity held by jobs of the given class, so that a station's total utilization lies in $[0, 1]$ irrespective of how many servers it owns. Three regimes must be distinguished, and they must not be mixed.

Load-independent stations. The Utilization Law applies,

$$U_{ir} = X_r D_{ir}/c_i = T_{ir}/(\mu_{ir} c_i), \tag{4.1}$$

Table 4.2: Common method aliases

Canonical method	Short alias(es)	Description
`get_avg_table`	`avg_table`	Average metrics table per (station, class); default view, matches `avg`
`get_avg_node_table`	`avg_node_table`	Average metrics table per (node, class); includes non-station nodes
`get_avg_chain_table`	`avg_chain_table`	Average metrics table per (station, chain)
`get_avg_node_chain_table`	`avg_node_chain_table`	Average metrics table per (node, chain)
`get_avg_sys_table`	`avg_sys_table`	System-level response time and throughput per chain
`get_avg`	`avg`	`QLen`, `Util`, `RespT`, `Tput`, `ArvR`, `ResidT` matrices per (station, class)
`get_avg_chain`	`avg_chain`	Chain-aggregated counterpart of `avg`
`get_avg_sys`	`avg_sys`	System-level counterpart of `avg`
`get_cdf_resp_t`	`cdf_resp_t`	Response time cumulative distribution function
`get_tran_avg`	`tran_avg`	Transient (time-dependent) average trajectories
`get_prob`	`prob`	Steady-state marginal state probability
`get_moment_table`	`moment_table`	Queue-length (and FCFS response-time) moments per (station, class)
`get_moment_chain_table`	`moment_chain_table`	Queue-length moments per (station, chain)
`get_moment_station_table`	`moment_station_table`	Total-queue-length moments per station
`get_sensitivity_table`	`sensitivity_table`	Derivatives of mean measures w.r.t. service rate per (station, class)
`get_avg_cache_table`	`avg_cache_table`	Cache hit/miss summary per (node, class)
`get_avg_item_table`	`avg_item_table`	Per-item cache hit probabilities
`get_avg_orbit_table`	`avg_orbit_table`	Retrial-orbit occupancy per (node, class)
`get_avg_loss_table`	`avg_loss_table`	Loss rate and loss ratio per (station, class)
`get_avg_region_loss_table`	`avg_region_loss_table`	Loss rate and loss ratio per finite-capacity region

with c_i the number of servers at station i, D_{ir} the service demand and X_r the class-r throughput. The division by c_i is what keeps a c-server station comparable to a single-server one: an M/M/3 offered $\lambda = 1.2$ with unit mean service reports $U = 0.4$, not 1.2. Infinite-server (delay) stations are the exception and report $U =$ `QLen`, the mean number of jobs in service, which may exceed one.

Load-dependent and class-dependent stations. When a station's rate varies with its occupancy (`setLoadDependence`), the capacity that normalizes the utilization is the effective one, $c_i^{eff} = \max(c_i, \max_n \alpha_i(n))$, since a load-dependent station emulates multiserver capacity while c_i remains one. A model built both ways – as a true c-server station and as a load-dependent station with $\alpha(n) = \min(n, c)$ – therefore reports the same utilization. For class-dependent rates the utilization is normalized the same way, as $U_{ir} = T_{ir}S_{ir}/p_{ir}$, where p_{ir} is the *peak rate scaling* per class that the user declares explicitly when the station is created (`setClassDependence(beta,peakRatePerClass)`, stored in the `NetworkStruct` field `cdscalingpeak`, see Section 5.2.5); a scalar peak is broadcast to every class. This makes the class-dependent utilization follow the same $T_{ir}S_{ir}/c$ convention as an ordinary multiserver station.

Order-independent and pass-and-swap stations. Here $U_{ir} = E[s_{ir}]/c_i$, the mean number of class-r jobs receiving a strictly positive rank rate. Note that s_{ir} counts *jobs*, not servers, so a job served concurrently by several compatible servers counts once; this coincides with the offered load only when a job engages a single server. `CTMC` and `LDES` observe s_{ir} directly on the ordered state, while `MVA` and `NC` recover its mean from a balanced-fairness recursion, agreeing with `CTMC` to machine precision. A multi-server station promoted internally to the `OI` representation for solution purposes (as `MVA` does for product-form multiserver queues) is still reported with the load-independent formula above; the `OI` convention applies only to stations the user declares as `OI` or `PAS`.

4.2.3 Station response time distribution

Several solvers (`CTMC`, `FLD`, `JMT`, `NC`, and `MAM`) support the computation of response time distributions for individual classes through the `get_cdf_resp_t` function. The function returns the response time distribution for every station and class. For example, the following code plots the cumulative distribution function at steady-state for class 1 jobs when they visit station 2:

```
solver = FLD(model)
fc = solver.get_cdf_resp_t()
import matplotlib.pyplot as plt
plt.plot(fc[1][0][:, 1], fc[1][0][:, 0])
plt.xlabel('t')
plt.ylabel('Pr(RespT<t)')
plt.show()
```

For the isolated single-server queue under egalitarian processor sharing, the exact sojourn time distribution [117, 165] is available separately at the API level as `qsys.qsys_mg1_ps`.

4.2.4 Station queue-length and response-time moments

For product-form networks LINE also returns the *higher moments* of the queue lengths and of the response times, exactly and without simulation, through three methods that differ only in how they group the job classes.

- `get_moment_table`: one row per station and class, reporting `QLen`, `QLenVar`, `QLenSCV` and, at FCFS stations, `RespT`, `RespTVar` and `RespTSCV`.
- `get_moment_chain_table`: one row per station and chain, for the queue length of all the classes of a chain taken together.
- `get_moment_station_table`: one row per station, for the total queue length of the station.

All three are the same computation under a different grouping of the classes, and they agree where they overlap. The optional `order` argument is a set of orders: a scalar asks for every order up to it, so `order=2` (the default) adds variances and squared coefficients of variation and `order=3` the skewness, while an explicit vector such as `[2,3]` selects exactly the orders listed.

```
solver = MVA(model)
solver.get_moment_table()
solver.get_moment_table(3)
solver.get_moment_chain_table(3)
solver.get_moment_station_table(3)
MVA(model, method='lin').get_moment_station_table(3)
```

Response-time moments are available at FCFS stations only and are `NaN` elsewhere, since the sojourn time distribution is not known in general for the other disciplines. The exact computation evaluates the MVA recursion over the whole population lattice, so its cost grows with the product of the class populations; a Linearizer-family method (`'lin'`, `'amva.lin'`, `'egflin'` or `'gflin'`) replaces it with a polynomial-time approximation of the per-station totals, so a per-chain grouping of more than one chain still requires an exact method.

On a closed single-server model `get_moment_table` also takes a `method` argument: `'exact'` always runs the recursion, and `'momlin'` always uses the moment linearizer [2], which reads the queue-length covariances off the demand derivatives of the linearized Schweitzer-Bard fixed point, at polynomial rather than exponential cost in the number of classes and with both moments carrying the AMVA error. The default runs the recursion unless the lattice is too large, in which case it falls back to the moment linearizer and says so. Queue-length moments are available for closed, mixed and open models, whereas the third moment and the response-time moments require a closed model, and a second output argument returns the full covariance matrices.

These three tables are marginal. The *joint* law of two or more queue lengths is reached from the API layer with `pfqn_qlen_joint_moments`, which returns the survival probabilities and the binomial, factorial, raw, central and cumulant moments over any chosen set of (station, class) coordinates.

4.2.5 Loss rates and loss ratios

In models where jobs can be dropped, at a station with finite capacity, blocking or reneging, the `get_avg_loss_table` method returns one row per station-class pair with the offered arrival rate (`ArvR`), the carried throughput (`Tput`), the loss rate, which is their difference, and the loss ratio, which is the loss rate as a fraction of the offered rate and, at a single finite-capacity queue, the blocking probability. Only pairs with a positive offered rate are listed, and a solver that does not model dropping reports the two rates as equal. For example, an `M/M/1/2` queue whose arrival rate is one and a half times its service rate loses about 47% of its arrivals:

```
queue.set_capacity(2)
CTMC(model).get_avg_loss_table()  # alias: a_lt, loss_avg_t
```

A quorum (partial) fork-join is the one exception to the rule that the loss rate is the difference of the two rates, because the join throughput is counted in parent units while its offered rate is in sibling units; on the join row the loss rate is therefore the directly measured sibling-discard rate.

Loss at a finite-capacity *region* boundary is reported separately by `get_avg_region_loss_table`, one row per region and class with the same columns, the offered rate being the carried throughput plus the region drop rate. `LDES` measures region drops directly, `JMT` reconstructs them by flow balance and reports class splits only approximately, and the table is empty for solvers that report neither.

4.2.6 System average performance

LINE also allows users to analyze models for end-to-end performance indexes such a system throughput or system response time. However, in models with class switching the notion of system-wide metrics can be ambiguous. For example, consider a job that enters the network in one class and departs the network in another class. In this situation one may attribute system response time to either the arriving class or the departing one, or attempt to partition it proportionally to the time spent by the job within each class. In general, the right semantics depends on the aim of the study.

LINE tackles this issue by supporting only the computation of system performance indexes *by chain*, rather than by class. Since a job that switches classes remains in the same chain by definition, the system metrics can be attributed to the chain without ambiguity. The solver functions `avg_sys` and `avg_sys_table` return system response time and system throughput per chain as observed: (i) upon arrival to the sink, for open classes; (ii) upon arrival to the reference station, for closed classes. The system queue length (`SysQLen`) is related to these two quantities through Littleś law and can be obtained as their product; for closed chains, `SysQLen` is simply equal to the fixed number of jobs in the chain.

In some cases, it is possible that a chain visits multiple times the reference station before the job completes. This also affects the definition of the system averages, since one may want to avoid counting each visit as a completion of the visit to the system. In such cases, LINE allows users to specify which classes of the chain can complete at the reference station. For example, in the code below we require that a job visits reference station 1 twice, in classes 1 and 2, but completes at the reference station only when arriving in class 2. Therefore, the system response time will be counted between successive passages in class 2.

```
class1 = ClosedClass(model, 'ClosedClass1', 1, queue, 0)
class2 = ClosedClass(model, 'ClosedClass2', 0, queue, 0)
class1.completes = False

P = model.init_routing_matrix()
P.set(class1, class1, [[0, 1], [0, 0]])
P.set(class1, class2, [[0, 0], [1, 0]])
P.set(class2, class1, [[0, 0], [1, 0]])
P.set(class2, class2, [[0, 1], [0, 0]])
model.link(P)
```

Note that the `completes` property of a class always refers to the reference station for the chain.

4.3 Controlling the analysis

The options below bound the effort an analysis may spend and fix what makes a run reproducible. They are set on the solver, either as name/value pairs in the constructor or on the `options` structure it exposes, and are shared by every solver of Chapter 1.2; the complete per-solver table is Appendix C.

4.3.1 Accuracy and cost

`options.iter_tol` sets the relative tolerance at which an iterative analysis is declared converged, and `options.iter_max` caps the number of iterations it may take; a run that reaches the cap without meeting the tolerance reports the result together with a warning, rather than failing. `options.tol` plays the same role for the numerical routines underneath. `options.timeout` bounds the wall-clock time of a single analysis and `options.cutoff` bounds the state space that an exact analysis is allowed to enumerate, by truncating the per-station queue length at the given value; raising it tightens the answer and enlarges the state space, and the analysis is refused when the resulting space does not fit the memory budget.

4.3.2 Simulation controls

When the answer is obtained by simulation, its statistical accuracy is under the analyst's control.

- `options.samples` is the sample budget of the run, measured in simulated events. The estimates converge as its square root, so a tenfold reduction of the confidence interval costs a hundredfold increase of the budget.
- `options.seed` fixes the seed of the pseudo-random number generators, so that repeated invocations return identical results. Two runs of the same model with the same seed and the same sample budget agree exactly.
- `options.confint` requests confidence intervals for the estimates, at the given confidence level, reported alongside the means.
- `options.tranfilter` selects the warm-up removal rule applied before the estimators are accumulated, so that the initial transient does not bias the steady-state estimates.
- `options.timespan` restricts the observation window, and is what turns a steady-state run into a transient one.

Two further controls change the nature of the sample path rather than its length. `options.slotted`, with `options.slotlength`, runs the model on a discrete time scale: every sampled interarrival and service time must fall on the slot lattice, a sample that does not is an error rather than being rounded, and intra-slot events are ordered by a fixed phase. Batch arrivals, declared on the source with `set_arrival_batch`, release a batch of jobs at each arrival epoch and are honoured on the sample path.

4.3.3 Warm start

An analysis may be seeded with the solution of another one, which is useful when a model is solved repeatedly along a parameter sweep, and when a cheap analysis is used to place a costly one near its fixed point. Passing a solver to `init_from_solver` seeds the initial state from that solver's steady state: an exact solver contributes the mode of its stationary distribution, an approximate one its rounded mean queue lengths, with the closed populations conserved. The same information may be supplied directly as `options.init_sol`, a station-major vector of queue lengths. A warm-started run disables warm-up removal, since its initial state is already representative.

4.3.4 Reproducibility and diagnostics

`options.verbose` controls how much the solver reports while it runs, from silent to a full trace of the analysis (Section 2.3). `options.cache` keeps the internal representation of the model between successive calls on the same solver, and `options.keep` retains the temporary files produced by an analysis, which is what makes a run reproducible outside LINE. `options.force` overrides the safety gates that refuse an analysis judged too large; it is the analyst's statement that the cost is acceptable, and it does not make the analysis cheaper.

4.3.5 Citations

The method that answers a request carries a bibliographic record with it. The method `citations` returns the references for the algorithms used in the last analysis, so that a result can be attributed also when the method was selected automatically. It is the recommended way to cite LINE results in a publication.

```
solver = MVA(model)
solver.get_avg_table()
refs = solver.citations()
```

The records themselves live in one table per codebase, keyed by method name: `matlab/src/io/line_citations.m`, `jline.io.LineCitations` in the JAR, and `line_solver.api.io.citations` in Python. Each entry carries the bibliography key as used in `doc/latex/biblio.bib`, a short author-title-venue-year reference, and a one-line `covers` hint naming the part of the solution process the paper accounts for. A method that is added or renamed gets its entry in the same change, so that `citations` stays complete.

Chapter 5

Extended queueing networks

The features in this chapter take a model out of product form: they are the reason LINE carries approximate and simulative algorithms next to the exact ones. Each of them is declared on the model exactly as the basic ones are, and each solver declares which of them it supports (Section 1.2); what changes is the cost of the solution, and whether it is exact.

5.1 Stations and buffers

5.1.1 Finite buffers

The functions `set_capacity` and `set_chain_capacity` of the `Station` class are used to place constraints on the number of jobs, total or for each chain, that can reside within a station. The capacity follows standard Kendall notation where K represents the *total system capacity* (queue + in-service jobs). For an M/M/c/K system with c servers, the buffer holds $K - c$ waiting jobs plus c jobs in service. A single class is capped with `set_class_capacity(jobclass, k)`, the station-level twin of `set_chain_capacity`. Unlike the latter, which caps every class in one call and can therefore total them into the station capacity, capping one class says nothing about the others, so the station-wide value is left untouched and the tighter of the two limits applies at analysis time. The capacity must be positive, `inf` standing for an unbounded class: a zero is how LINE internally marks a class the station does not serve at all, and so cannot also mean a buffer of size zero.

For example,

```
cqn_threeclass_hyperl()
delay.set_chain_capacity([1, 1])
model.refresh_capacity()
```

creates an example model with two chains and three classes (specified in `cqn_threeclass_hyperl.m`) and requires the second station to accept a maximum of one job in each chain. Note that if we were to ask for a higher capacity, such as `set_chain_capacity([1,7])`, which exceeds the total job population in

chain 2, LINE would have automatically reduced the value 7 to the chain 2 job population (2). This automatic correction ensures that functions that analyze the state space of the model do not generate unreachable states.

The `refresh_capacity` function updates the buffer parameterizations, performing appropriate sanity checks. Since `cqn_threeclass_hyperl` has already invoked a solver prior to our changes, the requested modifications are materially applied by LINE to the network only after calling an appropriate `refresh_struct` function, see the sensitivity analysis section. If the buffer capacity changes were made before the first solver invocation on the model, then there would not be the need for a `refresh_capacity` call, since the internal representation of the `Network` object used by the solvers is still to be created.

5.1.2 Heterogeneous servers

Queue nodes can be configured with heterogeneous servers that have distinct service rates and job class compatibilities. This feature enables modeling of systems where servers have different capabilities or speeds, such as heterogeneous CPU pools, mixed-skill service desks, or tiered processing resources. The `ServerType` class allows defining server types within a queue, each with its own characteristics. To add a server type, first create a `ServerType` object with a name and the number of servers of that type, then pass it to the `add_server_type` method. After creating server types, configure their service rates per job class using the same methods as for homogeneous queues, but targeting specific server types. Optionally, the scheduling policy for heterogeneous servers can be controlled using `HeteroSchedPolicy`, which determines how jobs are assigned to available server types. This allows specifying whether jobs can only be served by compatible server types or whether more sophisticated assignment policies are used. The following example demonstrates a queue with two server types, fast and slow servers, where each type has different service rates.

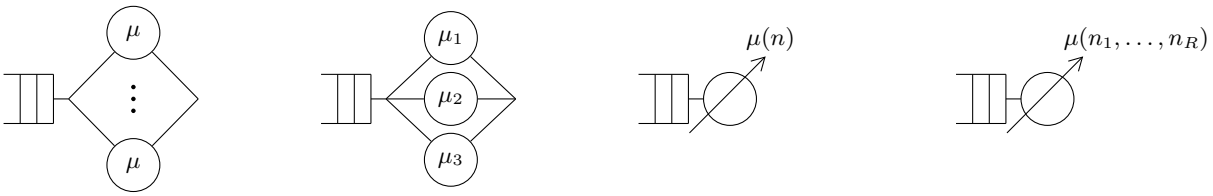
```
queue = Queue(model, 'HeteroQueue', SchedStrategy.FCFS)
fast_type = ServerType('FastServer', 2)
slow_type = ServerType('SlowServer', 2)
queue.add_server_type(fast_type)
queue.add_server_type(slow_type)
queue.set_hetero_service(jobclass, fast_type, Exp(1.0))
queue.set_hetero_service(jobclass, slow_type, Exp(0.5))
```

This capability is particularly useful for capacity planning studies where resource pools contain servers with varying performance characteristics, or for modeling systems where specialization or skill matching affects service delivery.

When a job is compatible with multiple server types, the `HeteroSchedPolicy` class determines the assignment strategy. The available policies are listed in Table 5.1. The policy is set via `queue.set_hetero_sched_policy(HeteroSchedPolicy.FSF)`. When the model is exported to JSIM XML for use with the `JMT` solver, the policy is serialised using JMT's descriptive string form (e.g. `"ALIS (Assign Longest Idle Server)"`) via `HeteroSchedPolicy.toJMTText`, so the JMT engine recognises it.

Table 5.1: Heterogeneous server assignment policies (`HeteroSchedPolicy`)

Policy	Description
`ORDER`	Assign to the first compatible server type in definition order (default)
`ALIS`	Assign Longest Idle Server: routes to the server type with the longest accumulated idle time (round-robin with busy servers placed at the back of the priority list)
`ALFS`	Assign Least Flexible Server: among compatible server types prefers the one that can serve the fewest classes (least cross-class flexibility), saving more flexible servers for classes that need them
`FAIRNESS`	Round-robin with fairness sorting: when a server type is used it is moved to the back of the priority list, so all compatible types are exercised before any one repeats
`FSF`	Fastest Server First: assigns the job to the compatible server type with the minimum expected service time
`RAIS`	Random Available Idle Server: selects uniformly at random among compatible server types that have at least one idle server



(a) homogeneous servers (b) heterogeneous servers (c) load-dependent rate (d) joint-dependent rate

Figure 5.1: (a) A station with m homogeneous servers, all with the same service rate. (b) A station with heterogeneous servers, each with its own service rate. (c) A load-dependent station, whose service rate is a function of the total number of jobs present. (d) A joint-dependent station, whose service rate is a function of the per-class populations.

Figure 5.1 contrasts homogeneous and heterogeneous servers with the load-dependent and joint-dependent rate mechanisms discussed in §5.2.4.

5.1.3 Cyclic polling

In the polling scheduling policy, the server cyclically visits the input buffer of each job class, processing jobs from that queue for a finite amount of time before switching to the next buffer. This scheduling policy further requires to specify the polling type at the station, chosen among:

- Exhaustive (`PollingType.EXHAUSTIVE`), where the server moves to the next input buffer only after finishing all jobs in the current buffer, including new arrivals during the service period;
- Gated (`PollingType.GATED`), where the number of served jobs for a buffer equals the jobs therein at the instant where the server switched to processing that buffer. Thus, newly arrived jobs during the service period will be processed at the next period.
- K-Limited (`PollingType.KLIMITED`), where the number of jobs processed is a constant K specified by the user. If the queue empties before k jobs have been served, the server will move on and attend the

next buffer.

To specify a polling type at a queue, we may write for instance

```
queue.set_polling_type(PollingType.EXHAUSTIVE)
# or for K-Limited with K=2
queue.set_polling_type(PollingType.KLIMITED, 2)
```

A polling station cannot at the same time be the blocking station of a true blocking-after-service (`BAS`) relationship: the two features both require the same per-station state slot, so a model that declares `BAS` blocking at a polling station is rejected with an explicit error rather than solved. Blocking-after-service at ordinary (non-polling) stations is fully supported.

5.1.4 Switchover times

In multiclass models, queueing stations alternate over time the processing of jobs of different classes. In some real-world situations, overheads may arise when a server needs to be reconfigured to process a different class of service. Such overheads are referred to as *switchover times*.

LINE supports switchover times in queueing stations. For example, to configure the overhead to begin processing *jobclass2* jobs after *jobclass1* we may write

```
queue.set_switchover(jobclass1, jobclass2, Exp(1))
```

A special case arises when the station uses (deterministic) polling scheduling. In this situation, the switchover time specifies the time for moving from the input buffer of class i to the input buffer of class $i+1 \bmod R$, where R is the total number of input classes to the queue. Using this fact, we need only to specify the switchover time after each class, e.g., the switchover time to begin processing *jobclass2* jobs after *jobclass1* is set as

```
queue.set_switchover(jobclass1, Exp(1))
```

5.2 Service processes

5.2.1 Load-dependent service

A queueing station i is called *load-dependent* whenever its service rate is a function of the number n_i of resident jobs at the station, summed across the ones in service and the ones in the waiting buffer. For example, a multi-server station with c identical servers, each with processing rate μ, may be shown to behave similarly to a single-server load-dependent station where the service rate is $\mu(n_i) = \mu\alpha(n_i) = \mu \min(n_i, c)$.

LINE presently supports *limited load-dependence* [29], meaning that it is possible to specify the form of the load-dependent service up to a finite range of n_i. As such, the support is currently limited to closed models, which are guaranteed to have a finite population at all times.

To specify a load-dependence service for a queueing station over the range $n_i \in [1, N]$ it is sufficient to call the `set_load_dependence` method, passing a vector of size N in its input with the scaling factor values for each n_i. For example, to instantiate a c-server node we write

```
queue.set_load_dependence(np.minimum(np.arange(1, N+1), c))
```

where the i-th element of the vector argument of `set_load_dependence` is the scaling factor $\alpha(n_i)$. It is assumed by default that $\alpha(0) = 1$. The mechanism is admitted at `PS`, `FCFS` and `DPS` stations: $\alpha(n_i)$ scales the total capacity the station clears and the scheduling discipline then divides that capacity among the classes, so the two compose without interfering.

Load-dependent closed networks may be analysed through the `NC` solver, either exactly with `exact` and `comomld`, or approximately with the saddle-point methods `nr.logit` (`nrl`), `nr.probit` (`nrp`), `nre` and `rd`, see §8.2.8. The probability metrics `get_prob`, `get_prob_aggr`, and `get_prob_norm_const_aggr` listed in Table 4.1 remain available in the load-dependent setting and are computed from the same normalising constant produced by these methods.

5.2.2 MAP and BMAP analysis

The `MAP` class exposes a number of inspection methods to assess the temporal characteristics of a Markovian arrival process. Given a `MAP` instance, the following methods are commonly used:

- `get_rate()` returns the mean arrival rate λ.
- `get_acf(lags)` returns the autocorrelation function of the inter-arrival times at the requested lags.
- `get_acf_decay()` returns the asymptotic (and optionally interpolated) decay rate of the autocorrelation function.
- `get_idc()` returns the asymptotic index of dispersion for counts; passing a timescale t returns the finite-horizon counterpart.
- `to_time_reversed()` returns the time-reversed MAP, useful in departure-process and reversed-time arguments.
- `MAP.rand(order)` produces a random MAP of the specified phase order.

For example, to inspect the burstiness of a MAP one may write

```
m = MAP.rand(3)
lam = m.get_rate()
acf = m.get_acf(list(range(1,11)))
idc = m.get_idc()
```

The counting process. The quantities above summarise the counting process $N(t)$ through its rate and its dispersion. Its distribution is available too: `mc.ctmc_saddlepoint` approximates $\Pr\{N(t) = k\}$, the probability that a MAP records exactly k events in $(0, t]$, by the saddlepoint method [50, 81]. The counting

generating function of a MAP is $\sum_{k\geq 0} P(k,t)z^k = e^{t(D_0+zD_1)}$, so its cumulant generating function is the Perron root $\eta(\theta)$ of $D_0 + e^{\theta}D_1$, and the contour integral inverting it is evaluated at the saddle θ^* solving $\eta'(\theta^*) = k/t$. Three variants are offered through the method argument: `daniels2` (the default) carries the amplitude's own curvature along the contour, `daniels` is the first-order form with the Perron amplitude, and `plain` is the bare first-order form.

The expansion parameter is the *variance* of the count, $\kappa_2 = t\,\eta''(\theta^*)$, and not its mean or the horizon: measured across the process family the relative error follows $0.083/\kappa_2$ for `daniels` and $0.017/\kappa_2^2$ for `daniels2`. Low variability shrinks κ_2 rather than breaking the method, so a renewal Erlang-r stream needs r times the horizon a Poisson one does; below $\kappa_2 = 5$ the call warns, and the per-point value is returned so that the caller can tell how far the answer can be trusted. The cost is one eigenproblem per Newton step and is independent of both t and k, and the logarithm is returned without ever forming the probability, so the accuracy holds well below the smallest positive double – on a three-phase MAP at $t = 500$, $k = 165$, where $\Pr\{N(t) = k\} \approx 1.3 \cdot 10^{-44}$, the second-order form is accurate to seven digits. This is an asymptotic method aimed at rare-event and large-deviation coefficients; for the bulk of the distribution at a moderate horizon, uniformization (`ctmc_uniformization`, `ctmc_foxglynn`) is both exact and faster.

```
p, logp, theta, info = mc.ctmc_saddlepoint(D0, D1, 200.0, 66)
p, logp, theta, info = mc.ctmc_saddlepoint(D0, D1, 100.0, range(120, 141))
```

The Batch Markovian Arrival Process `BMAP` extends `MAP` to allow batch arrivals via the cell array $D = \{D_0, D_1, \ldots, D_K\}$, where D_k is the rate matrix for transitions producing batches of size k. The most useful inspection methods are:

- `get_max_batch_size()` returns the largest batch size K supported.
- `get_mean_batch_size()` returns the average batch size $E[B]$ weighted by the batch arrival rates.
- `get_batch_rates()` returns a vector of per-batch-size arrival rates.
- `bmap.sample(n)` draws n batches, returning inter-batch times X and batch sizes B.
- `BMAP.rand(order,maxBatchSize)` generates a random BMAP of given phase order and maximum batch size.

A typical inspection of a BMAP combines its scalar summaries with sampling, e.g.,

```
bmap = BMAP.rand(2, 4)
maxK = bmap.max_batch_size
EB   = bmap.get_mean_batch_size()
```

A BMAP can also be constructed deterministically from an underlying MAP and a batch-size probability mass function via `BMAP.from_map_with_batch_pmf(D0,D1,batchSizes,pmf)`.

5.2.3 Non-Markovian distributions and moment fitting

A service or arrival process need not be given as an explicit Markovian representation: a distribution is more often specified by its moments, or drawn from a family with no phase structure at all. The distribution

Table 5.2: Non-Markovian distributions available in LINE

Distribution	Description
Deterministic	`Det`(μ) assigns probability 1.0 to the value μ
Uniform	`Uniform`(a, b) assigns uniform probability $1/(b-a)$ to the interval $[a, b]$
Gamma	`Gamma`(α, k) assigns a gamma density with shape α and scale k
Pareto	`Pareto`(α, k) assigns a Pareto density with shape α and scale k
Weibull	`Weibull`(r, α) assigns a Weibull density with shape r and scale α
Lognormal	`Lognormal`(μ, σ) assigns a lognormal density with parameters μ and σ
Zipf	`Zipf`(s, N) assigns power-law probabilities $p(k) \propto k^{-s}$ for $k = 1, \ldots, N$
Trace-based distributions (replay empirical data from files)	
Replayer	`Replayer`(filename) reads interarrival or service times cyclically from a CSV file
Trace	`Trace`(data) wraps an empirical dataset as a distribution
EmpiricalCDF	`EmpiricalCDF`(x,F) defines a distribution from CDF values F at points x

classes therefore expose fitting constructors that match a target mean, or a mean together with a squared coefficient of variation. For example, given mean $\mu = 0.2$ and squared coefficient of variation SCV=10, where SCV=variance/μ^2, we can assign to a node a 2-phase Coxian service time distribution with these moments as

```
queue.set_service(class2, Cox2.fit_mean_and_scv(0.2,10.0))
```

where `Cox2` is a static class to fit 2-phase `Coxian` distributions. Inter-arrival time distributions can be instantiated in a similar way, using `set_arrival` instead of `set_service` on the `Source` node. For example, if the `Source` is node 3 we may assign the inter-arrival times of class 2 to be exponential with mean 0.1 as follows

```
source.set_arrival(class2, Exp.fit_mean(0.1))
```

Is it also possible to plot the structure of a phase-type distribution using the `plot` instance method of the `Markovian` class.

Non-Markovian distributions are also available, but typically they can restrict the available solvers to the `JMT` simulator. They include the distributions shown in Table 5.2.

When non-Markovian distributions are used with the `CTMC`, `SSA`, or `MAM` solvers, they are automatically converted to phase-type (PH) distributions using a Bernstein polynomial approximation. A warning message is issued when this conversion occurs. The number of phases used in the approximation can be controlled via `options.config.bernstein` (default: 20 phases).

Table 5.3 shows the internal parameter storage format for each non-Markovian distribution. These parameters are stored in the `proc{ist}{r}` field of the network structure.

Lastly, we discuss two special distributions. The `Disabled` distribution can be used to explicitly forbid a class to receive service at a station. This may be useful to declare in models with sparse routing matrices to debug the model specification. Performance metrics for disabled classes will be set to `float('nan')`.

Table 5.3: Non-Markovian distribution parameter formats in `proc{ist}{r}`

Distribution	Format
Gamma	{shape (α), scale (β)}
Weibull	{shape (r), scale (α)}
Lognormal	{μ, σ}
Pareto	{shape (α), scale (k)}
Uniform	{min (a), max (b)}
Det	{t}

Conversely, the `Immediate` class can be used to specify instantaneous service (zero service time). Normally, solvers will replace zero service times with small positive values (ε =`GlobalConstants.FineTol`).

5.2.4 Class- and joint-dependent service

A generalization of load-dependent service lets the service rate depend on the per-class populations $n_i = [n_{i,1}, \ldots, n_{i,R}]$ at station i, where $n_{i,r}$ is the current number of class-r jobs. LINE exposes this through function handles supplied to the station, and distinguishes two mechanisms according to whether the resulting demands remain in product form.

Class dependence (`setClassDependence`, stored as `cdscaling`) is the *product-form* case of the QD-AMVA model [32]: the demand factorizes as $D_{i,r}(n_i) = \theta_{i,r}\,\beta_{i,r}(n_{i,r})$, where the scaling $\beta_{i,r}$ read by class r depends only on its own marginal count $n_{i,r}$ (or, equivalently, on the total n_i, which also preserves product form). The handle returns the per-class vector $[\beta_{i,1}(n_i), \ldots, \beta_{i,R}(n_i)]$, of which the solver selects class r; a scalar return is broadcast to all classes. For example

```
# Product-form class dependence: each class reads its own marginal
queue.set_class_dependence(lambda ni: [min(ni[0,0], c), 1], [c, 1])
```

gives class-1 jobs a multiserver-type scaling in their own count while class-2 jobs are always single-server.

Joint dependence (`setJointDependence`, stored as `jdscaling`) is the *non-product-form* case: the handle $\eta_i(n_i)$ may read the joint population vector arbitrarily – for instance a scalar rate driven by a foreign class marginal, $\min(n_{i,1}, c)$ shared across all classes – which violates the QD-AMVA product-form condition. Such models are solved by the same solvers as an approximation, with no exactness or uniqueness guarantee:

```
# Non-product-form joint dependence
queue.set_joint_dependence(lambda ni: min(ni[0,0], c), c)
```

In both cases the second argument is the required peak rate scaling per class (a scalar is broadcast to all classes) that normalizes utilization as $U = T\,S/\text{peak}$. The solver implicitly assumes the handles are smooth and defined also for fractional values of $n_{i,r}$.

Global dependence (`setGlobalDependence`, stored as `gdscaling`) differs from the three mechanisms above in what the handle is allowed to read. The first three are *station-local*: each receives only the population held at the station it belongs to. A global dependence instead receives the *entire* network state, the $(M \times R)$ matrix $n = (n_{i,r})$, and is declared on the model rather than on a node:

```
model.set_global_dependence(lambda n: phi(n), peak)
```

The handle returns a scalar broadcast to every station and class, an $(M \times 1)$ column read per station, or an $(M \times R)$ matrix; the service rate of class r at station i is its declared rate times $\phi_{i,r}(n)$, composing multiplicatively with any load, class or joint dependence already present. The peak argument is required and normalizes utilization exactly as above, the peaks multiplying when a station carries several of these mechanisms.

Since the Whittle balance property [163] is a property of the handle and not of the model text, it is worth checking rather than assuming: `sn_gd_balance` returns the worst relative violation over a bounded lattice, zero to rounding when ϕ is balanced. Note that any $\phi_s(n) = f(n_s)g(|n|)$ satisfies it trivially, so a probe outside that family is needed to exercise the check.

`SolverCTMC` and `SolverSSA` declare the `GlobalDependence` feature; every other solver refuses such a model rather than solving it unscaled, and the combination with a finite capacity region is refused as well. Both exploit the same observation: within one state $\phi(n)$ is a *constant*, so it need be evaluated only once per state and then multiplies the rate of every station service event there. The generator tabulates it over the enumerated state space; the simulator, which holds one state at a time, collapses that table to a single row. Within `SolverSSA` the next-reaction engine cannot carry it, because its propensity closures receive one station's population slice and never the whole matrix ϕ reads, so a model declaring a global dependence is routed to the serial engine.

A handle cannot cross a language boundary, so the JSON model interchange carries ϕ as a *table*, under the model-level key `globalDependence`. The lattice is that of the whole network state, restricted to the $(\text{station}, \text{class})$ slots a class can actually occupy: a source holds no jobs, and a class with zero per-class capacity at a station never appears there, so those entries carry no coordinate. The restriction is lossless, since no service event ever fires at a slot the class cannot occupy. A closed class is tabulated up to its own population; an open one up to the truncation given as the third argument of the setter, which should match the cutoff the model is solved at. A lattice above 2×10^5 points is refused by name rather than truncated, since a silently shortened table would make one language disagree with another on the same model.

Table 5.4 summarizes the queue-dependent service mechanisms by what the handle reads and returns, the field that stores them, and whether the resulting demands are product form.

5.2.5 Per-class dependence and tabulated scalings

The class-dependence function of §5.2.4 may also be made *per class*: instead of returning a single scaling shared by every class, $\beta_i(n_i)$ may return the vector $[\beta_{i,1}(n_i), \ldots, \beta_{i,R}(n_i)]$, so that each class r receives its own scaling at the same joint population $n_i = [n_{i,1}, \ldots, n_{i,R}]$. This is the chain-dependent service rate $\mu_{r,i}(n_i)$ of [137], and it is what allows a station to serve different classes at genuinely different rates in the same state. It is useful for systems whose server efficiency depends on the mix of job classes present, such as heterogeneous workloads competing for a shared resource.

A dependence obtained from measurements, or defined only over a bounded population range, is expressed by having the handle index a table and clamp the population to the range it was tabulated on. Earlier releases

Argument read	Return	Setter	Field	Prod. form
total $n_i = \sum_r n_{i,r}$	scalar (shared)	`setLoadDependence`	`lldscaling`	yes
own marginal $n_{i,r}$	per-class vector	`setClassDependence`	`cdscaling`	yes
total n_i	per-class vector	`setClassDependence`	`cdscaling`	yes
joint $[n_{i,1}, \ldots, n_{i,R}]$	scalar (shared)	`setJointDependence`	`jdscaling`	if OI
joint $[n_{i,1}, \ldots, n_{i,R}]$	per-class vector	`setJointDependence`	`jdscaling`	if OI
full state $(n_{i,r})$	scalar, $M \times 1$ or $M \times R$	`setGlobalDependence`	`gdscaling`	if balanced

Table 5.4: Queue-dependent service mechanisms. `setLoadDependence` takes a numeric vector $\alpha(n_i)$; `setClassDependence` and `setJointDependence` take a function handle of the joint population and a required per-class peak rate. The product-form column indicates whether the BCMP recurrence of [32] is preserved; a joint dependence preserves it only when the station is order independent (`OI`). A scalar handle that reads only the total n_i is equivalent to `lldscaling` and should be declared through `setLoadDependence`. The last row is declared on the MODEL rather than on a station, since its argument is the whole network state; it preserves a product form exactly when it satisfies the Whittle balance property [163], and only `SolverCTMC` and `SolverSSA` support it.

exposed this through separate limited joint dependence (LJD) and limited joint class dependence (LJCD) lookup tables, which the handle forms now subsume. A per-class tabulated scaling whose entry r reads only its own marginal is product form and belongs in `cdscaling`; a table that reads foreign marginals is joint and belongs in `jdscaling`.

Per-class dependence is used by the `MVA` solver during the mean value analysis iteration, and by the `NC` solver, which solves such models exactly by the multichain convolution of §5.6.3. Utilization at such a station is normalized by the per-class peak rate declared as the second argument of `setClassDependence`, following the convention of §4.2.2.

5.2.6 Discrete distributions

In addition to continuous service- and inter-arrival-time distributions, LINE provides a family of discrete distributions through the `DiscreteDistribution` class hierarchy. These distributions take values on a (possibly finite) integer support and are primarily used as count or popularity models, e.g. to specify the number of items requested from a cache, the number of arrivals per slot, or batch sizes. Table 5.5 summarizes the discrete distributions currently available.

All discrete distributions inherit the same inspection interface as the continuous ones: `get_mean`, `get_scv` and `sample(n)` return the first two moments and draw n pseudo-random values. In addition, `eval_pmf(k)` returns the probability mass at the integer k, and `eval_cdf(k)` returns the cumulative probability up to k.

A typical use case for `DiscreteSampler` is to specify the popularity profile of items in a `CacheTask` or `ItemEntry`. The example below assigns a non-uniform popularity to four items so that item 1 is requested with probability 0.5 and items 2–4 share the remaining mass:

```
items   = [1, 2, 3, 4]
```

Table 5.5: Discrete distributions available in LINE

Distribution	Description
Bernoulli	`Bernoulli`(p), single trial with success probability $p \in [0, 1]$. Mean $= p$, support $\{0, 1\}$
Binomial	`Binomial`(n, p), number of successes in n independent Bernoulli trials. Mean $= np$, support $\{0, \ldots, n\}$
Poisson	`Poisson`(λ), count of events occurring in a fixed interval at average rate λ. Mean $=$ Var $= \lambda$, support $\{0, 1, 2, \ldots\}$
DiscreteUniform	`DiscreteUniform`(a, b), uniform probability $1/(b - a + 1)$ on the integers $\{a, a + 1, \ldots, b\}$
Geometric	`Geometric`(p), number of Bernoulli trials needed to obtain the first success. Mean $= 1/p$, support $\{1, 2, \ldots\}$
DiscreteSampler	`DiscreteSampler`(p, x), fully user-specified discrete distribution. Probability vector p assigns mass p_i to value x_i. If x is omitted it defaults to $1{:}n$ where n is the length of p

```
weights = [0.5, 0.25, 0.15, 0.10]
popularity = DiscreteSampler(weights, items)
ids = popularity.sample(1000)
```

The same approach is used by `Zipf`(s, N) to assign power-law popularity to a finite catalogue of N items with exponent s, while `Bernoulli` and `Geometric` are convenient when modelling cache hit/miss processes or first-success counts.

Discrete-time modelling with DMAP. LINE supports discrete-time modelling through the `DMAP` (Discrete-time Markovian Arrival Process) class. Unlike the continuous-time `MAP`, where $D_0 + D_1$ is an infinitesimal generator, in a `DMAP` the matrix $D_0 + D_1$ is a (row-)stochastic transition matrix. A `DMAP` is constructed as follows:

```
dmap = DMAP(D0, D1)
```

A random two-phase DMAP can be generated with `DMAP.rand()`. The `DMAP` class inherits statistical accessors from its continuous-time counterpart (`get_mean`, `get_scv`, `sample`).

5.2.7 Temporal dependent processes

It is sometimes useful to specify the statistical properties of a *time series* of service or inter-arrival times, as in the case of systems with short- and long-range dependent workloads. When the model is stochastic, we refer to these as situations where one specifies a *process*, as opposed to only specifying the *distribution* of the service or inter-arrival times. In LINE processes include the 2-state Markov-modulated Poisson process (`MMPP2`) and empirical traces read from files (`Replayer`).

For the latter, LINE assumes that empirical traces are supplied as text files (ASCII), formatted as a column of numbers. Once specified, the `Replayer` object can be used as any other distribution. This means that it is possible to run a simulation of the model with the specified trace. However, analytical solvers will require tractable distributions from the `Markovian` class.

Table 5.6: Drop strategies available in LINE

Strategy	Description
`DropStrategy.Queue`	Default policy. Jobs that find the station full wait in an external (logical) waiting queue until capacity becomes available. Selected with `DropStrategy.Queue` or by passing `false` to the boolean overload of `setDropRule` on `Region`.
`DropStrategy.Drop`	Jobs that find the station full are rejected and leave the system permanently. Selected with the MATLAB constant `DropStrategy.DROP` or by passing `true` to the boolean overload of `setDropRule` on `Region`.
`DropStrategy.BlockingAfterService` (BAS)	Blocking-after-service. A job completing service waits at the server until the downstream destination has free capacity, blocking that server in the meantime. Selected via `DropStrategy.BAS` (MATLAB constant).
`DropStrategy.BlockingBeforeService` (BBS)	Blocking-before-service. A job is held back from entering service while the downstream destination is full, so that no service capacity is consumed for jobs that cannot complete. Selected via `DropStrategy.BBS` (MATLAB constant).
`DropStrategy.ReServiceOnRejection`	Re-service on rejection. A job rejected by a full downstream station is returned to the originating server and served again. Selected via `DropStrategy.RSRD` (MATLAB constant).
`DropStrategy.Retrial`	Job moves to a virtual orbit and retries after a random delay; the number of retry attempts is unlimited. Set automatically by `setRetrial(jobclass, delay)`.
`DropStrategy.RetrialWithLimit`	As `Retrial`, but with a finite cap on the number of attempts; once the cap is exhausted the job is dropped. Set automatically by `setRetrial(jobclass, delay, maxAttempts)`.

5.3 Job behaviour

5.3.1 Class priorities

If a class has a priority, **with 0 representing the highest priority**, this can be specified as an additional argument to both `OpenClass` and `ClosedClass`, e.g.,

```
class2 = ClosedClass(model, 'Class2', 5, queue, 0)
```

Important: Class priorities are only effective when a station uses a priority scheduling policy (see Table 3.3). If no priority scheduling policy is explicitly assigned to a station, class priorities are ignored. In `Network` models, priorities are intended as hard priorities. Weight-based policies such as DPS and GPS may be used, as an alternative, to prevent starvation of jobs in low-priority classes.

5.3.2 Drop strategies

When a station has finite capacity, the `DropStrategy` associated with each class determines what happens to a job that arrives at, or finishes service in, a full station. The drop rule is configured per class via `set_drop_rule` on a station node. The complete enumeration is summarised in Table 5.6; only `Queue` and `Drop` apply to per-station capacity in all four codebases, while `BAS`/`BBS`/`ReServiceOnRejection` are blocking semantics typically used in conjunction with finite buffers and routing to downstream finite-capacity destinations. The `Retrial` and `RetrialWithLimit` values are normally not set explicitly: they are assigned automatically when `setRetrial` is called on a queue node (see 5.3.5).

The next listing assigns different drop rules to two classes sharing the same finite-capacity queue: jobs of the first class are rejected on overflow, while jobs of the second class are admitted to the system-level waiting queue.

```
queue.set_drop_rule(class1, DropStrategy.DROP)
```

The same `setDropRule` API is exposed by the `Region` class for finite capacity regions (see 5.5); on `Region` the boolean overload `setDropRule(jobclass, true)`/`setDropRule(jobclass, false)` is also accepted, with `true` mapping to `DropStrategy.Drop` and `false` to `DropStrategy.Queue`. Solver support for the more advanced strategies depends on the chosen backend: simulators (`JMT`, `LDES`, `SSA`) handle the full range of blocking semantics, while analytical solvers typically require `Drop` or `Queue`.

5.3.3 Impatience (customer abandonment)

A job may leave a queue before being served. This models a caller who hangs up, a user who navigates away from a slow page, or a request that exceeds its deadline. On arrival the job draws a *patience* time from a distribution; if it has not entered service by the time that patience expires, it reneges, leaving the queue and being routed onward to the next node or to a sink. The patience distribution may be any standard one, exponential for memoryless impatience or deterministic for a fixed timeout, but not a modulated process (`MAP`, `BMAP`, `MMPP2`).

Patience is declared either on a queue, for one job class, or on the class itself, in which case it holds at every queue the class visits. When both are given, the queue-level declaration prevails.

```
queue = Queue(model, 'ServiceQueue', SchedStrategy.FCFS)
queue.set_patience(jobclass, Exp(0.1))  # mean patience = 10 at this queue
jobclass.set_patience(Det(5.0))         # fixed timeout = 5 everywhere else
```

LINE honours the feature both by tracking individual patience times on the sample path and, in exact analyses, by encoding reneging as an additional state transition of the Markov chain. The resulting abandonment rate is what separates offered from carried throughput, and is therefore the quantity of interest whenever a service level agreement is at stake.

5.3.4 Balking

An arriving job that balks inspects the state of the queue and leaves without joining, unlike reneging (see 5.3.3), where it joins and abandons later. A balking policy is configured per class on a queue node with `set_balking`, passing a `BalkingStrategy` constant and a list of thresholds `(minJobs, maxJobs, probability)`: the first range that contains the queue length at the arrival instant fixes the balking probability, and an unbounded upper limit is written as ∞ (MATLAB), `Integer.MAX_VALUE` (Java) or `float('inf')` (Python).

Three strategies are supported, listed in Table 5.7.

Table 5.7: Balking strategies available in LINE

Strategy	Description
`BalkingStrategy.QUEUE_LENGTH`	Balking decision is taken from the current queue length. The threshold list assigns a balking probability to each range of queue lengths.
`BalkingStrategy.EXPECTED_WAIT`	Balking decision is based on the expected waiting time, estimated from the current queue length and the per-class service rate. The customer balks when the estimated wait exceeds a tolerance threshold.
`BalkingStrategy.COMBINED`	Both queue-length and expected-wait conditions are evaluated; the customer balks when either condition triggers (logical OR). Useful when modelling customers that are sensitive to both visible queue length and perceived waiting time.

The following example configures a single class to balk with probability 0.3 when the queue holds 5 to 10 jobs, and with probability 1 (certain refusal) when 11 or more jobs are present.

```
queue.set_balking(jobclass, BalkingStrategy.QUEUE_LENGTH,
                  [(5, 10, 0.3), (11, float('inf'), 1.0)])
```

A simplified single-threshold form, `setBalking(jobClass, strategy, minJobs, probability)`, is available in Java and configures a single open-ended threshold. Balking combines with the impatience and retrial mechanisms above, as illustrated by the example `json_balking_retrial`.

5.3.5 Retrial queues

In a retrial queue a customer that finds all servers busy joins an orbit and reattempts service after a random delay, a pattern common in call centres, telecommunication systems and cloud platforms. The station is declared with `set_retrial`, giving the retrial delay distribution and optionally a maximum number of attempts (-1, the default, allows unlimited retries), and `set_orbit_impatience` makes orbiting customers abandon after a random time. The retrial delay must not be a modulated process, and `MAM`, `CTMC`, `SSA` and `LDES` declare support for the feature.

```
queue.set_retrial(jobclass, Exp(0.5))           # Unlimited retries
queue.set_retrial(jobclass, Erlang(0.3, 2), 3)  # Max 3 attempts
queue.set_orbit_impatience(jobclass, Exp(0.008))  # Orbit abandonment
```

Figure 5.2 shows the resulting flow.

First-class orbit declaration. The method `set_orbit` declares a retrial station in a single call, installing the retrial delay and recording the retrial policy and the orbit capacity. Its signature is `setOrbit(class, retrialDistribution, policy, maxOrbit)`, where `policy` is `RetrialPolicy.LINEAR` (default), under which each orbiting job retries at its own rate ν so that an orbit of n jobs retries at rate $n\nu$ (the classical Falin-Templeton queue), or `RetrialPolicy.CONSTANT`, under which the orbit retries as a whole at rate ν while non-empty. The argument `maxOrbit` is the orbit capacity, -1 (default) leaving it unbounded and a non-negative value making it finite, so that a job finding it full is lost; the station capacity is set accordingly, to

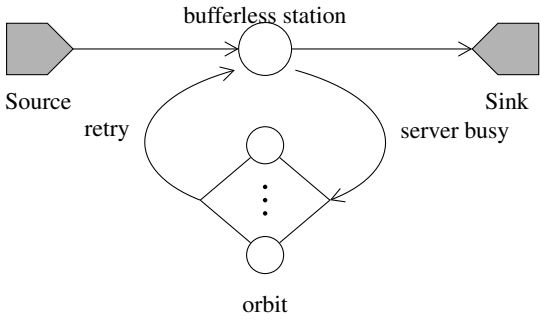


Figure 5.2: A retrial queue. The station has no waiting room, so a job that finds the server busy joins the orbit and retries after a random delay.

infinity or to the server count plus `maxOrbit`, since the state-space solvers hold the orbit in the station buffer and a capacity pinned to the server count would leave them nowhere to put an orbiting job. The configuration is read back with `get_orbit`.

```
queue.set_number_of_servers(1)
queue.set_orbit(jobclass, Exp(1.0))  # Unbounded orbit, linear policy
queue.set_orbit(jobclass, Exp(1.0), RetrialPolicy.CONSTANT, 20)
```

Note that the finite orbit declared by `maxOrbit` is a property of the model, not a truncation of the solution: it changes the loss probability seen by the arriving jobs, whereas the QBD truncation level discussed above is an accuracy control that leaves the model unchanged.

Orbit metric. At a retrial station `QLen` reports the whole station population, that is the jobs in service together with the jobs orbiting. The mean orbit length alone is returned by `get_avg_orbit`, an (*nstations* $\times$ *nclasses*) matrix that is zero at stations that are not retrial queues, and by its labelled variant `get_avg_orbit_table`.

```
solver = MAM(model)
ON = solver.get_avg_orbit()
print(solver.get_avg_orbit_table())
```

5.3.6 Batch arrivals

By default a `Source` releases one job per arrival epoch. A batch arrival stream is obtained by attaching a discrete batch-size distribution to a class with `set_arrival_batch`: at each epoch generated by the inter-arrival distribution, the source releases a number of jobs drawn from the batch distribution instead of a single job. The batch size must be at least one, so distributions that can return zero are rejected rather than clamped.

```
source.setArrival(oclass, Geometric(0.2))
source.setArrivalBatch(oclass, Geometric(0.5))
```

The batch distribution is stored in the `arrivalbatch` field of the network structure and is advertised through the `BatchArrival` language feature. Batch arrivals are simulated by the `LDES` solver. Combining a geometric inter-arrival distribution with a geometric batch size under slotted simulation (Section 8.2.5.1) yields the Geo^X arrival stream, whose analytical reference is `qsys_geoxgeo1`.

The dual construction applies at a station. When a `BMAP` is assigned as the *service* distribution, the `LDES` solver interprets it as a batch service process (BMSP): at a service epoch the server removes and completes a batch of jobs whose size is drawn from the BMAP batch-size distribution, truncated by the number of jobs present, so that the station behaves as an M/BMSP/1 queue. The same `BMAP` placed at a `Source` produces batch arrivals instead.

5.3.7 Server breakdowns

A station whose server is unreliable is declared with `set_breakdown` on a queue node: the server alternates between UP and DOWN, failing after the failure distribution and being restored after the repair distribution, with the solver expanding the joint (queue, server status) chain rather than the ensemble of Section 6.6. The failure clock runs whenever the server is up, so an idle station can fail; arrivals keep queueing while it is down, and a job in service is not lost but resumes on repair. An optional third argument gives the service distribution used while down, one distribution or a per-class list, and omitting it means no service at all while down. The configuration is read back with `get_breakdown`.

```
queue.set_breakdown(Exp(0.0001), Exp(0.1))
queue.set_breakdown(Exp(0.0001), Exp(0.1), Exp(0.5))
```

Only exponential failure and repair distributions are expanded into the joint chain; any other process is rejected with an explicit error, and the random-environment ensemble remains available for it. Breakdowns are registered as the `Breakdown` language feature, so a solver that does not build the joint chain rejects the model instead of silently solving it breakdown-free. The `CTMC` solver supports both its steady-state and its transient solution.

5.3.8 Slotted models and the discrete-time solver

A model whose interarrival and service laws all live on a slot lattice is a discrete-time model, and `SolverMAM` recognizes it from the distributions alone before any phase-type conversion takes place. The lattice families are `Geometric` (support $\{1, 2, \dots\}$ slots), `DMAP`, `DiscreteUniform` with integral bounds, and `Det` on an integral number of slots; a model that mixes these with continuous laws is solved on the continuous time scale as before. The decision can be forced with `options.config.timescale`, which takes `auto` (the default), `discrete` or `continuous`, and the slot is set with `options.config.slotlength`. Requesting `discrete` on a model that mixes lattice and non-lattice laws raises an error rather than answering on the

wrong time scale. A `DMAP` combined with continuous laws is always an error: its (D_0, D_1) are probability matrices, so the continuous machinery would form $(-D_0)^{-1}$ where the law needs $(I - D_0)^{-1}$.

The convention is the late arrival system with delayed access. Within a slot the service completion resolves first, the arrival is appended at the end of the slot and cannot enter service before the next one, and every metric is read after both events. This is the convention of the `Q_DT_*` queues, of the analytical references `dqsys_geogeo1` and `dqsys_geoxgeo1`, and of the slotted mode of the `LDES` simulator, so the four are directly comparable.

A single queueing station is solved exactly, under the method name `dt.qmam`: `Q_DT_PH_PH_1` when both the arrival and the service process are renewal discrete phase-type, `Q_DT_MAP_MAP_1` otherwise. Several stations are solved by a discrete-time parametric decomposition under the name `dt.dec`, which is an approximation: superposing slotted streams produces batches, since two streams fire in the same slot with positive probability, so each station is an M/G/1-type chain rather than a quasi-birth-death process, and the departure process handed downstream is truncated at a finite level and compressed back to the phase budget of `options.config.space_max`. A tandem of geometric-server queues is nonetheless exact, because the stationary departure stream of a $\mathrm{Geo}/\mathrm{Geo}/1$ queue is Bernoulli.

The discrete-time path covers open, single-class models whose queueing stations are single-server FCFS. Multiserver stations, infinite servers, closed populations and several classes are refused by name rather than silently solved on the continuous scale.

```
source.set_arrival(jobclass, Geometric(0.2))
queue.set_service(jobclass, Geometric(0.5))
solver = MAM(model)
print(solver.get_avg_table())
```

5.4 Routing and synchronization

5.4.1 Other routing strategies

The above routing specification style is only for models with probabilistic routing strategies between every pair of nodes. A different style should be used for scheduling policies that do not require to explicit routing probabilities, as in the case of state-dependent routing. Currently supported strategies include:

- Round robin (`RoutingStrategy.RROBIN`). This is a deterministic strategy that sends jobs to outgoing links in a cyclic order.
- Random routing (`RoutingStrategy.RAND`). This is equivalent to a standard probabilistic strategy that for each class assigns identical values to the routing probabilities of all outgoing links. When a target is invalid its probability is kept to zero, e.g., random routing will not send a job in a closed class to a sink.
- Join-the-Shortest-Queue (`RoutingStrategy.JSQ`). This is a non-probabilistic strategy that sends jobs to the destination with the smallest total number of jobs in it (either queueing or receiving service). If multiple stations have the same total number of jobs, then the destination is chosen at random with equal

probability.

- Weighted round robin (`RoutingStrategy.WRROBIN`). Routes jobs cyclically through a list in which each destination appears as many times as its integer weight, so destinations with higher weights are visited proportionally more often. Configure each weight with a per-destination call `router.set_routing(cls, RoutingStrategy.WRROBIN, dest, weight)`; with all weights equal to one this collapses to ordinary round-robin.
- Shortest queue of d, SQ(d) (`RoutingStrategy.SQ`, formerly `KCHOICES`). Samples d candidate destinations uniformly at random *with replacement* from the set of available outgoing links and dispatches the job to the one with the shortest queue, breaking ties by first occurrence in the candidate list [112, 156]. The parameter d defaults to 2 if not specified. Dispatcher memory is not supported. To set a specific value of d use `set_routing(cls, RoutingStrategy.SQ, d)`.
- Product-form state-dependent routing (`RoutingStrategy.SDR`), see §5.4.2.
- Event-based routing (`RoutingStrategy.FIRING`). This strategy is used internally by `Transition` nodes in stochastic Petri net models and routes tokens to output places according to the firing outcomes configured for each mode of the transition. It is set automatically when specifying Petri net firing outcomes and is not intended for direct assignment by the user.
- Disabled routing (`RoutingStrategy.DISABLED`). This pseudo-strategy explicitly marks a (node, class) pair as inactive and prevents jobs of that class from being dispatched out of the node. It is used internally by the model linker to suppress routing for classes that should not visit a given node, and is not normally set by user code.

For the above policies, the function `add_link` should be first used to specify pairs of connected nodes

```
model.add_link(queue, queue) #self-loop
model.add_link(queue, delay)
```

Then an appropriate routing strategy should be selected at every node, e.g.,

```
queue.set_routing(class1,RoutingStrategy.RROBIN)
```

assigns round robin among all outgoing links from the `queue` node.

A model could also include both classes with probabilistic routing strategies and classes that use round-robin or other non-probabilistic strategies. To instantiate routing probabilities in such situations one should then use, e.g.,

```
queue.set_routing(class1,RoutingStrategy.PROB)
queue.set_prob_routing(class1, queue, 0.7)
queue.set_prob_routing(class1, delay, 0.3)
```

where `set_prob_routing` assigns the routing probabilities to the two links.

5.4.2 Product-form state-dependent routing

Adaptive strategies such as `JSQ` and `SQ` take a model out of product form, so a network using them must be simulated or solved on its state space. The state-dependent routing of Krzesinski [90], the multiclass generalization of the rule of [151], is the exception: its routing probabilities depend on the branch populations and yet the stationary distribution keeps a product form, so `SolverNC` solves such a model exactly and `SolverCTMC` and `SolverSSA` accept it as well.

A node is declared the *entry center* e of a subnetwork $Q(V, V)$ whose branches it feeds:

```
node1.set_state_dep_routing(jobclass, node1, [[], [node2], [node3]], [0, 1, 2], ...
[-1, -1], d)
```

The arguments follow the paper's own indexing. The second names the departure center of $Q(V, V)$, which is the entry center itself in a central-server model. The branch list has an empty first entry, because branch index 1 denotes the complement $M - V$, and lists from index 2 onwards the nodes of each branch, entry center first and departure center last. The level vector gives, for each branch, the index t of the subnetwork holding it, its first entry being unused; C is the vector of coefficients C_t and d the matrix of coefficients d_{tb}. Negative C_t and positive d_{tb} make the routing prefer the least congested branches and impose the population bounds $m_b \leq d_{tb}/(-C_t)$ and $v_t \leq D_{tt}/(-C_t)$. With one level, four single-center branches, $C_1 = -1$ and $d_{1i} = d$, the entry center routes as $P_{1,i}(N) = (d - n_i)/(4d - V)$ with $V = n_2 + n_3 + n_4 + n_5$, which is adaptive routing with balanced loading; the state-independent counterpart is $P_{1,i} = 1/4$.

The default `NC` dispatch recognises the declaration and solves the model with it. The method can also be named explicitly: `sdr` evaluates the product form of the paper's eq. (16) exactly by state enumeration and accepts any branch topology, while `sdr.mva` runs instead the Section 4 mean value analysis and convolution, which costs polynomially in the populations rather than the size of the state space but is stated for single-center branches only and requires every C_t to be negative. The examples `sdroute_krzesinski`, which reproduces the paper's Section 5.1.1, and `sdroute_multibranch` illustrate both.

5.4.3 Fork and Join nodes

The fork and join nodes are supported by the `JMT` and `LDES` simulators, by the `CTMC` and `SSA` solvers through a native fork-join state representation (closed and multiclass models, including `set_tasks_per_link`), and, for mean performance metrics, by the `MVA` and `MAM` analytical solvers (and by `LN` for layered models). The `Fork` splits an incoming job into a set of sibling tasks, sending out one task for each outgoing link. These tasks inherit the class of the original job and are served as normal jobs until they are reassembled at a `Join` station.

Their specification of Fork and Join nodes only requires the name of the node

```
fork = Fork(model, 'Fork')
join = Join(model, 'Join', fork)
```

The number of tasks sent by a `Fork` on each output link can be set using the `set_tasks_per_link` method of the `fork` object. To enable effective analytical approximations, presently LINE requires that every join node is bound to a specific fork node, although specific solvers will ignore this information (e.g., JMT).

Quorum (partial) joins. A `Join` waits by default for every sibling of the forked job. It can instead be told to fire on the first k of n siblings, discarding the ones that arrive later, which is the early-reply pattern of replicated storage and quorum protocols:

```
join.set_strategy(jobclass, JoinStrategy.PARTIAL)
join.set_required(jobclass, 2)    # fire on the second of three siblings
```

The simulators `JMT` and `LDES` reproduce the discipline on the sample path. The analytical solvers `MVA` and `NC` charge the k-th order statistic $E[X_{(k)}]$ of the branch completion times at the join instead of $E[\max]$ (`Fj_ordstat_exp`), and the request tail is read from the k-of-n approximation `Fj_tail_ordstat` behind `get_perct_resp_t(p, None, 'forktail')`. The feature is declared as `JoinPartial` in the feature sets of `MVA`, `NC`, `JMT` and `LDES`; `CTMC`, `SSA` and `MAM` refuse it by name rather than answering the question of a standard join. A quorum join is also the one exception to the $\texttt{ArvR} - \texttt{Tput}$ identity of §4.2.5: the discarded siblings make the join loss rate $\max(0, \texttt{ArvR} - k\,\texttt{Tput})$. The example `gallery_fj_quorum` is a closed network with a 2-of-3 quorum.

Variable forking levels. The forking level need not be one number shared by every link and every job. Three overrides sit beside the scalar and may be given in any order with respect to the links themselves:

- `set_tasks_per_link(n, jobclass, dest_node)` gives one class, and optionally one destination, its own number of tasks, leaving the other classes and links on the node-wide value.
- `set_tasks_per_link_distribution(jobclass, dist, dest_node)` makes the number of tasks a draw from `DiscreteSampler`, redrawn independently for every link and every forked job. This is the variable forking level of JMT's `JobsPerLinkDis`.
- `set_branch_probability(jobclass, dest_node, p)` fires the branch towards `dest` only with probability p, and emits nothing otherwise. The branches are activated independently, so the number of siblings is random even when the tasks per link are deterministic. The matched `Join` must be told what to wait for, since a job that skipped a branch would block a standard join forever: a branch probability below one requires `JoinStrategy.PARTIAL` or a quorum (§5.4.3).

The scalar remains what every existing consumer reads and is now the *mean*, with the branch probability folded into it. The overrides are exact under `JMT` and `LDES`, which draw the degree at the fork epoch, and are seen as the expected degree by the analytical solvers, whose MMT correction is linear in it. The exact `CTMC` and `SSA` path serves an integer per-link count together with `JoinStrategy.PARTIAL`, but refuses a random degree and a branch probability below one by name, rather than answering with the certain-fork numbers, because its tag construction fixes the sibling count and the sibling set when the state space is built. The example `fj_variable_fanout` builds the same network in each of the four modes.

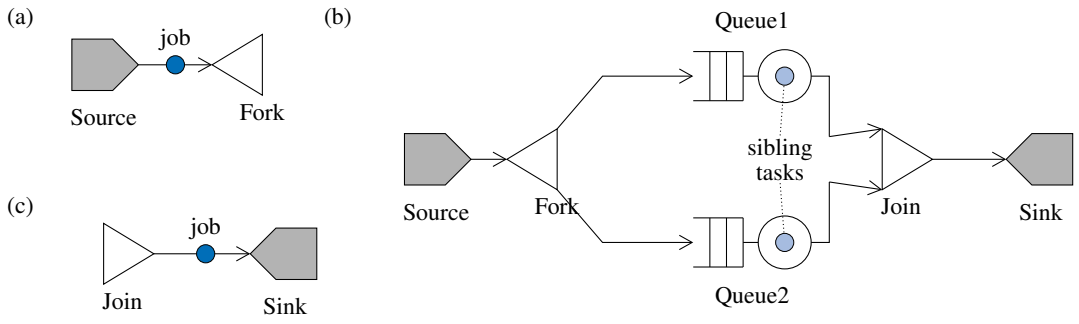


Figure 5.3: Fork and join. (a) A job travelling from the source to the fork. (b) The fork replaces that job with one sibling task per outgoing link, drawn in the lighter shade; the tasks are served as ordinary jobs and are reassembled at the join. (c) The job restored by the join, travelling on to the sink.

Also note that the routing probabilities out of the `Fork` node need to be set to 1.0 towards every other node connected to the `Fork`. For example, a `Fork` sending jobs in class 1 to nodes A, B and C, cannot send jobs in class 2 only to A and B: it must send them to all three connected nodes A, B and C. A new fork node visited only by class-2 jobs needs to be created in order to send that class of jobs only to A and B.

Fork nodes also support probabilistic (state-dependent) branching via `RoutingStrategy.PROB`, where different output links carry non-uniform routing probabilities. This is set using `set_prob_routing` on the Fork node and is available when using `JMT`. This allows a Fork node to direct tasks to different parallel branches with configurable proportions, enabling asymmetric fork-join models where not all branches are equally loaded.

After splitting a job into tasks, LINE takes the convention that visit counts refer to the average number of passages at the target resources for the original job, scaled by the number of tasks. For example, if a job is split into two tasks at a fork node, each visiting respectively nodes A and B, the average visit count at A and B will be 0.5.

For Fork-Join response time percentiles, `MAM` automatically detects valid FJ topologies (Source → Fork → K Queues → Join → Sink) and applies percentile computation as follows

```
solver = MAM(model)
perc_rt = solver.get_perct_resp_t([90, 95, 99])
```

Figure 5.3 shows the resulting topology.

5.4.4 Class switching with non-probabilistic routing strategies

In the presence of non-probabilistic routing strategies, such as round-robin or join-the-shortest-queue, one may need to manually specify the details of the class switching mechanism. This can be done through addition to the network topology of `ClassSwitch` nodes.

The constructor of the `ClassSwitch` node requires a probability matrix C such that the lement in row r and column s is the probability that a job of class r arriving into the node switches to class s during the visit. For example, in a 2-class model, the following node will switch all visiting jobs into class 2

```
# Block 1: nodes
...
csnode = ClassSwitch(model, 'ClassSwitch 1')
# Block 2: classes
jobclass = np.empty(2, dtype=object)
jobclass[0] = OpenClass(model, 'Class1', 0)
jobclass[1] = OpenClass(model, 'Class2', 0)
...
# Block 3: topology
C = csnode.init_class_switch_matrix()
C[0][1] = 1.0
C[1][1] = 1.0
csnode.set_class_switching_matrix(C)
```

Note that for a network with M stations, up to M^2 `ClassSwitch` elements may be required to implement class-switching across all possible links, including self-loops.

5.4.5 Logger node

A logger node is a node that closely resembles the logger node available in the JSIMgraph simulator within JMT. Models that include this element can be solved using the `JMT` and `LDES` simulators.

A `Logger` node records information about passing jobs in a `csv` file, such as the timestamp of passage and general information about the jobs. The node can be instantiated as follows

```
logger = Logger(model, 'LoggerNode', 'logfile.csv')
```

The file is written in CSV format, one row per job passage, with a column for each enabled field. The `LogTunnel` section that implements the logging is transparent: a job crosses the logger instantaneously and no queueing delay is introduced, so a logger may be placed anywhere in the topology without perturbing the metrics being measured. Placing several loggers yields a trace of job trajectories from which empirical inter-passage distributions can be estimated offline.

The following methods can be used to specify the information that needs to be stored in the `csv` file

- `set_start_time`: record a timestamp for the wallclock time when the simulation started.
- `set_job_id`: record a unique id for the passing job.
- `set_job_class`: record the class of the passing job.
- `set_timestamp`: record a timestamp for the simulated time when the job passed in the logger.
- `set_time_same_class`: record the time elapsed since the last passage of a job of the same class.
- `set_time_any_class`: record the time elapsed since the last passage of a job of any class.

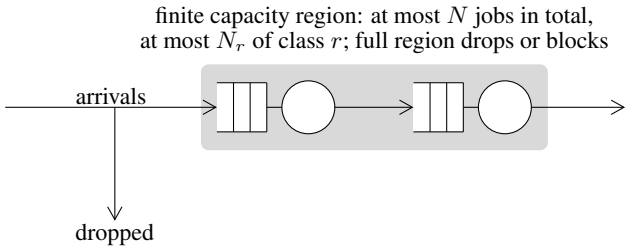


Figure 5.4: A finite capacity region. The constraint applies to the total occupancy of the member stations rather than to any single buffer, and an arrival that would violate it is dropped or blocked according to the drop rule.

Each method can be called either with a single `True` or `False` argument, to enable or disable the recording of the corresponding information, e.g.

```
logger.set_job_class(True)
```

The routing behavior of jobs can be set up as explained for regular nodes such as queues or delay stations.

5.5 Finite capacity regions

A Finite Capacity Region (FCR) provides a mechanism to constrain the total number of jobs that can simultaneously reside within a designated group of stations in a queueing network. Unlike per-station buffer limits, which restrict occupancy at individual queues, an FCR imposes a collective constraint across multiple stations, limiting the aggregate number of jobs across all stations in the region. When the region reaches its capacity, incoming jobs are either dropped or blocked depending on the configured drop strategy. This modeling capability is particularly useful for representing systems with admission control policies, finite buffers spanning multiple processing stages, resource constraints that affect groups of servers, or loss networks where jobs are rejected when the system lacks sufficient capacity.

An FCR is created using the `add_region` method of the `Network` class, which takes as input a cell array or list of stations belonging to the region. The maximum occupancy across all stations in the region is set via `set_global_max_jobs`, which specifies the total number of jobs allowed in the region. The drop behavior for each class is configured using `set_drop_rule`, which determines whether arriving jobs are dropped when the region is at capacity.

Figure 5.4 illustrates the mechanism.

```
fcr = model.add_region(queue1, queue2)
fcr.set_global_max_jobs(K)
fcr.set_drop_rule(jobclass1, True)  # True = drop jobs when region full
```

Two primary drop strategies are available. The `DropStrategy.Drop` strategy causes jobs to be permanently lost when they arrive at a full region, modeling systems where arrivals are rejected or abandoned. The `DropStrategy.Queue` strategy causes jobs to wait in a virtual queue until capacity becomes available, modeling systems with admission control or backpressure mechanisms. In addition to global capacity constraints, per-class capacity limits can be specified within a region to enforce class-specific occupancy restrictions.

Beyond the global job cap, a region may declare several further admission constraints, and they compose. Per-class occupancy limits are set with `set_class_max_jobs`. A pooled resource whose classes consume unequal shares is expressed by giving each class a footprint with `set_class_size` and a budget with `set_global_max_memory`, which admits a job only while the weighted occupancy stays within the budget. Arbitrary linear admission rules are supplied directly with `set_constraint(A,b)`, where $\mathbf{A}$ has one column per class and one row per constraint.

Taken together these declarations define the admission rule $\mathbf{An} \leq \mathbf{C}$ on the vector $\mathbf{n}$ of per-class occupancies of the region, an arriving job of class r being admitted only when $\mathbf{A}(\mathbf{n} + \mathbf{e}_r) \leq \mathbf{C}$. Every row is a function of the per-class occupancy of the region alone, so no constraint can distinguish the individual stations inside it: a job moving from one station of the region to another leaves every left-hand side unchanged. The admissible set is coordinate convex, which is what allows the truncated model to retain a product form under the `Drop` policy.

The simulators (`JMT`, `LDES`, `SSA`) and `SolverCTMC` honour the full rule for any region contents. Among the analytical solvers, `SolverNC` recognizes the open single-`Delay` region under `Drop` as a loss network and solves it from the exact normalizing constant of the product form, assembling $\mathbf{A}$ and $\mathbf{C}$ from all four declarations above. Four methods are offered there through `options.method`: `exact`, the default on an integral admission rule, which evaluates the normalizing constant exactly by contour integration and residues [104]; `rec`, which evaluates the same constant exactly as the sum of the product form over the admissible set $\{n \geq 0 : \mathbf{A}n \leq \mathbf{C}\}$, by one memoised walk of a multi-valued decision diagram holding that set [7], and which is therefore the default when the region declares fractional class sizes or capacities, where the residue argument cannot count whole units; `erlangfp`, the Erlang fixed-point (reduced-load) approximation [84]; and `mci`, importance-sampling Monte Carlo summation with confidence intervals [133], for regions whose capacities make the exact evaluation too large. The loss-network tutorial in the LINE primer works through a complete example. `SolverFLD` carries a region under its `dae` method (§8.2.4.2), where the cap becomes a linear algebraic constraint solved beside the fluid drift and the blocked jobs are held in an explicit waiting room; that route accepts the waiting-queue rule only, and refuses per-class admission weights and the blocking and retrial rules, which change the event set rather than constrain the drift. The remaining analytical solvers, and every other fluid method, reject models with a finite capacity region rather than returning the unconstrained answer.

A special case of interest is an FCR containing a single queue with dropping enabled. This configuration behaves identically to an M/M/1/K queue, where K is the maximum capacity including both jobs in service and jobs waiting in the buffer. This equivalence provides a convenient way to model loss systems and finite-buffer queues. The examples `fcr_mm1kdrop`, `fcr_mm1waitq`, `fcr_constraints`, and `fcr_oqndrop` demonstrate various applications of finite capacity regions, including single-queue finite buffers, multi-station

regions with different drop policies, and open queueing networks with job loss.

Linear admission constraints. Besides the global and per-class job caps, a region admits an arbitrary set of linear inequalities $A\mathbf{n} \leq \mathbf{b}$ on the vector $\mathbf{n} = (n_1, \ldots, n_R)$ of per-class job counts inside it. A job is admitted only while every inequality stays satisfied. The pair is set with `set_linear_constraints(A, b)` and read back with `get_linear_constraints()`; the example `fcr_lincon` illustrates it.

```
fcr = model.add_region(queue1, queue2)
fcr.set_linear_constraints(A, b)
```

Two further per-region metrics are available with the `JMT` and `LDES` solvers, and are useful when classes consume unequal shares of a pooled resource, as in cache or storage models. `FCRWeight` (`MetricType.FCRWeight`) is the time-average total weight of the in-region jobs, summing the per-class weight set with `set_class_weight`; with the default weight 1 it coincides with the number of customers. `FCRMemOcc` (`MetricType.FCRMemOcc`) is the time-average memory occupation, summing the per-class footprint set with `set_class_size`.

In `JMT` the two map to the native region measures `FCR Capacity` and `FCR Memory`; `LDES` accumulates the same time-weighted quantities directly. They are returned per region and per class as `WeightNfcr` and `MemOccNfcr`, whose row sums are the region aggregates. They also appear in the table returned by `get_avg_region_table()`, next to the region queue length, response time and throughput.

5.6 Model adaptation

The `ModelAdapter` class provides static methods for transforming and adapting `Network` models. These methods are useful for model reduction, response time analysis, and preprocessing operations.

5.6.1 Chain aggregation

The `ModelAdapter.aggregate_chains` function transforms a multi-class model with K classes organized into C chains into a model with C classes, preserving chain population, arrival rates, service demands, and routing while reducing state space when $C \ll K$. The function returns the aggregated model, an aggregation factors matrix α satisfying $\sum_{r \in c} \alpha(i, r) = 1$, and deaggregation information for transforming chain-level results back to class-level metrics.

```
chain_model, alpha, deagg_info = ModelAdapter.aggregate_chains(model)
```

The same transformation is reachable from the model itself, which is the recommended form:

```
chain_model, alpha, deagg_info = model.aggregate_chains()
```

Class switching is eliminated by the transformation, since it is internal to a chain. The aggregation is exact for product-form models and approximate otherwise, because a single service process fitted on the chain

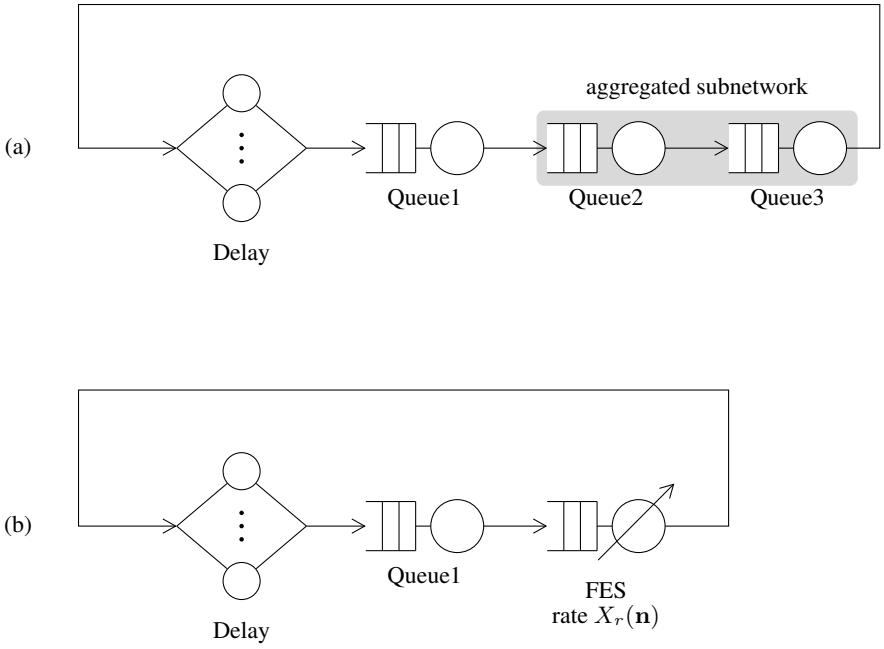


Figure 5.5: Flow-equivalent server aggregation. (a) The closed network before the transformation, with the aggregated subnetwork shaded. (b) The reduced network, in which the two aggregated stations are replaced by a single load-dependent station whose class-r rate is the throughput $X_r(\mathbf{n})$ of the isolated subnetwork.

mean and squared coefficient of variation replaces the per-class ones weighted by α. Chain-level metrics are mapped back to class level with `sn_deaggregate_chain_results`, passing the chain queue lengths and utilizations as empty: splitting the chain queue length by α alone ignores the per-class service times, whereas the response-time branch rescales them by S_{ir}/S_{ic}. The `CTMC` solver can drive the whole round trip itself, through `options.config.chain_aggregation` (§8.2.3.1).

5.6.2 Flow-equivalent server aggregation

The `ModelAdapter.aggregate_fes` function replaces a chosen subset of the stations of a closed product-form network with a single flow-equivalent server, returning the reduced model, the aggregated station and the information needed to recover the metrics of the original stations. The reduced model is smaller and cheaper to solve, and its throughput and utilization metrics are exact. The `CTMC` solver applies the transformation and recovers the collapsed stations' own metrics in a single solve when `options.config.fes_stations` names them (§8.2.3.1). The transformation is shown in Figure 5.5.

5.6.3 Convolution-based FES analysis

The FES station produced by `aggregateFES` stores its throughput lookup table as a *per-class dependence* function (§5.2.5). For each aggregated station the per-class throughput $X_r(n)$ is tabulated over all population vectors up to a per-class cutoff, so that the exact dependence of throughput on the joint class population is preserved; the table is then wrapped as the handle $\beta_{i,r}(n) = X_r(n)$, i.e. Sauer's chain-dependent service rate $\mu_{r,i}(n)$ [137], and stored in the `NetworkStruct` field `cdscaling{ist}`. The population is clamped to the cutoffs the table was built on, so the rate saturates beyond the tabulated range.

When a model contains one or more class-dependent stations, the `NC` solver uses the multichain convolution algorithm (`pfqn_conv`) to compute exact normalizing constants and performance metrics, building each station factor $X_m(n)$ by the recurrence $X_m(n) = (u_{km}/\mu_{km}(n))\, X_m(n - e_k)$ [137]. The convolution is invoked automatically by the `NC` solver when a class-dependence function is detected and `method='exact'` is requested; no explicit user action is required beyond creating the FES model via `aggregateFES` with appropriate cutoffs.

Burstiness-preserving FES analysis. The flow-equivalent server of §5.6.2 keeps the mean throughput of the aggregated subnetwork and discards every other property of its output. That is exactly right for a product-form subnetwork, by Norton's theorem [37], and it is the reason the reduced model of Figure 5.5 is exact. It is also why the transformation cannot represent a *bottleneck switch*, in which the congested resource alternates over time among the aggregated stations: the alternation is a property of the departure stream that a single mean rate cannot carry. LINE therefore also provides the load-dependent MAP flow-equivalent server of [30], in which the composite station serves according to a Markovian arrival process that changes with the population it holds.

The burstiness of a stream is measured by its asymptotic index of dispersion, which combines the squared coefficient of variation of the inter-departure times with their autocorrelation at every lag. It equals one for a Poisson process and grows as samples of similar size aggregate into bursts. A subnetwork made of one station with MAP service and a flow-equivalent server emits inter-departure times that are themselves a MAP, block bidiagonal over the population held by the flow-equivalent server. Its first three moments and its index of dispersion are obtained by a vector recursion that never forms a dense inverse, and a second-order MAP is then fitted to them: a MAP(2) has a geometrically decaying autocorrelation, so the index of dispersion inverts in closed form into a decay rate, which is handed to the explicit characterization in [74]. Stations are folded one at a time, so a subnetwork of any size reduces to a single load-dependent MAP at a cost independent of the number of stations.

When the aggregate is not overdispersed, that is when its inter-departure times are no more variable than an exponential or their index of dispersion falls below their squared coefficient of variation, the fit returns an exponential. The rule is not merely a feasibility guard and should not be relaxed: an exponential load-dependent server is exact for a product-form subnetwork, whereas a MAP(2) fitted to the marginal inter-departure statistics is not, because the departure stream of a subnetwork is not independent of the rest of the model. With the rule in place the construction degrades gracefully to the classic flow-equivalent server, and reproduces exact mean value analysis to machine precision on an exponential subnetwork.

The API is available in all four languages, as `fes_map_interdeparture`, `fes_map_moments`, `fes_map_aggregate` for the aggregation, and `fes_map_solve`, `fes_map_deaggregate` for the reduced model. The latter reads the block-bidiagonal inter-departure MAP as the generator of the closed model made of the think times and the aggregate, its level being the number of jobs held by the flow-equivalent server; the think time may itself be a MAP, which is how a bounded flash crowd is modelled. Per-station metrics follow by conditioning on the population of the aggregate, which is the decomposition step of the hierarchical analysis [37] and is exact for a product-form subnetwork.

```
fes, info = fes_map_aggregate(maps, [1, 1], N)
X, R, Q, pk = fes_map_solve(fes, map_exponential(Z), N)
```

On a closed model with two bursty MAP stations and a delay, the reduced model built this way predicts the throughput within 2.8–5.0% of the exact Markov chain, where the product-form solution of the same model is in error by 21.7–47.0%; when the subnetwork holds a single station, so that no aggregation approximation is involved, the reduced model reproduces the exact chain to machine precision.

5.6.4 Tagged job models

The `ModelAdapter.tag_chain` function creates a tagged job model for response time distribution analysis. It isolates a single job from a specified chain by creating duplicate tagged classes, enabling the computation of per-job response time distributions using methods such as passage time analysis.

5.6.5 Class removal

The `ModelAdapter.remove_class` function removes a specified job class from a model, updating all station configurations, routing matrices, and class-dependent parameters accordingly. This is useful for analyzing submodels or simplifying complex multi-class networks. The removal is delegated to the nodes, each of which drops its own per-class configuration (service and arrival processes, class capacities, scheduling parameters, routing strategies, and the class-switching matrix of a `ClassSwitch`). Removing the last remaining class, or any class of a model containing a `Cache`, is rejected: the cache item state is indexed by class and cannot be re-indexed on the node alone.

Two variants are available. The model method removes the class in place, whereas `ModelAdapter` returns a copy and leaves the original model solvable, which is the form to use for ablation studies or a per-class decomposition:

```
model.remove_class(jobclass)              # in place
newmodel = model.without_class(jobclass)  # on a copy
```

Chapter 6

Other stochastic models

Most formalisms in this chapter are described with the same `Network` class as a queueing network, and are solved through the same solver, but they model something else: an item cache, a stochastic Petri net, and a network in which jobs may be created or destroyed by signals. The chapter opens instead with the two classes that carry a Markov chain written down directly, without a network around it. They are collected here so that the queueing-network chapters stay about queues.

6.1 Markov chains and processes

A model need not be a network. LINE accepts a chain written down by the user, namely a `MarkovProcess`, holding an infinitesimal generator Q, or a `MarkovChain`, holding a transition matrix P. No state space is generated in this mode, the given matrix being solved as it stands; a discrete-time chain is carried as its image $P - I$, which has the same stationary distribution. The methods that describe the chain remain available – `getProbSys` for the stationary distribution, `getGenerator`, `getStateSpace`, `getProb` for a single state, `getTranProbSys` for the transient distribution and `sampleSys` for a sample path – whereas the average performance metrics are refused, a chain having neither stations nor classes. A state is identified by its row in the state space attached to the chain with `setStateSpace`, or by its index when the chain carries none. In transient analysis the initial distribution is `options.init_sol`, or the uniform distribution when none is given.

```
pi = SolverCTMC(MarkovProcess(Q)).getProbSys()
pi_dtmc = SolverCTMC(MarkovChain(P)).getProbSys()
pi_t = SolverCTMC(MarkovProcess(Q)).getTranProbSys(10.0)
```

The two classes also expose the Markov-chain algorithms of the API directly, so that a chain can be analysed without going through a solver at all. On a `MarkovProcess`:

- `solve` and `solveRelative`: the stationary distribution, normalized or relative to a reference state.

- `transient` (`transientProb` in Java, `transient` being a keyword there): the distribution at a finite horizon, its method argument selecting Jensen uniformization [80] or the Fox-Glynn weights [58].
- `timeAverage`: the time-averaged distribution over a horizon, read off the same uniformization series [80] that gives the endpoint distribution.
- `sens`: the sensitivity of the stationary distribution to the generator entries, from the differentiated balance equations [153].
- `aggregate`: an approximate distribution obtained by aggregation-disaggregation over a macrostate partition, by the Courtois [47], Koury-McAllister-Stewart [88], Takahashi [149] or multi-level [146] method, returned together with the nearly-complete-decomposability index of the partition.

On a `MarkovChain`:

- `solve`: the stationary distribution.
- `transient`: the distribution after a number of steps, and `transientUnif` over a continuous horizon.
- `hittingTime`: the expected time to reach a set of states.

On both classes: `stochComp` and `stochCompFull` (the stochastic complement of a subset of states, the latter returning also the blocks it was built from), `isFeasible`, `toTimeReversed` and `sample`. A continuous-time chain converts to discrete time in two distinct ways: `toDTMC` uniformizes it, preserving the stationary distribution, whereas `toEmbedded` returns the embedded jump chain, which records the sequence of states visited and drops the holding times.

6.2 Caches

6.2.1 Cache node

This is a stateful node to store one or more items in a cache of finite size, for which it is possible to specify a replacement policy. The *cache* constructor requires the total cache capacity and the number of items that can be referenced by the jobs in transit, e.g.,

```
cacheNode = Cache(model, 'Cache1', nitems, capacity, ReplacementStrategy.LRU)
```

If the capacity is a scalar integer (e.g., `15`), then it represents the total number of items that can be cached and the value cannot be greater than the number of items. Conversely, if it is a vector of integers (e.g., [10,5]) then the node is a list-based cache, where the vector entries specify the capacity of each list. We point to [65] for more details on list-based caches and their replacement policies.

Available replacement policies are specified within the `ReplacementStrategy` static class and include:

- First-in first-out (`ReplacementStrategy.FIFO`)
- Random replacement (`ReplacementStrategy.RR`)
- Least-recently used (`ReplacementStrategy.LRU`)

- Strict first-in first-out (`ReplacementStrategy.SFIFO`)

Upon cache hit or cache miss, a job in transit is switched to a user-specified class. More details are given later in Section 3.2.5.

Beyond basic capacity constraints, cache nodes in LINE support item sizes, where each cached item consumes a configurable amount of storage rather than occupying a single slot; this is described in Section 6.2.2. State-dependent routing exploits detailed cache state information to direct jobs along different paths depending on whether their requested item is present in the cache. When a job requests a cached item (a hit), it is classified into a hit class and routed to a delay node representing fast cache access, while jobs experiencing misses are switched to a miss class and directed to a queue representing slower backend retrieval. Solvers such as CTMC and NC employ stochastic complementation techniques to efficiently analyze cache models by eliminating intermediate Router states and computing effective transition rates between observable cache states.

6.2.2 Item sizes and storage cost caps

By default every item occupies one slot of a list, so a list holds as many items as its capacity, whatever they are. When the cached objects have heterogeneous sizes, each item may instead be given a storage cost, a positive integer, and each list a cap on the total cost of the items resident in it:

```
cacheNode.set_item_sizes([1, 1, 1, 2, 2, 2])
cacheNode.set_cost_caps([2, 1])
```

A single value passed to `set_cost_caps` declares one cap for the whole cache. The policy is extended as in [27]: a request whose promotion, or whose insertion after a miss, would push a list above its cap is served, but leaves the cache state unchanged. This applies to the miss list as well, so an item too large for the list it would enter is served without being cached.

The state space is a restriction of a reversible chain, so the product form survives. The constrained normalizing constant obeys a recursion on an arbitrary item, which is either absent from the cache or resident in one of the lists; each residency deducts that item's storage cost from the remaining cap of the list it sits in, and a term whose residual cap has gone negative contributes nothing. The probability that an item resides in a given list is the term it contributes to that recursion, normalized by the constant itself, and the mean storage cost a list holds is the cost-weighted sum of those probabilities over the catalogue, reported as the `ListCost` column of the cache table.

`NC` evaluates these expressions with `method='exact'`, or by Monte Carlo summation with `method='sampling'` on larger catalogues; the approximate methods have no cost-capped counterpart and are switched automatically, with a warning. `LDES` simulates the rule directly and is the reference when the caps make a list unreachable: the recursion sums over every size-feasible state, which is a strict superset of the states the cache can visit whenever a cap on an intermediate list blocks a promotion path, and LINE warns when it detects that case.

Figure 6.1 summarizes the node.

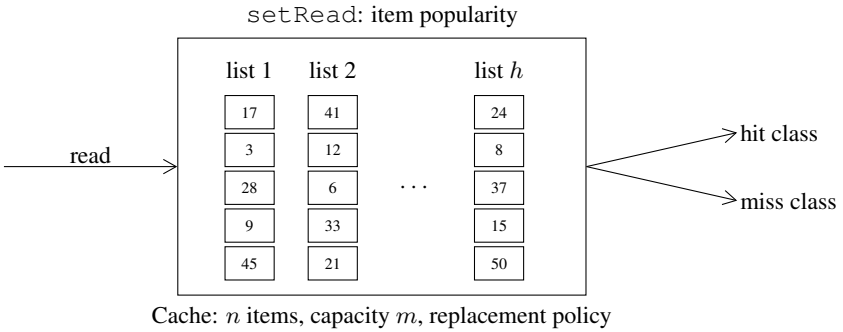


Figure 6.1: The `Cache` node. A read draws an item from the popularity distribution declared with `setRead`; the job then leaves the node in the hit class or in the miss class, and the item lists are updated by the replacement policy. The numbers in the slots are the identifiers of the items currently memorized, all distinct across the h lists.

6.2.3 Cache-based class-switching

An advanced feature of LINE available for example within the `Cache` node, is that the class-switching decision can dynamically depend on the state of the node (e.g., cache hit/cache miss). However, in order to statically determine chains, LINE requires that every class-switching node declares the pair of classes that can potentially communicate with each other via a switch. This is called the *class-switching mask* and it is automatically computed. The boolean matrix returned by the `model.get_class_switching_mask` function provides this mask, which has an entry in row r and column s set to true only if jobs in class r can switch into class s at some node in the network.

Upon cache hit or cache miss, a job in transit is switched to a user-specified class, as specified by the `set_hit_class` and `set_miss_class`, so that it can be routed to a different destination based on whether it found the item in the cache or not. The `set_read` function allows the user to specify a discrete distribution (e.g., `Zipf`, `DiscreteSampler`) for the frequency at which an item is requested. For example,

```
refModel = Zipf(0.5,nitems)
cacheNode.set_read(initClass, refModel)
cacheNode.set_hit_class(initClass, hitClass)
cacheNode.set_miss_class(initClass, missClass)
```

Here `init_class`, `hit_class`, and `miss_class` can be either open or closed instantiated as usual with the `OpenClass` or `ClosedClass` constructors.

6.2.4 Cache networks

A model may hold several `Cache` nodes arranged in a hierarchy, so that a request that misses at one level is served by the next. The difficulty such a model raises is not the number of nodes but the identity of the item

requested. A cache node draws the item referenced by an arriving job from the popularity distribution declared for its class, so a miss routed into a second cache would make that cache sample an item independently of the one that was just missed. Only the aggregate miss rate would then cross the boundary between the two levels, whereas the item-level correlation of the miss stream is precisely the phenomenon a cache hierarchy exists to exhibit.

LINE resolves this by carrying item identity in the job class. Each item is given a dedicated class, whose popularity is a point mass on that item, and the miss class of level l for item f is declared to be the read class of level $l+1$ for the same item f. Item identity is then preserved by ordinary routing, no class switching takes place on the arc between two caches, and the number of classes grows as nL rather than combinatorially in the number of levels L and items n. Throughout we use f as the item index and l as the level index. Item popularity is expressed by the per-class request rates, or by the per-class populations in a closed model, rather than by a distribution the cache draws from, which is also how the analysis of cache networks is usually parameterized in the literature.

Four functions on the `Cache` node build such a model. The cache the exogenous requests enter declares the user's own per-item classes with `set_item_read_classes`. Each subsequent level is attached with `set_miss_cache`, which mints the per-item classes of the next cache, binds the miss class of each item to the corresponding read class there, and registers the cache-to-cache routing arc. That arc is injected when the model is linked, in the same way as the arcs of a retrieval system, since the classes it connects do not yet exist when the user builds the routing matrix. Lastly, the root of the hierarchy is terminated with `set_item_miss_class`, whose misses leave the network towards the origin server. A cache that is fed by another cache but reached by some other construction may mint its classes directly with `set_item_classes`, which `set_miss_cache` itself calls and which is idempotent. All the caches of a network are required to share one item set, and a mismatch in the number of items is refused naming both counts, since padding the shorter catalogue would silently misalign the read probabilities and the item bitmaps.

A tandem of two caches over $n = 3$ items is built as follows.

```
cache1 = Cache(model, 'Cache1', n, 1, ReplacementStrategy.LRU)
cache2 = Cache(model, 'Cache2', n, 2, ReplacementStrategy.LRU)
read, hit1, hit2, miss = [], [], [], []
for f in range(n):
    read.append(ClosedClass(model, f'Read{f+1}', 1, think, 0))
    hit1.append(ClosedClass(model, f'Hit1_{f+1}', 0, think, 0))
    hit2.append(ClosedClass(model, f'Hit2_{f+1}', 0, think, 0))
    miss.append(ClosedClass(model, f'Miss_{f+1}', 0, think, 0))
    think.set_service(read[f], Exp(pAccess[f]))
cache1.set_item_read_classes(read, hit1)
cache1.set_miss_cache(read[0], cache2, hit2)
cache2.set_item_miss_class(read[0], miss)
```

The user then draws only the arcs of the intended topology, namely the request arc into the first cache and the return arcs of the hit and miss classes, one per item. The miss hop between the two caches belongs to the construction and is added automatically. The hit and miss classes may be given either one per item, as above,

or as a single class shared by every item when the per-item split is not of interest.

Each cache reports one row per item in the table returned by `get_avg_cache_table`, the item being named in the `Item` column that this construction adds. The corresponding structural field is `nodeparam{i}.classitem`, which records the item each class reads and is zero for a class that reads none, and which is exchanged as the `itemClass` key of the JSON model interchange. Inferring the same information by inspecting the read probabilities would be ambiguous, since a one-hot popularity is also a legitimate way of describing a single-item catalogue.

Two properties of the construction are worth stating. First, the per-item hit probabilities of the first level reproduce, under the independent reference model, the classical result that an LRU cache of unit capacity holds item f with probability p_f, which offers an exact closed-form check of any solver on the model. Second, an LRU cache of unit capacity at both levels makes the second level unreachable, in the sense that it can never register a hit. A miss at the first level inserts the requested item there, so the next miss is necessarily for a different item and consecutive arrivals at the second level are always distinct. A hierarchy of unit caches is therefore exclusive by construction, and the second level must be given room for more than one item before it can exhibit any hit at all.

Cache networks are solved by `CTMC`, which is exact on them, and by the simulation solvers `SSA` and `LDES`, the latter treating hits and misses as exact sample-path events under every replacement policy it supports. They may not be solved by `JMT`. Its cache section admits only a parametric popularity, either Zipf or uniform, and has no empirical discrete form, so a point-mass popularity has no faithful representation there. Rather than export such a model as a uniform draw over the whole catalogue, and thereby solve a different model without saying so, LINE refuses it naming the cache and the class concerned.

6.3 Stochastic Petri nets

6.3.1 Place and Transition nodes

LINE supports stochastic Petri net modeling via `Place` and `Transition` nodes. A `Place` holds tokens representing jobs or resources, while a `Transition` fires when its enabling conditions are met, consuming tokens from input places and producing tokens at output places. Transitions support multiple firing modes, each with its own enabling/inhibiting conditions, firing priorities, and firing rate distributions. Modes allow a single transition to represent different behaviors depending on system state. For example:

```
place = Place(model, 'Buffer')
trans = Transition(model, 'Process')
fast_mode = trans.add_mode('fast')  # returns Mode object
slow_mode = trans.add_mode('slow')
trans.set_distribution(fast_mode, Exp(2.0))  # fast processing mode
trans.set_distribution(slow_mode, Exp(0.5))  # slow processing mode
trans.set_enabling_conditions(fast_mode, jobclass, place, 1)  # requires 1 token
trans.set_enabling_conditions(slow_mode, jobclass, place, 1)
```

Table 6.1: Per-mode setters of a `Transition` node

Setter	Effect on the mode
`set_distribution`	Firing time distribution
`set_enabling_conditions`	Token multiplicities required on the input arcs
`set_inhibiting_conditions`	Token-level inhibition arcs
`set_firing_priorities`	Integer priority breaking ties among enabled modes
`set_firing_weights`	Weight randomizing the choice among modes of equal highest priority
`set_number_of_servers`	Per-mode server count; `float('inf')` gives an infinite-server transition
`set_timing_strategy`	With `TimingStrategy.IMMEDIATE`, marks the mode immediate and replaces its firing distribution by the `Immediate` singleton

Each mode of a `Transition` is configured through the per-mode setters of Table 6.1. Their full use is illustrated in the dedicated Petri-net section (§6.3.3); worked models are `spn_basic_open`, `spn_twomodes` and related files in the `examples/` folder.

Figure 6.2 shows the graphical conventions used for these elements.

6.3.2 Queueing places

By default a `Place` is an *ordinary place*: a token deposited by an input transition is immediately available to the output transitions. LINE also supports *queueing places* in the sense of queueing Petri nets (Bause, 1993), where a scheduling station is embedded inside the place. A place becomes a queueing place as soon as a service process is assigned to a token colour with `set_service`: an arriving token then enters an embedded queue served under the place's scheduling strategy and, upon service completion, moves to a *depository* from which it becomes available to the output transitions. The scheduling strategy is chosen through the optional third constructor argument, and the number of embedded servers through `set_number_of_servers`. For example, a processor-sharing CPU place and an infinite-server think place:

```
cpu = Place(model, 'CPU', SchedStrategy.FCFS)
cpu.set_service(jobclass, Exp(2.0))    # embedded FCFS queue, rate 2
cpu.set_number_of_servers(2)           # two embedded servers (M/M/2)
```

The order in which depository tokens become available to the output transitions is governed by the departure discipline (`DepartureDiscipline.NORMAL`, the default, or `DepartureDiscipline.FIFO`), set with `set_departure_discipline`. Queueing places are simulated by the `LDES` solver, which supports the `FCFS`, `LCFS`, `SIRO` and `INF` embedded scheduling strategies (single- or multi-server) with exponential, deterministic, immediate and renewal phase-type service. The result reported for the place (its token count) is the total marking, i.e. the sum of the tokens waiting, in service and in the depository. Because a queueing place embeds a service station inside a Petri-net place, it cannot be represented as a single node in `JMT` (whose engine keeps token storage and job service in separate nodes); a queueing-place model is therefore rejected by `SolverJMT` and must be expanded into a place plus an adjacent queueing station to be simulated there.

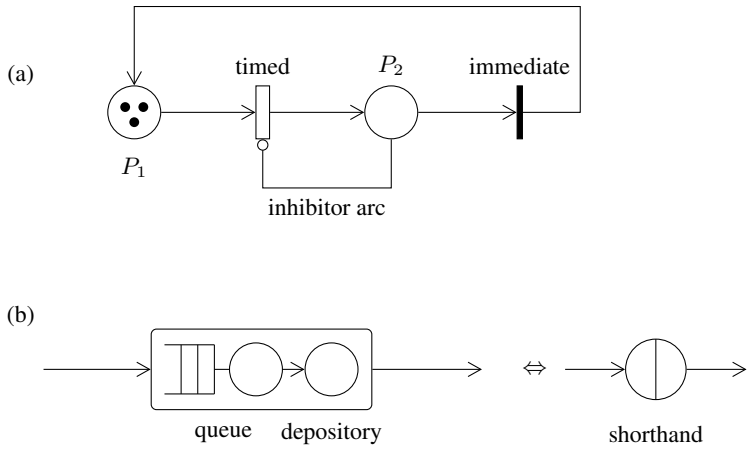


Figure 6.2: Petri-net elements. (a) A timed transition (hollow bar) and an immediate transition (solid bar), with an inhibitor arc that disables the timed transition while P_2 is marked. (b) A queueing place: an arriving token is first served by the embedded queue and only then moves to the depository, where it becomes available to the output transitions; the shorthand symbol on the right is the circle of an ordinary place bisected by a vertical bar.

6.3.3 Petri net specification

LINE supports the specification of stochastic Petri net (SPN) models through dedicated `Place` and `Transition` node types, providing an alternative modeling paradigm to queueing networks. In an SPN, the system state is represented by the distribution of tokens across places, and the system dynamics are governed by transition firings that consume tokens from input places and produce tokens in output places. This formalism is particularly well-suited for modeling systems with synchronization, resource sharing, and complex control flow that may be difficult to express naturally in the queueing network framework.

A `Place` is a stateful node that holds tokens, which correspond to jobs in the queueing network interpretation. Places can have capacity limits and initial token populations. A `Transition` is a node that defines one or more firing modes, each representing a distinct way the transition can fire. Each mode has its own timing distribution, which governs the delay between enabling and firing, its own enabling condition, which specifies the number of tokens required in each input place, and its own firing outcome, which specifies the number of tokens deposited in each output place when the transition fires. The ability to define multiple modes on a single transition allows modeling of complex state-dependent behaviors and routing decisions.

Internal sections of Place and Transition nodes. Like every node in LINE, a `Place` and a `Transition` are decomposed into three internal sections, listed in Table 6.2, that determine how tokens are accepted, processed and dispatched. The sections are instantiated by the constructors and are not manipulated directly; they are described here because they name the entries appearing in the `nodeparam` tables of §3.2.2.

Table 6.2: Internal sections of `Place` and `Transition` nodes

Node	Section	Role
`Place`	`Storage` (input)	Token buffer. Accepts tokens deposited by upstream firings and applies the per-class capacity set with `set_class_capacity` (unbounded by default). The policy is non-preemptive, since tokens in an ordinary place are not served
	`ServiceTunnel` (server)	Pass-through with no service delay, which is what makes an ordinary place a passive marker store rather than a station
	`Linkage` (output)	Dispatcher towards the downstream transitions. Each class defaults to `RoutingStrategy.RAND`, overwritten by the routing matrix installed with `model.link(P)`
`Transition`	`Enabling` (input)	Evaluates the per-mode enabling and inhibiting conditions
	`Timing` (server)	Draws the firing delay from the per-mode distribution
	`Firing` (output)	Applies the per-mode firing outcome to the connected places

The following example demonstrates the creation of a simple SPN model. A place `P1` and a transition `T1` are created, and a firing mode is added to the transition. The mode is configured with an exponential firing delay, an enabling condition requiring one token in `P1`, and a firing outcome producing one token in a sink node.

```
P1 = Place(model, 'P1')
T1 = Transition(model, 'T1')
mode = T1.add_mode('Mode1')
T1.set_number_of_servers(mode, float('inf'))
T1.set_distribution(mode, Exp(4))
T1.set_enabling_conditions(mode, jobclass1, P1, 1)   # requires 1 token in P1
T1.set_firing_outcome(mode, jobclass1, sink, 1)      # produces 1 token in sink
```

In addition to standard enabling conditions, SPN models support inhibiting arcs, which prevent a transition from firing when a place contains too many tokens. This mechanism is useful for modeling resource constraints or conditions where an event should not occur when a system is in a particular state. The `set_inhibiting_conditions` method specifies that a transition cannot fire when a designated place holds more than a specified threshold number of tokens.

```
T1.set_inhibiting_conditions(mode, jobclass1, P2, 2)  # inhibit if P2 has >2 tokens
```

SPN models in LINE are analyzed by default with the `JMT` solver, which translates the Petri net specification into its native simulation engine and performs discrete-event simulation to estimate performance metrics; the `CTMC`, `SSA`, and `LDES` solvers can also analyze models containing `Place` and `Transition` nodes. This allows SPN models to benefit from `JMT`'s efficient simulation algorithms while retaining the expressive power of the Petri net formalism. The examples `spn_basic_open`, `spn_twomodes`, `spn_inhibiting`, and `spn_closed_fourplaces` demonstrate various applications of stochastic Petri nets, including open and closed systems, multi-mode transitions for state-dependent routing, inhibiting arcs for resource constraints, and complex token flows.

6.3.4 PNML import and export

A Petri net built in LINE can be written to, and read back from, a PNML document (ISO/IEC 15909-2) in the place/transition grammar http://www.pnml.org/version-2009/grammar/ptnet, which is the interchange format of the tools built around that corpus, among them GreatSPN, TINA, PIPE and the Model Checking Contest harnesses.

```
save_pnml(model, 'net.pnml')
model = load_pnml('net.pnml')          # first net of the document
model = load_pnml('net.pnml', 'n1')    # the net with the given id
```

The place/transition grammar is *uncoloured*, so it carries less than LINE can express, and the difference is refused rather than approximated. Writing fails, naming the construct, on more than one job class, whose tokens would make the net coloured; on an open class, a `Source` or a `Sink`, for which the grammar has no unbounded token source; on a queueing place, which is a station rather than a place; on a firing-rate dependence, which no PNML element can carry; and on a distribution outside the scalar-parameter families, such as a phase-type law or a Markovian arrival process given by matrices.

Timing rides in a `toolspecific` block, which is where the grammar puts what it does not define. Each LINE firing mode becomes one PNML transition, so that the arcs of a mode are the arcs of a transition as the grammar requires, and the block records which LINE transition and mode it came from, so that the reader regroups the modes it split. A reader that ignores the block still sees a correct untimed net. In the other direction the grammar says nothing about how long a transition takes to fire, so a file written by another tool is read with every transition timed and exponential at rate 1, the convention of the stochastic Petri net literature and GreatSPN's own default, and its tokens become a single `ClosedClass` holding the initial marking. A file written by LINE reads back with the distributions, servers, priorities and weights it was written with. Inhibitor arcs are read from the `<type value="inhibitor"/>` extension that GreatSPN, TINA and PIPE all write, the grammar itself having no inhibitor arc.

A PNML document is also accepted directly by the command-line front ends, as `-i pnml` or, where the format is inferred from the path, a bare `.pnml` file. Only the solvers whose feature set declares `Transition` can answer for it, namely `CTMC`, `SSA`, `JMT` and `LDES`:

```
line -f net.pnml -s ctmc -a avg
```

6.4 Networks with signals

6.4.1 Signal classes

Networks with signals are supported by the `LDES` and `SSA` simulators, by the `CTMC` solver, which encodes signal arrivals and the induced job removals directly in the state-transition rates, and by the `AG` solver (§8.2.1), whose agent-based analyzer is the only analytical one that reads the signal marking. No `MAM` method reads it:

Table 6.3: Signal types available in `Network` models.

SignalType	Description
`NEGATIVE`	Removes a random number of jobs from the queue according to a removal distribution (default: 1 job)
`CATASTROPHE`	Removes all jobs from the queue upon arrival
`REPLY`	Triggers a reply action for synchronous call semantics

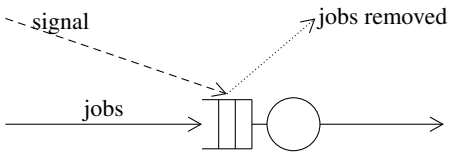


Figure 6.3: Signal classes. A signal arriving at a station does not queue: it removes jobs already present, a random number of them for `NEGATIVE` and the whole queue for `CATASTROPHE`.

each would answer with every signal turned into an ordinary customer, so `MAM` refuses such a model by name and points at `AG` instead. Table 6.3 summarizes the signal types available in LINE.

All signal classes are created using the `Signal` class with a required `SignalType` parameter. A `Signal` class can be configured with:

- **Removal distribution**: A discrete distribution specifying how many jobs to remove (default: 1 job). An alternative to the default is a `Geometric` distribution for batch removals.
- **Removal policy**: Determines which jobs are selected for removal when multiple jobs are present:
  - `RemovalPolicy.RANDOM` – Select jobs uniformly at random (default)
  - `RemovalPolicy.FCFS` – Remove the oldest job in the station first
  - `RemovalPolicy.LCFS` – Remove the newest job in the station first

Figure 6.3 illustrates the effect of a signal arrival.

The following example shows how to create a negative signal class:

```
neg_signal = Signal(model, 'NegativeCustomer', SignalType.NEGATIVE,
                    removal_distribution=Geometric(0.5),  # batch removal
                    removal_policy=RemovalPolicy.LCFS)    # remove newest first
source.set_arrival(neg_signal, Exp(0.1))
```

A catastrophe signal removes all jobs from a queue. Use `SignalType.CATASTROPHE`:

```
disaster = Signal(model, 'Disaster', SignalType.CATASTROPHE)
source.set_arrival(disaster, Exp(0.01))  # rare catastrophic events
```

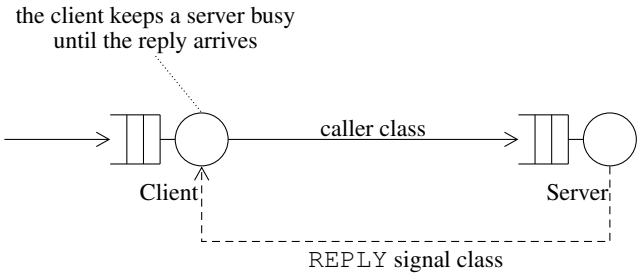


Figure 6.4: A synchronous call. The caller class is routed from the client to the server station and is switched to the `REPLY` signal class on departure from it. The reply releases the server the caller holds at the client, which resumes from where it stopped.

Synchronous calls. A signal of type `SignalType.REPLY` carries the return leg of a synchronous call, that is a call whose caller stops and waits for the callee to finish. The signal is bound to the calling class with `for_job_class`, which names the class whose server is released when the reply arrives. The canonical topology, shown in Figure 6.4, routes the caller from the client station to the callee and switches it to the reply class on departure from the callee, so that the reply travels back to the client. The client keeps a server busy for the whole call, which is what makes the call synchronous, while the callee is not blocked, since its own departure is what creates the reply. The holding station is identified structurally as the station the reply class is routed into, and is recorded in the fields `syncreply` and `replyblock` of the network structure. Because a held server is encoded as a per-class counter, which is exact only where servers are interchangeable or unlimited, the holding station must use `FCFS` or `INF` scheduling. Synchronous calls are also how `LQN2QN` (Section 6.5) flattens the calls of a layered network into an ordinary `Network`.

6.4.2 Signal semantics

LINE supports the modeling of networks with signals through the `Signal` class. Signals extend traditional queueing networks by introducing negative customers, which have the distinctive property of removing positive customers from queues upon arrival rather than joining them. This mechanism provides a natural way to model phenomena such as job cancellations, cache invalidations, service interrupts, or resets in computing systems. The interaction between positive and negative customers creates rich dynamics that capture feedback control, load regulation, and catastrophic events in networked systems.

A signal is defined using the `Signal` constructor, which takes a model reference, a descriptive name, and a signal type that specifies the behavior of the signal class. The `SignalType.NEGATIVE` type creates a negative customer class with the semantics that each arrival at a queue removes one positive customer from that queue. If the queue is empty when a negative customer arrives, the negative customer has no effect and is simply absorbed.

```
neg_class = Signal(model, 'Negative', SignalType.NEGATIVE)
source.set_arrival(neg_class, Exp(lambda_neg))
```

In this example, a negative customer class is created and assigned an arrival process at a source node. When negative customers arrive at queues in the network, they reduce the queue populations by removing positive customers. The routing of negative customers is specified using the same mechanisms as for ordinary job classes, allowing complex patterns of cancellation and control.

```
solver = AG(model, method='inap')
QN, UN, RN, TN, AN, WN = solver.get_avg()
```

The example `ag_gnetwork` demonstrates the construction and analysis of a network with signals featuring both positive customers and negative arrivals, illustrating how signal classes interact with ordinary job classes to produce non-trivial equilibrium behavior. Networks with signals are particularly useful for modeling systems with dynamic load control, self-regulation, or reset mechanisms, where the conventional queueing network paradigm does not naturally capture the feedback effects of cancellations and interruptions.

6.5 Layered network models

In this section, we present the definition of the `LayeredNetwork` class, which encodes the support in LINE for a class of generalized layered stochastic networks. In their basic form, these models are called layered queueing networks (LQNs) and differ from regular queueing networks as servers, in order to process jobs, can issue synchronous and asynchronous calls among each others. We point to [59] and to the LQNS user manual for an introduction [60]. Contrary to the original LQNs, layered networks in LINE can also include non-queueing servers, such as caches, hence they may be conceptualized as more general layered stochastic networks.

The topology of call dependencies in a layered network makes it possible to partition the model into a set of layers, each consisting of a subset of the servers. Each of these layers is then solved in isolation, updating with an iterative procedure its parameters and performance metrics until the layers solutions jointly converge to a consistent solution.

6.5.1 Basics about layered networks

Layered network models describe a collection of resources called *tasks*, each representing for example a software server, that run on resources called *host processors*. Classes of service exposed by a task are called *entries*. Each entry is an endpoint at which a task can be invoked; for example, if a task represents a web server then its web pages may be described as different entries.

A special task, called the *reference task* is used to represent a group of system users. In this case, the host processor for a reference task can either be real, as in the case of users that are themselves software systems, or fictitious, as in the case of human users.

Figure 6.5 shows a small layered network.

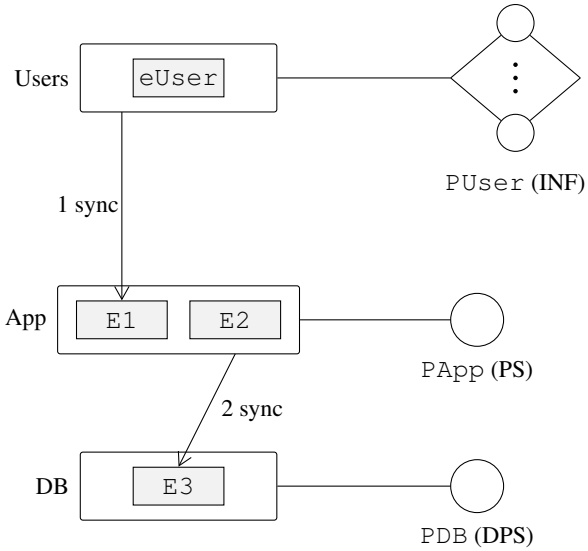


Figure 6.5: A layered network with three tasks. Each task exposes one or more entries and runs on a host processor, drawn to its right; an arc between two tasks is a synchronous call from an entry of the caller to an entry of the callee, labelled by its mean multiplicity.

Each entry can be specified by a workflow of operations called *activities*, typically organized as a directed acyclic graph. The time demand that each activity places at the underpinning host processor is called its *host demand* and it is a random variable with a user-specified distribution.

Figure 6.6 shows the precedences an activity graph can express.

Activity graphs may include *calls* to entries exposed by other tasks. This is an abstraction of the calls that distributed system components have among themselves. Calls can be *synchronous*, *asynchronous*, or *forwarding*. Synchronous calls are requests that block the sender until a reply is received, while asynchronous calls are non-blocking and the sender execution can continue after issuing the call. Calls can either be repeated either *deterministic* or *stochastic*, meaning in the latter case that the number of calls issued is a random variable, e.g. geometrically distributed.

Forwarding calls provide a mechanism for delegation chains in layered networks. When a task receives a synchronous request, instead of replying directly to the caller, it can forward the request to another entry with a given probability. The forwarding task is released immediately (it does not block waiting for the downstream reply), while the original caller remains blocked until the final entry in the forwarding chain sends the reply. This semantics is useful for modeling multi-tier systems where a front-end server dispatches requests to a back-end without itself waiting for the result—the client blocks end-to-end, but intermediate servers are freed. Forwarding is specified at the entry level using the `forward` method:

```
E1.add_forwarding(E2, 0.8)  # forward 80% of requests from E1 to E2
```

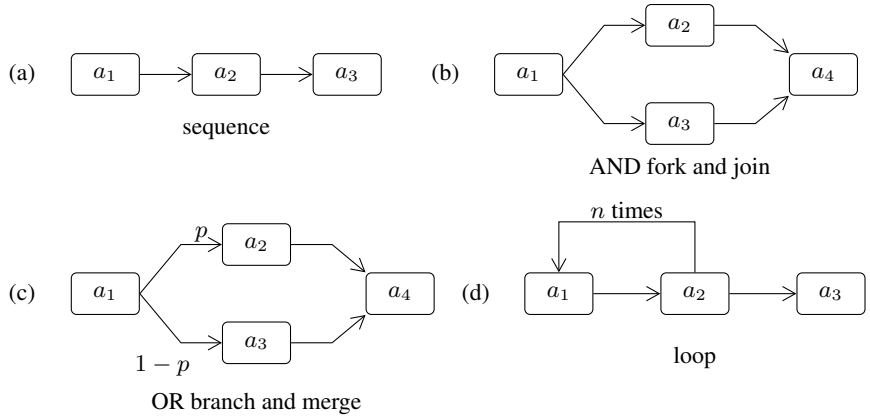


Figure 6.6: Activity graph precedences. (a) A sequence, each activity enabling the next. (b) An AND fork, whose branches run concurrently and are synchronized by an AND join. (c) An OR branch, taking one successor with probability p, merged by an OR join. (d) A loop, repeating an activity a mean number n of times before continuing.

Multiple forwarding destinations can be specified for the same entry, provided that the sum of forwarding probabilities does not exceed one. The remaining probability mass corresponds to the entry replying directly to its caller. Forwarding calls are serialized in the LQNX format using the `<forwarding>` XML element. They are supported by the `LQNS` and `LQSIM` wrappers and by the native `LN` solver, which represents a forwarding call as a caller-side pseudo rendezvous: the forwarding chain is rebuilt so that the original client remains blocked until the last entry in the chain replies, and phase-2 and multiple-destination forwarding are handled by the same construction.

Contrary to ordinary layered queueing networks, a layered network in LINE can also feature *cache tasks*, *item entries*, and *cache-access* precedence relations.

- Cache tasks have the basic properties of tasks, but add three specific properties for caching: the total number of items, the cache capacity and the cache replacement policy. Cached items can be either contents or services. Cache capacity indicates the storage constraints of the cache.
- An item-entry provides instead access to a group of entries of a cache, Item-entries have the basic properties of entries, but add the property of the popularity of the items they give access to.
- A precedence relationship called *cache-access* is defined for the cache hit and miss activities under each item-entry. That is, it is possible to proceed to a different activity depending on whether the cache access produced a cache hit or cache miss. For example, a cache miss can produce a call to a remote entry to retrieve the missing content.

Note that the above extensions are not queueing-based and this explains why these models are referred to in LINE as layered networks and not as layered queueing networks. Similar to the latter, the analysis of a layered networks uses a decomposition of the model into a set of submodels, each being a object, which are

then iterative analyzed using different solution methods.

6.5.2 LayeredNetwork object definition

Creating a layered network topology

A layered queueing network consists of four types of elements: processors, tasks, entries and activities. An entry is a class of service specified through a finite sequence of activities, and hosted by a task running on a (physical) processor. A task is typically a software queue that models access to the capacity of the underpinning processor. Activities model either demands required at the underpinning processor, or calls to entries exposed by some remote tasks.

In the `LayeredNetwork` class, the terms *host* and *processor* are entirely interchangeable.

To create our first layered network, we instantiate a new model as

```
model = LayeredNetwork('myLayeredModel')
```

We now proceed to instantiate the static topology of processors, tasks and entries:

```
P1 = Processor(model, 'P1', 1, SchedStrategy.PS)
P2 = Processor(model, 'P2', 1, SchedStrategy.PS)
T1 = Task(model, 'T1', 5, SchedStrategy.REF).on(P1)
T2 = Task(model, 'T2', float('inf'), SchedStrategy.INF).on(P2)
E1 = Entry(model, 'E1').on(T1)
E2 = Entry(model, 'E2').on(T2)
```

An equivalent way to specify the above example is to use the `Host` class rather than the `Processor` class, with identical parameters.

In the above code, the `on` method specifies the associations between the elements, e.g., task `T1` runs on processor `P1`, and accepts calls to entry `E1`. Furthermore, the multiplicity of `T1` is 5, meaning that up to 5 calls can be simultaneously served by this element (i.e., 5 is the multiplicity of servers in the underpinning queueing system for `T1`).

Both processors and tasks can be associated with the standard LINE scheduling strategies. For instance, `T2` will process incoming requests in parallel according as an infinite server node, since we selected the `SchedStrategy.INF` scheduling policy. An exception is that `SchedStrategy.REF` should be used to denote the reference task (e.g. a node representing the clients of the models), which has a similar meaning to the reference node in the object.

Tasks with setup and delay-off times

A task whose servers are switched off while idle does not begin service as soon as a request arrives: the server must first be reactivated, and it is switched off again only after it has been idle for some time. LINE represents this behaviour on any `Task` through two timing properties:

- Setup time: the activation delay paid by a server that resumes from the off state before it can serve the request that woke it up.
- Delay-off time: the idle period a server remains available after emptying its queue, before it powers off. A further arrival within this window is served without paying the setup time again.

These are the setup and close-down times of a server with vacations, and they arise wherever activation is not instantaneous: virtual machines and containers started on demand, power-managed servers under a timeout policy, machines requiring a tool change or a warm-up, and serverless (function-as-a-service) platforms, where the setup time is the cold-start delay and the delay-off time is the keep-alive window. Such a task is instantiated as follows:

```
F1 = SetupTask(model, 'F1', 4, SchedStrategy.FCFS).on(P1)
F1.set_setup_time(Exp(1.0))       # activation time (mean = 1.0)
F1.set_delay_off_time(Exp(2.0))   # idle time before power-off (mean = 2.0)
```

Both `setSetupTime` and `setDelayOffTime` accept either a numeric value (which creates an exponential distribution with the given mean) or a `Distribution` object. Both default to `Immediate`, i.e., a task that sets neither is an ordinary always-on task. Since the two properties belong to `Task`, the `SetupTask` subclass, used above, is only a naming convenience that makes the intent explicit; a plain `Task` carrying either time behaves identically. The former name of this subclass, `FunctionTask`, is retained as a deprecated alias so that existing serverless models keep working.

The same setup and delay-off semantics are available at the level of a flat `Network`, without a layered model, through `Queue.set_delay_off(jobclass, setup_time, delay_off_time)`, which attaches a per-class setup distribution and delay-off distribution to a queueing station. The two distributions can be read back with `get_setup_time` and `get_delay_off_time`. A model that uses them declares the `SetupDelayOff` language feature and is simulated by the `LDES` solver. A complete example is provided in `lqn_setup` in the `examples/` folder.

Cache tasks

The `CacheTask` class extends `Task` to model in-memory caches embedded in a layered network. A cache task stores up to a finite number of items chosen out of a catalogue, evicting them according to a configurable replacement policy. Cached items are exposed to the rest of the model as `ItemEntry` objects, which behave like ordinary entries but are additionally annotated with a discrete popularity distribution describing how frequently each item is requested.

The `CacheTask` constructor takes the following arguments:

- `model`: the parent `LayeredNetwork`;
- `name`: a unique task name;
- `nitems`: the catalogue size, i.e., the total number of distinct items that may be requested;
- `itemLevelCap`: cache capacity. A scalar value specifies the size of a single-level cache, while a row vector specifies per-level capacities for a multi-level cache hierarchy (e.g., `[2,4]` for an L1 of size 2 and

an L2 of size 4);

- `replStrat`: a `ReplacementStrategy` enum value. The supported policies are `ReplacementStrategy.RR` (random replacement), `ReplacementStrategy.FIFO`, `ReplacementStrategy.SFIFO` (strict FIFO), and `ReplacementStrategy.LRU`. Note that this is narrower than the policy set accepted by a flat `Cache` node, which additionally supports `HLRU`, `CLIMB` and `QLRU`;
- delayed-hit retrieval, enabled with `set_retrieval`. When it is set, concurrent misses for the same item that arrive while a fetch is already in flight (the fetch being the miss-branch activity together with its backend calls) are parked and released together as delayed hits when that fetch completes, instead of each triggering an independent fetch. Without it, every miss issues its own fetch. Layers that declare retrieval are solved by a dedicated analytic sublayer, and support for them is currently experimental;
- `multiplicity` (optional, default 1): number of concurrent threads that may serve cache accesses;
- `scheduling` (optional, default `SchedStrategy.FCFS`): scheduling discipline used when multiple requests compete for the cache.

An `ItemEntry` extends `Entry` with a popularity descriptor and is constructed as `ItemEntry(model, name, cardinality, popularityDistribution)`, where `cardinality` is the number of items reachable through the entry and `popularityDistribution` is a discrete `Distribution` object (e.g., `DiscreteSampler` or `Zipf`). The hit/miss outcome of a cache access is then routed in the activity graph using the `ActivityPrecedence.CacheAccess` construct (see Section 6.5.2).

The following example defines a single-host LQN where a client task `T1` issues calls to a cache task `C2` hosting four items with capacity two and a round-robin eviction policy: The two post-cache activities passed to `CacheAccess` are interpreted in order: the first is executed on a cache hit, the second on a cache miss. A complete worked example is given in `lcq_singlehost` in the `examples/advanced/layeredCQ/` folder; `lcq_threehosts` illustrates a hierarchical, multi-tier cache configuration.

Replication and fan-out

Layered networks support replication of processors and tasks, together with explicit fan-in and fan-out annotations on synchronous and asynchronous calls. Replication multiplies a single declared element into a number of identical copies that share the same activity graph but provide independent service capacity, and is the recommended way to model horizontally scaled microservices, clusters, or any tier with a number of equivalent instances.

The `set_replication` method on `Processor` and `Task` sets the replica count $r \geq 1$. The `set_fan_out(dest, value)` method on `Task` declares that each replica of the calling task issues calls to `value` replicas of the called task `dest`; symmetrically, `set_fan_in(source, value)` declares that each replica of the called task receives calls from `value` replicas of the source task. The product of fan-in and fan-out values must be consistent with the replica counts of the two tasks involved in each call. The `lqn_sockshop` example in the `examples/basic/layeredModel/` folder demonstrates a microservice-style topology in which an edge task `T1` is replicated $2\times$ and front-end / back-end tasks declare fan-in

and fan-out values around it: Fan-out destinations and fan-in sources are referenced by task name, so the corresponding tasks must already exist (or be created later in the model). Multiple fan-out destinations may be specified by repeated calls to `setFanOut`; each call appends a new `(destination, value)` pair.

Servers with queue-dependent service rates

When `LN` decomposes a layered model, every processor and every non-reference task becomes a queueing station in the layer where it acts as a server. The three queue-dependent service mechanisms of §5.2.1–§5.2.4, which for a `Network` model are attributes of a `Queue`, may be declared directly on that server, so that a host can slow down as its tasks pile up, or a task can serve one of its entries at a rate that depends on how many requests of each kind it currently holds. The setters carry the same names as on a queue and are available on `Processor`/`Host` and on `Task`:

- `set_load_dependence(alpha)` takes the numeric vector $\alpha(n)$ that scales the rate of the layer station when it holds n jobs in total, exactly as for a queue. The scaling multiplies the station rate on top of the multiplicity, so a host of multiplicity m applies $\min(n, m)\,\alpha(n)$.
- `set_class_dependence(beta, peak)` takes the product-form handle $\beta(n)$, and `set_joint_dependence(eta, peak)` the non-product-form handle $\eta(n)$. Their argument is the per-*operand* population vector of the server: $n(j)$ counts the jobs the layer station holds on behalf of operand j, which is task j of a processor or entry j of a task, in declaration order. The handle returns a scalar shared by every operand or a per-operand vector, and the peak rate is required, as it normalizes utilization as $U = T\,S/\text{peak}$.

Declarations are admitted only on `PS` and `FCFS` servers, the disciplines whose layer station admits a rate scaling; any other scheduling strategy is rejected at declaration time. For example, a processor that gains capacity as its tasks queue up, and a task whose first entry is served by two internal workers:

```
P1 = Processor(model, 'P1', 1, SchedStrategy.PS)
P1.set_load_dependence([1, 1.6, 1.9, 2, 2])   # alpha(n), n = 1..5
T1 = Task(model, 'T1', 5, SchedStrategy.FCFS).on(P1)
E1 = Entry(model, 'E1').on(T1)
E2 = Entry(model, 'E2').on(T1)
T1.set_class_dependence(lambda n: [min(n[0], 2), 1], [2, 1])
```

The declarations are stored in the `LayeredNetworkStruct` fields `lldscaling`, `cdscaling` and `jdscaling` and are applied by `LN` while it builds the layers, so any layer solver that supports the mechanism can then be used: `MVA` through its queue-dependent arms and `NC` through the multichain convolution. A dependence declared over operands is lifted onto the classes of the layer, since a station speaks classes and an operand may occupy the station through several of them. Where each operand maps to exactly one class the lift is faithful and a class dependence remains product form; otherwise `LN` emits the same numbers as a joint dependence, which is not accompanied by an exactness guarantee. A scalar handle applies to every class of every operand, while classes that belong to no operand keep a unit scaling.

Two limitations follow from the interchange formats. The LQN schema has no element for a rate scaling,

so `write_xml` warns and omits the declaration, and a model exported to `.lqnx` (hence also one solved by the `LQNS` wrapper) is analysed without it. Under `options.lang='java'` the load-dependence vector is marshalled to the JAR, whereas the class- and joint-dependence handles are not transferable across the language boundary and are rejected with an explicit error; solve those models natively instead.

Servers with class compatibilities

A processor or a task is by default a *homogeneous* pool: any of its servers may run any operand. When only some are eligible – a core pool in which only some cores carry an accelerator – the multiplicity is partitioned into *pools*, each naming the operands it serves, through the `ServerType` class of §5.1.2:

```
P1 = Processor(model, 'P1', 3, SchedStrategy.PS)
T2 = Task(model, 'T2', 6, SchedStrategy.FCFS).on(P1)
T3 = Task(model, 'T3', 6, SchedStrategy.FCFS).on(P1)
P1.addServerType(ServerType('S1', 1, [T2]))
P1.addServerType(ServerType('S2', 1, [T2, T3]))
P1.addServerType(ServerType('S3', 1, [T3]))
```

The pools partition the servers, so their counts must sum to the multiplicity and every operand must be reachable; both are checked when the struct is built. A pool's relative speed, one by default, is per pool and not per (pool, operand), since a server free to take several operands does not announce which it picked.

The station clears the *activated-server* rate

$$\mu(n) \;=\; \sum_{t} k_t\, r_t\, \min\Big(1, \textstyle\sum_{j\,:\,\mathrm{compat}(t,j)\neq 0} n_j\Big), \tag{6.1}$$

where pool t holds k_t servers of rate r_t: a pool contributes its full rate as soon as a compatible operand is present. At integer states this depends on n only through its support, the order-independent condition; the $\min(1,\cdot)$ matters only at the fractional argument a mean-value solver supplies, without which a barely-busy operand would activate every pool it touches.

`LN` lowers it to a joint dependence normalised by the peak $\sum_t k_t r_t$ of (6.1), so a fully compatible pool reproduces the neutral scaling. That makes the declaration an *approximation* within a layer and confines it to the class-switching layerings `srvn.cs` and `flat.cs`; `srvn.ph` refuses it. The LQN schema cannot express a compatibility graph either, so an `.lqnx` export warns and reads back homogeneous, and `LQNS` refuses the model.

Describing host demands of entries

The demands placed by an entry on the underpinning host (also called in layered queueing networks the *host demand*) is described in terms of execution of one or more activities. Although in tools such as LQNS activities can be associated with either entries or tasks, LINE supports only the more general of the two options, i.e., the definition of activities at the level of tasks. In this case:

- Every task defines a collection of activities.

- Every entry needs to specify an initial activity where the execution of the entry starts (the activity is said to be "bound to the entry") and a replying activity, which upon completion terminates the execution of the entry.

For example, in our running example, we may now associate an activity to each entry as follows:

```
A1 = Activity(model, 'A1', Exp(1.0)).on(T1).bound_to(E1).synch_call(E2, 3.5)
A2 = Activity(model, 'A2', Exp(2.0)).on(T2).bound_to(E2).replies_to(E2)
```

Here, `A1` is a task activity for `T1`, acts as initial activity for `E1`, consumes an exponentially distributed time on the processor underpinning `T1`, and requires on average 3.5 synchronous calls to `E2` to complete. Each call to entry `E2` is served by the activity `A2`, with a demand on the processor hosting `T2` given by an exponential distribution with rate $\lambda = 2.0$.

Activity graphs Often, it is useful to structure the sequence of activities carried out by an entry in a graph. Activity graphs can be characterized by precedence relationships of the following kinds:

- *sequence*: two activities are executed sequentially, one after each other. This is implemented through the `ActivityPrecedence.Serial` construct.
- *loop*: an activity is repeated a number of times. This is implemented in `ActivityPrecedence.Loop`.
- *and-fork*: a serial execution is forked into concurrent activities. This can be materialized using the `ActivityPrecedence.AndFork` construct.
- *or-fork*: the server chooses probabilistically which activity to execute next among a set of alternatives. This is implemented in `ActivityPrecedence.OrFork`.
- *and-join*: concurrent activities are joined into a single serial execution. This is implemented in `ActivityPrecedence.AndJoin`.
- *or-join*: merge point for alternative activities that may execute in parallel after a *or-fork*. This is implemented in `ActivityPrecedence.OrJoin`.
- *cache-access*: split point for cache hit/cache miss results in an activity graph. This is implemented in `ActivityPrecedence.CacheAccess`. For usage examples, see `cache_replc_lru` and `cache_compare_replc` in the `examples/` folder.

A composite example showing fork/join precedences and loops is given in `lqn_workflows` in the `examples/` folder.

An AND-fork precedence spawns several execution paths that proceed independently until they converge at the matching AND-join, which blocks until every spawned branch has completed and then resumes a single thread of control. A branch may contain any sequence of activities, including processor demands, synchronous calls to lower-layer entries, and nested fork-join structures. The construct differs from the fork-join of ordinary queueing networks in that activities are bound to tasks and processors, so contention arises at both levels; and because branches may call entries in different layers, the response time of the forking task is the maximum over branch completion times, each of which depends in turn on lower-layer

response times. An entry is therefore complete only when all activities in its activity graph, including every branch of an AND-fork, have synchronized.

Solvers evaluate these structures by treating each fork-to-join branch as a subnetwork, estimating its completion-time distribution, and deriving the distribution of the maximum, which is the synchronization delay at the join. `LN` approximates that maximum by moment matching on matrix-geometric branch representations, whereas `LQNS` decomposes the model layer by layer and refines the parameterization iteratively. The `lqn_workflows` example exercises several AND-fork/AND-join pairs spanning multiple layers.

For instance, we may replace in the running example the specification of the activities underpinning a call to `E2` as

```
A20 = Activity(model, 'A20', Exp(1.0)).on(T2).bound_to(E2)
A21 = Activity(model, 'A21', Erlang.fit_mean_and_order(1.0, 2)).on(T2)
A22 = Activity(model, 'A22', Exp(1.0)).on(T2).replies_to(E2)

T2.add_precedence(ActivityPrecedence.Serial(A20, A21, A22))
```

such that a call to `E2` serially executes `A20`, `A21`, and `A22` prior to replying. Here, `A21` is chosen to be an Erlang distribution with given mean (1.0) and number of phases (2).

OrFork with branch probabilities The `OrFork` construct (also available as `Xor`) selects probabilistically between alternative successors of a single activity. The third argument is a row vector of branch probabilities whose entries sum to one and are listed in the same order as the destination activities: Here, after completing `A20` the task executes `A21` with probability 0.3 and `A22` with probability 0.7.

Loop precedence The `Loop` construct repeats a body activity a configurable number of times before continuing with a successor. In its three-argument form `Loop(preAct, {loopBody, endAct}, count)`, the activity `preAct` is executed once, then `loopBody` is executed `count` times, and finally `endAct` runs once before exiting the loop: A four-argument form `Loop(preAct, loopBody, endAct, counts)` is also accepted as a more explicit equivalent.

AndJoin with quorum By default, `AndJoin` blocks until all parallel branches spawned by the matching `AndFork` have completed. The optional third argument `quorum` relaxes this synchronization so that the join fires as soon as any `quorum` of its predecessors have reported completion, modeling early-reply patterns and partial joins: The example releases `A24` as soon as any two of `A21`, `A22`, `A23` have completed.

CacheAccess precedence The `CacheAccess` construct routes the control flow downstream of a cache lookup based on the lookup outcome. The post-cache activity list is interpreted positionally: the first element is the hit branch, the second the miss branch: This precedence is meaningful only when `preAct` is bound to an `ItemEntry` of a `CacheTask`; see Section 6.5.2 for the cache modeling primitives.

Activity configuration

Beyond the host demand specified at construction time, individual activities expose a number of methods that influence how they participate in the activity graph and in the analysis.

Two-phase activities (setPhase). The two-phase model allows part of an activity's execution to be carried out after a synchronous reply has been sent to the caller. Activities are by default in phase 1 (executed before the reply) and can be moved to phase 2 (executed after the reply, while the caller proceeds in parallel) using `A.set_phase(2)`. Only phase values of 1 or 2 are accepted. Phase-2 activities show up in the layered network struct under the `actthink_*` fields.

Per-activity think time (setThinkTime). An activity can carry an additional think-time component, separate from the host demand and from the task-level think time, modeling for example a deferred operation. The method `A.set_think_time(d)` accepts either a numeric mean (which produces an exponential distribution) or any `Distribution` object. A zero or negligible mean is automatically replaced by an `Immediate` distribution.

Asynchronous calls (asynchCall). In addition to `synchCall`, an activity can issue non-blocking calls to entries of other tasks via `A.asynch_call(E, m)`. The destination `E` can be passed either as an `Entry` object or by name; the second argument `m` is the mean number of calls per activity execution and defaults to 1.0. Multiple distinct destinations can be added by repeated calls, but a single destination cannot be registered twice – to model multiple calls to the same entry, increase `m` accordingly.

Call groups (synchCallRoundRobin, synchCallJSQ). An activity's synchronous calls can also be dispatched to a *group* of target entries rather than to one fixed destination, by `A.synch_call_round_robin(dests, m)` or `A.synch_call_jsq(dests, m)`, where `dests` is a list of two or more target entries and `m` is the total mean number of calls per activity execution, defaulting to 1.0. Round robin sends successive calls to the targets in cyclic order, so each receives `m/numel(dests)` of them deterministically; JSQ instead sends every call to whichever target task's station currently holds the fewest jobs, breaking ties uniformly at random. Both differ from an equal-probability `synchCall` split with the same per-target means only in how the individual calls are correlated over time, not in the long-run rate each target receives.

A call group is representable only under the squashed (`'flat'`) layering, since under `'srvn'` the grouped targets never share a submodel; the solve must therefore be asked for that layering, either as `options.method='flat'` or through `options.config.layering`. Honoring the dispatch rule itself further requires a layer solver capable of state-dependent routing, which only `CTMC` and `SSA` provide (`SSA` alone in the C++ port); `SolverLN` rejects a model containing a call group before it reaches an MVA, NC or FLD layer solver, rather than silently returning a probabilistic split mislabeled as round-robin or JSQ. The examples `lqn_rrobin` and `lqn_jsq` illustrate both constructs. A group survives an export to `.lqnx` as

Error Type	Error Type
Activity in REF task replies	Entry called both synchronously and asynchronously
Entry on task calls itself	Repeated definition of parent task
Entry on task calls entry on the same task	Invalid .on() argument for an activity
Cycle in activity graph	Invalid .on() argument for a task
Unsupported reply_to	Repeated synch calls
Activity with bound_to specification	Repeated asynch calls

a `<call-group>` element naming its strategy and its target entries, a LINE extension of the schema: the member calls stay ordinary `<synch-call>` elements carrying the same per-target means, so a reader that ignores the extension — `lqns` and `lqsim`, which have no dispatcher — analyses the ungrouped twin instead of a model with the calls dropped.

Open arrivals at entries (Entry.setArrival). An entry can be turned into an open arrival point by attaching an arrival distribution. The method `E.set_arrival(dist)` accepts any `Distribution` object, e.g., `Exp(2.5)` for Poisson arrivals at rate 2.5 or `Erlang(2,0.4)` for an Erlang-2 stream. Open-arrival entries surface in the layered network struct through the `arrival_*` fields and let the entry serve external workload alongside (or instead of) calls received from a reference task.

Debugging and visualization

The structure of a `LayeredNetwork` object can be graphically visualized as follows

```
model.view()
```

```
model.view()
```

The `jsimg_view` and `jsimw_view` methods can be used to visualize in JMT each layer. This can be done by first calling the `get_layers` method to obtain a list consisting of the Network objects, each one corresponding to a layer, and then invoking the `jsimg_view` and `jsimw_view` methods on the desired layer. This is discussed in more details in the next section.

Lastly, we note a number of specification issues that trigger errors in the LQN definition:

The internal representation of a `LayeredNetwork`, returned by `get_struct`, and the decomposition of a layered model into layers are documented field by field in the LINE internals manual (`LINE-internals-python.pdf`, appendix *The `LayeredNetworkStruct` reference*).

6.5.3 Solvers

LINE offers two solvers for the solution of a `LayeredNetwork` model consisting in its own native solver (`LN`) and a wrapper (`LQNS`) to the LQNS solver [60]. The latter requires a distribution of LQNS to be available on the operating system command line.

LQNS is an external tool developed by the Real-time and Distributed Systems Group at Carleton University. The wrapper writes an XML file conforming to the LQN metamodel, runs the `lqns` (or `lqsim`) executable as a subprocess, and parses the XML output back into LINE metrics. Its solution methods, and the convergence tolerance and iteration cap that govern them, are selected through command-line flags set from the solver options.

The two layered solvers differ in where the decomposition happens. LN converts the layered model into a set of ordinary queueing networks in which cross-layer synchronization appears as artificial delay stations, then iterates a network solver until the delays are mutually consistent; this makes the whole LINE solver portfolio, and features such as sensitivity analysis and optimization, available to layered models. LQNS instead applies layered algorithms directly to the hierarchy without that intermediate conversion, which converges faster on deeply nested models and supports constructs, notably quorum joins, that the conversion does not yet cover. On standard layered models the two agree; divergences are confined to combinations of scheduling disciplines, replication and activity precedences.

The solution methods available for `LayeredNetwork` models are similar to those for Network objects. For example, the `avg_table` can be used to obtain a full set of mean performance indexes for the model, e.g.,

```
avg_table = LQNS(model).getAvgTable()
print(avg_table)
```

Note that in the above table, some performance indexes are marked as `NaN` because they are not defined in a layered queueing network. Further, compared to the `avgTable` method in objects, `LayeredNetwork` do not have an explicit differentiation between stations and classes, since in a layer a task may either act as a server station or a client class.

The main challenge in solving layered queueing networks through analytical methods is that the parameterization of the artificial delays depends on the steady-state performance of the other layers, thus causing a cyclic dependence between input parameters and solutions across the layers. Depending on the solver in use, such issue can be addressed in a different way, but in general a decomposition into layers will remain parametric on a set of response times, throughputs and utilizations.

This issue can be resolved through solvers that, starting from an initial guess, cyclically analyze the layers and update their artificial delays on the basis of the results of these analyses. Both LN and LQNS implement this solution method. Normally, after a number of iterations the model converges to a steady-state solution, where the parameterization of the artificial delays does not change after additional iterations.

LQNS

The LQNS wrapper operates by first transforming the specification into a valid LQNS XML file. Subsequently, LQNS calls the solver and parses the results from disks in order to present them to the user in the appropriate LINE tables or vectors. The `options.method` can be used to configure the LQNS execution as follows:

- `options.method='default'` or `'lqns'`: LQNS analytical solver with default settings.

- `options.method='exactmva'`: the solver will execute the standard LQNS analytical solver with the exact MVA method.
- `options.method='srvn'`: LQNS analytical solver with SRVN layering.
- `options.method='srvn.exactmva'`: the solver will execute the standard LQNS analytical solver with SRVN layering and the exact MVA method.
- `options.method='sim'` or `'lqsim'`: LQSIM simulator, with simulation length specified via the `samples` field (i.e., with parameter `-A options.samples`).
- `options.method='lqnsdefault'`: alias of `default` retained for backward compatibility with earlier scripts.

Upon invocation, the `lqns` or `lqsim` commands will be searched for in the system path, and the solver is unavailable if they are not found there. LINE ships no LQNS binary, names no container image holding one, and never pulls one: LQNS is distributed by its authors under an evaluation agreement that forbids providing the software to third parties, so the binary must be one the user has obtained and is licensed for. Obtain it from http://www.sce.carleton.ca/rads/lqns/. To exercise a containerised LQNS build against the test suite, `run_tests.sh --lqns-docker <image>` places `lqns`, `lqsim` and `qnsolver` shims on the `PATH` that run the named local image; the solvers themselves still see ordinary local binaries. The same applies to `qnsolver`, which is part of the LQNS distribution and backs the `QNS` solver. Local execution is distinct from `options.config.remote`, which dispatches instead to a remote `lqns-rest` service.

The pragma string passed to `lqns` can be customised through the `options.config.multiserver` entry, which forwards a `-Pmultiserver=...` pragma selecting the LQNS multi-server approximation (`'conway'`, `'reiser'`, `'rolia'`, `'zhou'`, `'suri'`, or `'schmidt'`; the `'default'` value maps to `'rolia'`). The value is passed verbatim to the back-end and therefore accepts any multi-server pragma supported by the installed LQNS version.

Remote execution As an alternative to local installation of the LQNS tools, LINE supports remote execution via a Docker container running the `lqns-rest` API. This is useful when LQNS binaries are not available locally or when running on systems where compilation is difficult. To enable remote execution, configure the following options:

- `options.config.remote = True`: Enable remote execution via REST API.
- `options.config.remote_url = 'http://localhost:8080'`: URL of the lqns-rest server (default port is 8080).

The service at that URL is one the user runs; LINE publishes no LQNS image and names none here, for the licensing reason given above. When remote execution is enabled, the model is serialized to LQNX format and sent via HTTP POST to the remote server, which executes the solver and returns the results. This approach supports all solver methods (`lqns`, `lqsim`, etc.) and pragmas available in the local execution mode.

LN

The native LN solver iteratively applies the layer updates until convergence of the steady-state measures; the layering, the parameter updates, and the convergence control are described in the LINE internals manual (LINE-internals-python.pdf, Chapter LN). Since updates are parametric on the solution of each layer, LN can apply any of the solvers described in the solvers chapter to the analysis of individual layers, as illustrated in the following example for the MVA solver

```
avg_table = LN(model, lambda layer: MVA(layer)).getAvgTable()
print(avg_table)
```

Options parameters may also be omitted. The LN method converges when the maximum relative change of mean response times across layers from the last iteration is less than options.iter_tol.

A method name of the LN solver carries *two* decisions, and its two-part form says so: the part before the dot is the **layering**, which fixes what a submodel is, and the part after it is the **encoding**, which fixes how an activity graph is written into that submodel.

Methods supported by the LN solver include:

- options.method='srvn.cs': the SRVN layering with the activity graph encoded as **class switching**. There is one submodel per served element – a host layer for each processor and a task layer for each called task – and inside it the graph becomes routing: one class per task, entry, activity and call, plus the Fork, Join and Router nodes that an AND precedence needs and the class switch that an off-diagonal $P_{r,s}$ mints. This is the encoding of every earlier release and it serves every layered feature the solver implements.
- options.method='srvn.ph': the same layering with the activity graph composed instead into one **phase-type** service law per entry, by the exact series-parallel reduction of [136] (sequence by convolution, OR by mixture, loop by the geometric compound, AND by the serialized host law). A layer is then a two-station cycle – a client delay and the server – carrying one closed class per calling task instead of one per graph element, which makes it markedly smaller and faster. It cannot represent second phases, forwarding calls, cache tasks, quorum AND joins, routed call groups or per-entry admission constraints, and it refuses such a model by name rather than degrading it silently.
- options.method='srvn': the **alias**, and the default. It resolves to srvn.ph where that encoding can serve the model and to srvn.cs where it cannot. The decision is made once, at layer-build time, by a non-destructive feasibility probe (the feature gate, then the composition of every entry workflow), and the outcome is reported in solver.lnmethod. Naming srvn.ph explicitly is therefore *not* the same as taking the default: the alias falls back, the explicit name raises.
- options.method='flat.cs': the **squashed** layering, in which every processor and every called task is a station of one single submodel and a call routes the job to the callee's station rather than to a surrogate delay. It is the layering that lets two servers contend within one network, and the only one under which a routed call group (Section 6.5.2) can be expressed at all, since the group's targets must share a submodel for a dispatch decision to mean anything. Squashing is refused rather than approximated for a replicated

element, a cache task or a setup task, each of which carries state that only a submodel of its own can hold.

- `options.method='flat.ph'`: the squashed layering with the activity graph composed into a **phase-type** service law, that is `flat.cs`'s single submodel carrying `srvn.ph`'s encoding. A caller task is one closed class and visits each server it uses once per invocation, served there by the composed law of the demand it places on that server, so the submodel carries one class per calling task instead of one per entry, activity and call. Squashing does not conflict with the composition: a task station's service time is already the *inflated* entry time — host demand plus call counts times callee service and waiting — which the outer fixed point updates, as in `lqns --squashed-layering`. It refuses what `srvn.ph` refuses, and additionally the replicated elements and setup tasks that any squashing refuses, and the routed call groups that only the routing encoding can dispatch.
- `options.method='flat'`: the alias for `flat.cs`. It resolves unconditionally rather than probing `flat.ph`, because a model is squashed in order to express what only the routing encoding carries.
- `options.method='default'`: a synonym of `srvn`, so a model solved without naming a method takes the better of the two SRVN encodings.
- `options.method='moment3'`: solution by recursive 3-moment approximation of response time distributions. This method matches the first three moments (mean, variance, skewness) of response times at each layer by fitting acyclic phase-type distributions, improving accuracy for non-exponential service demands compared to the default mean-value method. It builds the class-switching layers and adds the distribution propagation on top of them.

The layering may equivalently be selected through `options.config.layering`, whose values are `'srvn'` (the default) and `'flat'` (`'squashed'` is accepted as a synonym). The two channels agree: a method that names a layering sets it, and a layering named in the configuration is reported back through `lnmethod` under its method name, so a model squashed through the configuration reads back as `flat.cs` even when no method named it.

Relaxation controls. The convergence behaviour of the layered iteration is governed by a family of `options.config` entries:

- `options.config.relax` (default `'fixed'`) selects the under-relaxation mode used when blending the new and previous iterates: `'none'` (no relaxation), `'fixed'` (constant factor), `'adaptive'` (factor adjusted from the recent error history), or `'auto'`.
- `options.config.relax_factor` (default 0.5) is the relaxation factor $\omega \in (0, 1]$ applied in the `'fixed'` mode. Smaller values yield more conservative updates and improve convergence on stiff models.
- `options.config.relax_min` (default 0.1) and `options.config.relax_history` (default 5): respectively the lower clamp on the relaxation factor and the error-history window used by the `'adaptive'` mode.

- `options.config.stochiter` (default `'auto'`) selects how the layered fixed point is driven when the layer solvers return noisy estimates, as they do when a layer is analysed by simulation. The value `'off'` keeps the plain Picard iteration, `'rm'` switches to Robbins-Monro stochastic approximation, in which successive iterates are averaged with a decaying step size so that the noise is filtered rather than propagated, `'crn'` additionally couples the layer solvers with common random numbers, and `'auto'` selects between them according to whether the model is classified as stochastic. The schedule is controlled by `options.config.stochiter_alpha` (default 0.6), the step-decay exponent, which must lie in $(0.5, 1]$ for the averaged iterate to converge, `options.config.stochiter_a0` (default 1.0), the initial step after burn-in, `options.config.stochiter_burnin` (default 5), the number of Picard iterations executed before the step starts decaying, and `options.config.stochiter_conseq` (default 3), the number of consecutive sub-tolerance iterations required before the iteration is declared converged. The last of these matters because a single sub-tolerance iterate is not evidence of convergence when the residual is itself a random variable.
- `options.config.layer_init` (default none) selects the initialization of the layer throughput state. Setting it to `'bound'` seeds each element's initial throughput with the geometric mean $\sqrt{X^- X^+}$ of the Majumdar-Woodside robust box bounds (algorithm `lqn_boxbounds`). The initialization is correctness-preserving; for closed-chain layered models the outer iteration re-derives the throughputs from the first solve, so the converged solution is unchanged.
- `options.config.interlocking` (default `true`) applies the LQNS-style interlock correction to the residence time a layer attributes to a call, accounting for the traffic two tasks impute to each other along a shared downstream path ([59], following the static-analysis construction of LQNS V5). It is applied only when the layer's underlying MVA dispatch reaches an algorithm able to carry the correction matrix; a layer routed elsewhere (for instance to `qna`, `rqna`, `rqt` or the Smith Queue Decomposition path) computes without it, silently rather than by falling back to a different algorithm to force the correction in, because the algorithm swap itself was found to induce non-convergence on some models. See the internals manual, Chapter `LN`, for the mechanism and the reason it is a gate rather than a substitution.

Layer-wise sensitivities. `SolverLN` also reports the derivatives of the layer measures with respect to the service rates, by delegating to the layer solvers and concatenating their tables under a leading `Layer` column; see Section 10.9.1 for branch selection and options. In every edition, these are partial derivatives within a layer rather than total derivatives of the layered model.

Layered box bounds. As a fast, non-iterative alternative to the full decomposition, `SolverLN` exposes the throughput bound methods `options.method='mwba.upper'` and `'mwba.lower'` (algorithm `lqn_boxbounds`). These build a closed multiclass queueing network whose stations are the processors and whose classes are the reference-task call chains, and apply the Majumdar-Woodside robust box bounds [103] to return per-task upper and lower throughput bounds. This generalizes the classical LQN Type-1 throughput bound $X \leq N/(Z + D)$ (computed, e.g., by `lqns -b`) with a utilization-based upper bound and a contention-aware lower bound.

Transient analysis. `SolverLN` supports transient analysis of a layered model through `get_tran_avg`, which returns the time-varying mean queue lengths, utilizations and throughputs of every layer in the same block-diagonal layout (station by class, per layer) used by the steady-state results. Two schemes are available. The decoupled scheme freezes the inter-layer demands at the converged fixed point and then integrates each layer's transient independently, which is inexpensive but ignores the fact that a layer's demand on another layer itself varies over the transient. The coupled scheme reconciles the layers by waveform relaxation [94]: each layer's fluid transient is driven by the time-varying demand trajectories taken from the other layers' most recent transients, and the sweep is repeated until the trajectories stop changing in the sup-norm over time. The coupled channels are the task think times, which enter as a client delay, and the synchronous-call service demands, which enter at the caller's client station; intra-layer host service is held at its equilibrium value. The time-varying demands are injected into each layer's closing ODE through a per-event rate multiplier. The first coupled sweep uses the frozen equilibrium demands and therefore reproduces the decoupled result exactly, and at convergence every layer relaxes to its own fixed point, so the endpoint of the transient agrees with `getAvg`.

State transfer and solver switching The `LN` solver supports transferring its internal convergence state between solver instances, enabling hybrid solving schemes where a fast solver provides initial estimates that a more accurate solver then refines. This avoids restarting the iterative decomposition from scratch when switching solvers.

Three methods are provided:

- `get_state()`: exports the current solver state, including service time processes, throughputs, utilizations, and relaxation parameters.
- `set_state(state)`: imports a previously exported state into a new solver instance, allowing it to continue from the previous solution.
- `update_solver(solver_factory)`: replaces the per-layer solver while preserving the current convergence state.

A typical use case is to obtain a quick initial solution with `MVA` and then refine it with a simulation-based solver:

```
# Fast initial analysis with MVA
solver1 = LN(model, lambda m: MVA(m))
T_fast = solver1.getAvgTable()
state = solver1.get_state()

# Refine with NC solver
solver2 = LN(model, lambda m: NC(m))
solver2.set_state(state)
T_refined = solver2.getAvgTable()
```

6.5.4 Conversion between Network and LayeredNetwork models

LINE provides two converters that bridge the `Network` (queueing network) and `LayeredNetwork` (layered queueing network) abstractions. These converters are useful when one wishes to apply a layered solver to an ordinary queueing network, or vice versa to analyse a layered model with a flat queueing network solver.

From Network to LayeredNetwork

The function `QN2LQN` maps a closed queueing network into a layered queueing network. Each closed chain is modelled by a reference task whose entries replicate the chain's visit pattern, and each queueing or delay station is mapped onto a processor hosting the corresponding task. Routing probabilities and class switching are reproduced through OR-fork activity precedences. Open classes (`Source`/`Sink`) are not currently supported by this converter and the input model must be closed. The conversion is invoked as

```
from line_solver import LQNS
from line_solver.api.io import qn2lqn
lqnmodel = qn2lqn(model)
avg_table = LQNS(lqnmodel).getAvgTable()
```

The `QN2LQN` converter is invoked automatically by the `QNS` solver in MATLAB when the input model lacks a product-form solution and contains only closed classes; in this case the model is converted on the fly and dispatched to `LQNS`.

From LayeredNetwork to Network

The reverse converter `LQN2QN` maps a `LayeredNetwork` into an extended queueing network in which synchronous calls are realised through reply signal classes (`SignalType.REPLY`). The caller class is blocked at the calling station until the reply class returns from the callee, and a class switch at the leaf node restores the caller class so that the request cycle can repeat. Phase-2 activities, when present, are mapped onto fork-join constructs so that the post-reply work is executed in parallel with the released caller. The conversion is invoked as Beyond synchronous calls, the converter maps the remaining call and task constructs of the layered model. Asynchronous calls become ordinary routing without a reply signal, since the caller does not block. Forwarding calls are translated into the caller-side pseudo rendezvous described in the forwarding section, so that the original client stays blocked until the last entry of the forwarding chain replies. Fork calls and their joins are mapped onto the corresponding `Fork` and `Join` nodes. Cache tasks are mapped onto a `Cache` node with the item entries as read classes and the hit and miss branches as the corresponding outgoing classes. The multiplicity of a task is enforced as a thread pool rather than as a plain multi-server station: a task with multiplicity m admits at most m concurrent requests across all of its entries, which is expressed as a finite capacity region over the task's stations, so that a task whose entries have different demands cannot exceed its declared concurrency. Open arrivals declared on an entry become a `Source` and `Sink` pair carrying an open chain into that entry, rather than a closed chain with a large population, so that the arrival rate is honoured exactly. The converter is available in all three codebases.

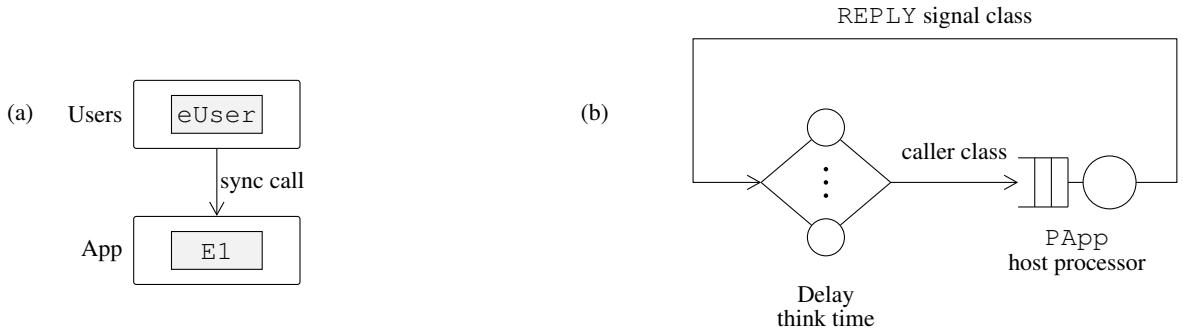


Figure 6.7: The output of LQN2QN. (a) A reference task issuing one synchronous call to an entry of a server task. (b) The flattened queueing network: the host processor of each task becomes a station, the reference task becomes a Delay holding its think time, and each step of the activity graph becomes a closed class. The caller is blocked while the callee runs, and is released by the REPLY signal class emitted by the replying activity.

Figure 6.7 shows the resulting network for a single synchronous call.

This route is useful for validating an LN or LQNS solution against discrete-event simulation, since the resulting network can be analysed with any solver that supports signal classes (currently LDES and JMT).

6.5.5 Model import and export

A LayeredNetwork can be easily read from, or written to, a XML file based on the LQNS meta-model format[1]. The read operation can be done using a static method of the LayeredNetwork class, i.e.,

```
model = LayeredNetwork.parse_xml(filename)
```

Conversely, the write operation is invoked directly on the model object

```
model.write_xml(filename)
```

In both examples, filename is a string including both file name and its path.

Finally, we point out that it is possible to export a LQN to an XML file in the LQNX format by means of the write_xml(filename) function.

Ensemble merging

The Ensemble.merge function combines multiple independent models into a single Network containing disconnected subnetworks. Node and class names are prefixed with their model name to prevent conflicts, while all Source and Sink nodes are merged into single MergedSource and MergedSink nodes.

[1] https://raw.githubusercontent.com/layeredqueuing/V5/master/xml/lqn.xsd

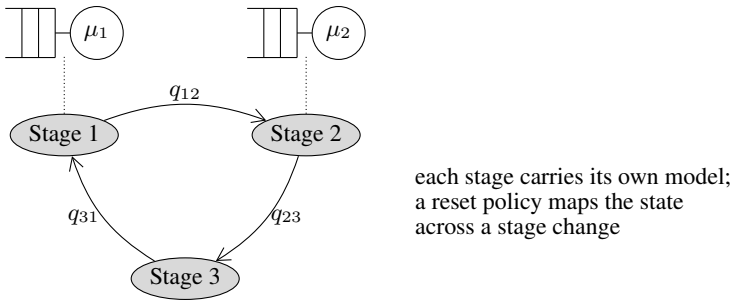


Figure 6.8: A random environment with three stages. The stage process is a continuous-time Markov chain or a semi-Markov process; each stage carries its own model, and a reset policy maps the state of the outgoing stage onto the incoming one.

```
merged_model = Ensemble.merge([model1, model2])
```

6.6 Random environments

Random environments provide a framework for modeling systems whose input parameters, such as service rates, scheduling weights, or number of stations, vary according to the state of an external environment. We refer to these environmental states as *stages*. For instance, a system that alternates between normal-load and peak-load conditions can be represented by a two-stage environment, with each stage determining the number of servers available at a queueing station. Systems of this type arise in many settings, including servers subject to breakdown and repair, auto-scaling applications, and reliability models.

In LINE, an environment is described as a continuous-time Markov chain (CTMC) or as a semi-Markov process (SMP). Compared to CTMCs, SMPs relax the restriction of having exponential transitions between stages, but they are computationally harder to evaluate. As a first approximation, one may solve independent models for each stage and weight their solutions. However, depending on the frequency of transitions between stages, this can incur significant errors [35]. LINE automates the evaluation of such scenarios, applying the appropriate corrections based on the environment and system characteristics.

6.6.1 Environment definition

Figure 6.8 shows the structure of such a model.

Specifying the environment

To specify an environment, we first create an `Environment` object

```
E = 2
env_model = Environment('UnreliableEnv', E)
```

where the second parameter indicates the number of stages in the environment. We then add two stages

```
env_model.add_stage(0, 'Online', 'UP', network1)
env_model.add_stage(1, 'Offline', 'DOWN', network2)
```

where the constructor specifies the stage name, an arbitrary string to classify the stage semantics, followed by the system model used while that environment stage is active.

We now describe that the transitions between stages are both exponential, with different rates

```
env_model.add_transition(0, 1, Exp(1))
env_model.add_transition(1, 0, Exp(2))
```

Self-loops are allowed. When multiple transitions are possible, the timer that elapses first determines the next stage.

The `get_stage_table` method summarizes the properties of an environment:

```
env_model.getStageTable()
  Stage    Name Type    Model
      0  Online   UP network1
      1 Offline DOWN network2
```

In the table, the `Stage` column gives a numerical identifier for each stage, followed by its name, type classification, and a pointer to the sub-model associated with that stage.

Specifying system models for each stage

LINE places loose assumptions in the way the system should be described in each stage. It is just expected that the user supplies a model object, either a or a `LayeredNetwork`, in each stage, and that a transient analysis method is available in the chosen solver, a requirement fulfilled for example by `FLD`. When the stage models are `LayeredNetwork` objects, each stage is analysed by `LN` and the blending iteration is carried out over the layered ensembles, so that a layered application whose demands or multiplicities differ between environment stages can be studied without flattening it first. A worked two-stage layered example is provided in `renv_lqn_twostages` in the random-environment examples folder.

Specifying a reset policy

A reset policy specifies whether an environment transition instantaneously brings a redistribution of the jobs across the network. When the environment transitions, by default jobs in execution at a server are required all to restart execution at that server upon occurrence of a transition. This may not be appropriate in some models, for example when a station is removed from the model and its jobs can therefore no longer execute at that station. For such cases, one may define a custom reset policy, e.g.,

```
import numpy as np

# Define a reset function that moves all jobs into station 0
def reset_rule(q_exit):
    num_stations, num_classes = q_exit.shape
    q_reset = np.zeros((num_stations, num_classes))

    # Move all jobs to station 0, preserving their classes
    for c in range(num_classes):
        q_reset[0, c] = q_exit[:, c].sum()
    return q_reset

# Add transition with reset rule
# Assuming stages are indexed (0 for 'Online', 1 for 'Offline')
env_model.add_transition(0, 1, Exp(1.0), reset_rule)
```

In the above code, `q_exit[i, r]` is the queue-length of class-`r` jobs observed at station `i` upon exiting the online state. The `reset_rule` is a function that takes a 2D numpy array and must return an array of the same shape. The reset policy in this example moves instantaneously all jobs in the network into station 0 upon entering into the offline state. Note that `reset_rule` can be configured differently for each stage transition and the default value is simply the identity function `lambda q: q`.

State-dependent environment rates In addition to the queue-length reset rule, the `add_transition` method accepts an optional sixth argument, a *reset rule for the outgoing environment-transition distribution*. This produces a state-dependent semi-Markov environment, in which the timing of stage transitions adapts to the system state at the moment of the previous stage change. The signature is

```
env_model.add_transition(from_, to, distrib, reset_fun, reset_env_rates_fun)
```

where `reset_env_rates_fun` is a callable `lambda original_dist, q_exit, u_exit, t_exit: new_dist` that returns the updated distribution. The default `lambda d, q, u, t: d` preserves the original distribution. When at least one transition supplies a non-trivial `resetEnvRatesFun`, `ENV` automatically switches to the state-dependent solution mode (see `options.method='statedep'` in the `ENV` configuration table below); this mode is incompatible with the semi-Markov mode `options.method='smp'`.

6.6.2 Solvers

The steady-state analysis of a system in a random environment is carried out in LINE using the blending method [35], which is an iterative algorithm leveraging the transient solution of the model. In essence, the model looks at the *average* state of the system at the instant of each stage transition, and upon restarting the system in the new stage re-initializes it from this average value. This algorithm is implemented in LINE by the `ENV` class, which is described next.

ENV

The ENV class (alias: SolverENV) applies the blending algorithm by iteratively carrying out a transient analysis of each system model in each environment stage, and probabilistically weighting the solution to extract the steady-state behavior of the system.

The solver offers two analyzer modes, selected through options.method. The default 'meanfield' mode couples stages through the marginal mean queue length (the classic blending method). The alternative 'statevec' mode carries the full per-stage state-probability vector across stage transitions instead of only its mean, yielding higher accuracy at the cost of an explicit state space; both are described in the LINE internals manual (LINE-internals-python.pdf, Chapter ENV). The 'statevec' analyzer supports a CTMC backend for closed models and a MAM/LDQBD backend for open and load-dependent queues, and it accepts the additional option options.sojourn='deterministic', which models each stage sojourn as lasting exactly its mean (exact via uniformization) rather than as an exponential holding time.

As in the transient analysis of objects, LINE does not supply a method to obtain mean response times, since Little's law does not hold in the transient regime. To obtain the mean queue-length, utilization and throughput of the system one can call as usual the avg method on the ENV object, e.g.,

```
# Configure solvers for each environment
solvers = []
for e in range(env_model.num_stages):
    solvers.append(FLD(env_model.get_model(e)))
    solvers[e].options = SolverOptions(SolverType.FLUID)

env_solver = ENV(env_model, solvers, options)
env_solver.avg()
result = env_solver.result
```

Note that as model complexity grows, the number of iterations required by the blending algorithm to converge may grow large. In such cases, the options.iter_max option may be used to bound the maximum analysis time.

Per-stage results While the avg family of methods returns environment-averaged metrics, the underlying per-stage results computed by the blending algorithm remain accessible through the ENV solver. The full per-stage solver result for stage e is available at env_solver.get_stage_result(e), with the corresponding stage-local child solver retrievable via env_solver.solvers[e]. The steady-state probability of being in each environment stage is exposed as env_obj.prob_env; the same values appear as the Prob column of the table returned by get_stage_table. The environment-averaged metrics returned by getAvg are the weighted combination $\sum_e \pi_e X_e$ where X_e is the per-stage queue-length, utilization or throughput and π_e is the corresponding entry of probEnv.

Stage holding-time inspection The semi-Markov holding time in each stage is precomputed when the environment is initialized. Stage e's holding-time MMAP representation is stored as env_obj.hold_time[e]

and is also reported in the `HoldT` column of `get_stage_table`, with the convenience alias `getStageT` producing the same table. When using the semi-Markov method (`options.method='smp'`), the per-stage sojourn-time CDF is exposed by the `computeSojournCdf(e, t)` method of `ENV`, which evaluates $\Pr\{S_e \leq t\}$ for the chosen stage `e`.

Decomposition/Aggregation Methods When the environment has many states, the `ENV` solver can use decomposition/aggregation methods to reduce computational complexity. The decomposition method can be configured using `options.config.da`. The available methods are based on nearly completely decomposable (NCD) Markov chain analysis:

- *Courtois decomposition* (`'courtois'`): The default method, based on [47]. It partitions the environment state space into macro-states and computes approximate steady-state probabilities.
- *Koury-McAllister-Stewart* (`'kms'`): An iterative aggregation-disaggregation method [88] that refines the Courtois solution through successive approximations.
- *Takahashi's method* (`'takahashi'`): Another iterative aggregation-disaggregation approach [149] with different convergence properties than KMS.
- *Multigrid* (`'multi'`): A multi-level decomposition method [146] that applies hierarchical aggregation.

For iterative methods (`'kms'` and `'takahashi'`), the number of iterations can be controlled using `options.config.da_iter` (default: 10).

The configuration options are summarized in Table 6.6.2.

Table 6.4: `ENV` configuration options (Python)

Option	Value	Description
`options.config.da`	`'courtois'`	Courtois decomposition (default).
`options.config.da`	`'kms'`	Koury-McAllister-Stewart iterative method.
`options.config.da`	`'takahashi'`	Takahashi's iterative method.
`options.config.da`	`'multi'`	Multigrid decomposition method.
`options.config.da_iter`	`int`	Number of iterations for iterative methods (default: 10).
`options.config.env_alpha`	`float`	Alpha parameter for macro-state search (default: 0.01).
`options.method`	`'meanfield'`	Default mean-field blending method, coupling stages through the marginal mean queue length. Alias of `'default'`.
`options.method`	`'statevec'`	State-vector analyzer propagating the full per-stage state-probability distribution across stage transitions (CTMC backend for closed models, MAM/LDQBD for open/load-dependent queues).
`options.sojourn`	`'deterministic'`	With `'statevec'`, models each stage sojourn as lasting exactly its mean (exact via uniformization) instead of as an exponential holding time.
`options.method`	`'smp'`	Semi-Markov mode using DTMC-based sojourn-time integration; required for non-exponential stage transitions. Equivalent to `set_smp_method(True)`.
`options.method`	`'statedep'`	State-dependent mode used when at least one transition supplies a non-trivial `reset_env_rates_fun`; auto-enabled and equivalent to `set_state_dep_method('statedep')`. Incompatible with `'smp'`.
`set_compression(flag)`	`bool`	Enable Courtois decomposition (default: `False`).

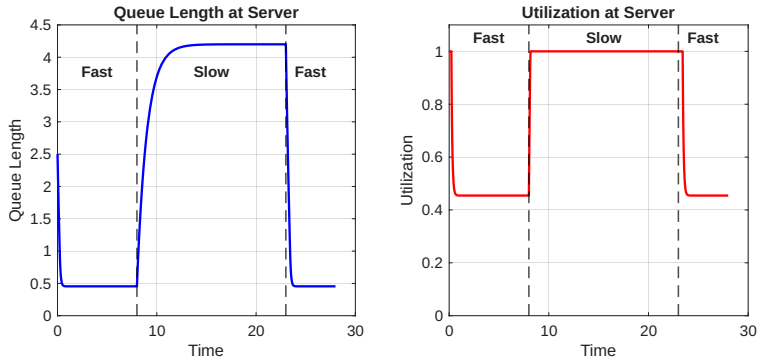


Figure 6.9: Transient queue length and utilization evolution along a sample path.

6.6.3 Examples

Example 1: Fast and slow service

This example demonstrates how to model a queueing system operating in a random environment, where system parameters (e.g., service rates) change according to an underlying environmental process.

The scenario models a server that alternates between "Fast" and "Slow" modes. In Fast mode, the service rate is 10.0 (low utilization). In Slow mode, the service rate is 0.8 (high utilization, near saturation). The environment switches from Fast→Slow at rate 0.5 (mean time in Fast mode = 2.0) and from Slow→Fast at rate 1.0 (mean time in Slow mode = 1.0).

We start by creating a base closed queueing network with a delay and a queue:

```
base_model = Network('BaseModel')
delay = Delay(base_model, 'ThinkTime')
queue = Queue(base_model, 'Fast/Slow Server', SchedStrategy.FCFS)

# Closed class with 5 jobs
N = 5
jobclass = ClosedClass(base_model, 'Jobs', N, delay)
delay.setService(jobclass, Exp(1.0))  # Think time = 1.0
queue.setService(jobclass, Exp(2.0))  # Placeholder

# Connect nodes in a cycle
base_model.link(Network.serialRouting(delay, queue))
```

Next, we create the random environment with two stages, each having a different service rate:

```
env = Environment('ServerModes', 2)

# Stage 0: Fast mode (service rate = 10.0, low utilization)
fast_model = base_model.copy()
```

```
fast_queue = fast_model.getNodeByName('Fast/Slow Server')
fast_queue.setService(fast_model.classes[0], Exp(10.0))
env.addStage(0, 'Fast', 'operational', fast_model)

# Stage 1: Slow mode (service rate = 0.8, high utilization)
slow_model = base_model.copy()
slow_queue = slow_model.getNodeByName('Fast/Slow Server')
slow_queue.setService(slow_model.classes[0], Exp(0.8))
env.addStage(1, 'Slow', 'degraded', slow_model)

# Define transitions between stages
env.addTransition(0, 1, Exp(0.5))  # Fast->Slow
env.addTransition(1, 0, Exp(1.0))  # Slow->Fast
```

Finally, we solve the model using `ENV` with the Fluid solver (`FLD`) as the sub-solver for each stage:

```
# Initialize and inspect the environment
env_table = env.getStageTable()
print(env_table)

# Solve using ENV with Fluid sub-solver
env_solver = ENV(env, lambda m: FLD(m))
Q, U, T = env_solver.getAvg()

# Display environment-averaged results
env_avg_table = env_solver.getAvgTable()
print(env_avg_table)
```

The results show that the server spends approximately 66.67% of time in Fast mode and 33.33% in Slow mode. The environment-averaged queue length at the server is approximately 2.0 jobs, which is a weighted combination of the queue lengths in the Fast and Slow stages.

Analyzing sample paths with getSamplePathTable In addition to computing steady-state metrics averaged over the environment distribution, `ENV` provides a method to analyze specific sample paths through the environment states. This is useful when you want to understand the transient behavior of the system as it transitions through a known sequence of environment states.

The `getSamplePathTable` method accepts a specification of a sample path—a list of (stage, duration) pairs—and computes the transient performance metrics for each segment. For each segment, the method:

1. Initializes the model from the previous segment's final queue lengths (or uniform distribution for the first segment)

2. Runs transient analysis for the specified duration using the Fluid solver

3. Extracts the initial and final values of queue length, utilization, and throughput

The output is a table with columns for segment index, stage name, duration, station, job class, and the initial/final values of each metric. An example of the transient behavior along a sample path is shown in Figure 6.9, which depicts a sample path transitioning through environment states Fast → Slow → Fast. In the Fast stage (service rate = 10.0), the server operates at low utilization (≈45%), while in the Slow stage (service rate = 0.8), the server becomes saturated (≈100% utilization), causing queue buildup as arrivals accumulate faster than they can be processed.

```
# Define a sample path: Fast for 8.0, Slow for 15.0, Fast for 5.0
sample_path = [('Fast', 8.0), ('Slow', 15.0), ('Fast', 5.0)]

# Compute transient metrics along the sample path
df = solver.getSamplePathTable(sample_path)

# Display the results DataFrame
print(df)
```

The sample path can be specified using either stage names (strings) or stage indices (integers). Stages are 1-indexed in MATLAB and 0-indexed in Java/Python. The duration for each segment must be positive.

Example 2: Breakdown and repair

LINE provides convenience methods to simplify the common case of modeling node failures and repairs in random environments. These methods automatically create UP and DOWN stages for nodes that can break down, configure service rates appropriately, and set up breakdown and repair transitions.

The primary method is `addNodeFailureRepair`, which creates both breakdown and repair transitions in a single call:

```
env = Environment('ServerEnv', 2)
env.add_node_failure_repair(base_model, 'Server1',
    Exp(0.1), Exp(1.0), Exp(0.5))
env.init()
```

where `baseModel` is the network with normal (UP) service rates, `'Server1'` is the node name, `Exp(0.1)` is the breakdown distribution (mean time to failure = 10 time units), `Exp(1.0)` is the repair distribution (mean time to repair = 1 time unit), and `Exp(0.5)` is the service distribution when the node is down.

This method automatically:

- Creates an UP stage with the service rates of the base model
- Creates a DOWN stage with the specified reduced service rate for the failed node
- Adds a breakdown transition (UP → DOWN) with the given breakdown distribution
- Adds a repair transition (DOWN → UP) with the given repair distribution

Alternative methods for more control include `add_node_breakdown` and `add_node_repair`, which can be called separately. Their full signatures are

```
env.add_node_breakdown(base_model, node_or_name, breakdown_dist, ...
down_service_dist, reset_fun)
env.add_node_repair(node_or_name, repair_dist, reset_fun)
```

where `nodeOrName` is either a `Node` object or a string node name, `breakdownDist` and `repairDist` are Markovian distributions controlling the timing of the corresponding stage transitions, and `downServiceDist` is the service distribution applied to all classes at the failed node while it is in the DOWN stage. The first call to `addNodeBreakdown` also creates the UP stage from `baseModel` (if it does not yet exist), and each invocation creates a DOWN stage with auto-generated name `DOWN_<nodeName>`, so that several nodes can break down independently within the same environment. The optional `resetFun` argument follows the same convention as in the previous section and defaults to the identity `lambda q: q`, leaving queue lengths unchanged on the corresponding transition. Custom reset policies can also be supplied at the convenience entry point `add_node_failure_repair` as optional parameters:

```
reset_breakdown = lambda q: q * 0  # Clear queues on breakdown
reset_repair = lambda q: q         # Keep jobs on repair
env.add_node_failure_repair(base_model, 'Server1', Exp(0.1),
    Exp(1.0), Exp(0.5), reset_breakdown, reset_repair)
```

Reset policies can also be modified after environment creation using `setBreakdownResetPolicy` and `setRepairResetPolicy` methods.

Retrieving reliability metrics After solving an environment model with node breakdown and repair, reliability metrics can be retrieved using the `get_reliability_table` method (aliases: `relT`, `getRelT`, `relTable`, `getRelTable`):

```
metrics = env.get_reliability_table()
print(f"MTTF: {metrics['MTTF']}")
print(f"MTTR: {metrics['MTTR']}")
print(f"MTBF: {metrics['MTBF']}")
print(f"Availability: {metrics['Availability']}")
```

The reliability metrics returned are:

- *MTTF* (Mean Time To Failure): Expected time until breakdown from the UP state. For multiple failure modes, this uses the combined failure rate: $\lambda_{\text{total}} = \sum_i \lambda_i$.
- *MTTR* (Mean Time To Repair): Expected time to restore from the DOWN state. For multiple DOWN states, this is a weighted average based on steady-state probabilities.
- *MTBF* (Mean Time Between Failures): Sum of MTTF and MTTR.
- *Availability*: Steady-state probability of the system being in the UP state.

Example 3: Markov-modulated service

This example shows how to convert a queueing network with MMPP2 (Markov-Modulated Poisson Process) service into an equivalent random environment model using the `MAPQN2RENV` utility. MMPP2 is a 2-phase service process where the service rate switches between two values according to an underlying Markov chain. The `MAPQN2RENV` function automatically transforms such a model into a random environment with two stages, each having exponential service at the corresponding MMPP rate.

We create a closed queueing network with an MMPP2 service process:

```
from line_solver import Network, Queue, ClosedClass, SchedStrategy, Exp
from line_solver.distributions.markovian import MMPP2
from line_solver.api.io.converters import mapqn2renv

model = Network('ClosedQN')
queue1 = Queue(model, 'Queue1', SchedStrategy.FCFS)
queue2 = Queue(model, 'Queue2', SchedStrategy.FCFS)

# Closed class with 3 jobs
N = 3
job_class = ClosedClass(model, 'Class1', N, queue1)

# MMPP2 service: slow_rate=2.0, fast_rate=5.0, slow_to_fast=0.5, fast_to_slow=0.3
queue1.set_service(job_class, MMPP2(2.0, 5.0, 0.5, 0.3))
queue2.set_service(job_class, Exp(3.0))

model.link(model.serial_routing(queue1, queue2))

# Convert to random environment
env = mapqn2renv(model)
env.print_stage_table()
```

The output shows the resulting environment structure:

```
Stage Table:
============
Stage 1: Phase0 (Type: item)
  - Network: ClosedQN_Phase0
  - Nodes: 2
  - Classes: 1
Stage 2: Phase1 (Type: item)
  - Network: ClosedQN_Phase1
  - Nodes: 2
  - Classes: 1
Transitions:
  Phase0 -> Phase1: rate = 0.5000
  Phase1 -> Phase0: rate = 0.3000
```

The resulting environment can then be analyzed using `ENV` as shown in Example 1.

Example 4: semi-Markov process environments

This example demonstrates the analysis of random environments with non-Markovian transition distributions using semi-Markov Processes. When environment transitions follow general distributions (e.g., Lognormal), the ENV solver should be run with the `'smp'` method.

Consider a system that alternates between two operational modes with log-normally distributed holding times in each stage.

```
from line_solver import Network, Queue, Source, Sink, OpenClass
from line_solver import SchedStrategy, Exp, Lognormal, Environment, ENV, FLD
from line_solver import SolverOptions

# Create two network models for different modes
model1 = Network('Mode1')
queue1 = Queue(model1, 'Queue1', SchedStrategy.PS)
job_class1 = OpenClass(model1, 'Jobs')
source1 = Source(model1, 'Source')
sink1 = Sink(model1, 'Sink')
source1.set_arrival(job_class1, Exp(1.0))
queue1.set_service(job_class1, Exp(2.0))
model1.link(model1.serial_routing(source1, queue1, sink1))

model2 = Network('Mode2')
queue2 = Queue(model2, 'Queue1', SchedStrategy.PS)
job_class2 = OpenClass(model2, 'Jobs')
source2 = Source(model2, 'Source')
sink2 = Sink(model2, 'Sink')
source2.set_arrival(job_class2, Exp(1.0))
queue2.set_service(job_class2, Exp(0.8))
model2.link(model2.serial_routing(source2, queue2, sink2))

# Create environment with LogNormal transitions (non-Markovian)
env_model = Environment('SemiMarkovEnv', 2)
env_model.add_stage(0, 'Stage1', 'UP', model1)
env_model.add_stage(1, 'Stage2', 'UP', model2)
env_model.add_transition(0, 1, Lognormal(1.0, 0.5))  # Non-exponential
env_model.add_transition(1, 0, Lognormal(0.5, 0.3))  # Non-exponential

# Enable SMP method for non-Markovian transitions
options = SolverOptions()
options.method = 'smp'
solver = ENV(env_model, lambda m: FLD(m), options=options)

# Run each solver
for s in solver._solvers:
    if s is not None:
        s.runAnalyzer()
```

```
# Get average results
QN, UN, TN = solver.avg()
print('Average Queue Lengths:', QN)
print('Average Utilizations:', UN)
print('Average Throughputs:', TN)
```

The ENV solver correctly handles this by using numerical integration to compute transition probabilities from the non-Markovian distribution functions, rather than assuming Markovian (memoryless) transitions.

Example 5: MMAP-modulated cache workload

A random environment can also drive a multiclass workload generated by a marked Markovian arrival process (MMAP). We feed a small Round-Robin cache with a marked MMPP2 whose two marks are bound, via `setMarkedArrival`, to two open read classes that share the modulating chain: phase 1 is bursty and emits mostly class-1 references at a high rate, phase 2 is calm and emits mostly class-2 references at a low rate. Viewing the modulating phase as the environment turns the MMAP into two phase-conditional Poisson streams (the diagonal D_1 blocks), so the cache can be solved by the ENV meta-solver. We use the state-dependent method (`options.method='blend'`), which carries the cache-state distribution across phase switches and averages each phase's sojourn-weighted distribution; for a Markovian environment it recovers the exact joint (cache $\times$ phase) solution, i.e., it matches CTMC on the true MMAP-fed cache. The per-read-class hit ratios returned by the state-dependent ENV solver (`hitBLEND`) match those of the exact CTMC on the true MMAP-fed cache (`hitCTMC`). The faster `'avg'` (fast-environment, rate-averaged) and `'dec'` (slow-environment, quasi-stationary decomposition) methods discard the within-phase temporal correlation of references and are therefore only approximate. A complete runnable version, including the LDES simulation and the two approximate methods, is in `examples/basic/cacheModel/cache_mmap_rr_env.m`.

Mean-field blending. When the per-phase cache state space is too large for the exact `'blend'` inner solver, the default (mean-field) ENV analyzer provides a fully fluid alternative driven by an FLD inner solver. Its refined mean-field (RMF) algorithm analyses each phase's cache by integrating the mean-field occupancy drift over the phase sojourn; the mean cache occupancy is then carried across phase switches (the cache analog of the queue-length hand-off performed by `initFromMarginal`), and the per-read-class hit ratio is the `probEnv`-weighted, sojourn-averaged product of arrival rate and hit probability. This trades the exact joint (cache $\times$ phase) distribution of `'blend'` for a cheaper fluid approximation, and is available identically in the MATLAB, Java, and Python codebases. Figure 6.10 shows the mean-field blending iteration converging, from an initially empty cache, to the fluid equilibrium `hitMF`; the residual gap to the exact CTMC blend is the mean-field error of the small (n=4, m=2) cache.

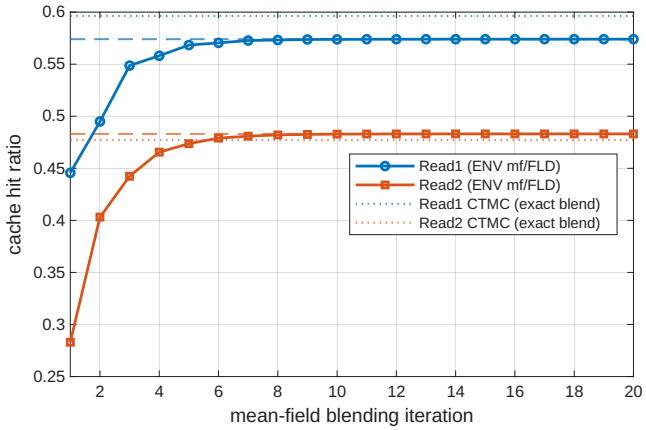


Figure 6.10: Mean-field ENV blending for the MMAP-modulated cache of Example 6.6.3. Starting from an empty cache, the per-read-class hit ratios produced by the default (mean-field) ENV analyzer with an FLD inner solver converge over the blending iteration to the environment-blended equilibrium (dashed), obtained by carrying each phase's mean cache occupancy across phase switches. The exact CTMC blend (dotted) is approached up to the mean-field error of the small (n=4, m=2) cache.

Chapter 7

Advanced analysis methods

The analyses collected here go beyond the steady-state averages of Chapter 4: they name a state of the model, follow it in time, or observe a single realization of it. They are requested through the same handles, but not every solver implements them; the feature tables of Section 1.2 state which one does.

7.1 Specifying states

In some analyses it is important to specify the state of the network, for example to assign the initial position of the jobs in a transient analysis. We thus discuss the support in LINE for state modeling.

7.1.1 Station states

We begin by explaining how to specify a state s_0 for a station. For example, it is not supported for shortest job first (`SchedStrategy.SJF`) scheduling, in which state must include the service time samples for the jobs and it is therefore a continuous quantity.

Suppose that the network has R classes and that service distributions are phase-type, i.e., that they inherit from `Markovian`. Let K_r be the number of phases for the service distribution in class r at a given station. Then, we define three types of state variables:

- c_j: class of the job waiting in position $j \leq b$ of the buffer, out of the b currently occupied positions. If $b = 0$, then the state vector is indicated with a single empty element $c_1 = 0$.
- k_j: service phase of the job waiting in position $j \leq b$ of the buffer, out of the b currently occupied positions.
- n_r: total number of jobs of class r in the station
- b_r: total number of jobs of class r in the station's buffer
- s_{rk}: total number of jobs of class r running in phase k in the server

Table 7.1: State descriptors for Markovian scheduling policies

Sched. strategy	Station state vector	State condition
EXT	$[\texttt{Inf}, s_{11}, ..., s_{1K_1}, ..., s_{R1}, ..., s_{RK_R}]$	$\sum_k s_{rk} = 1, \forall r$
FCFS, FCFSPRIO, LCFS	$[c_b, ..., c_1, s_{11}, ..., s_{1K_1}, ..., s_{R1}, ..., s_{RK_R}]$	$\sum_r \sum_k s_{rk} = S$
LCFSPR, LCFSPI	$[c_b, k_b, ..., c_1, k_1, s_{11}, ..., s_{1K_1}, ..., s_{R1}, ..., s_{RK_R}]$	$\sum_r \sum_k s_{rk} = S$
SEPT, SIRO	$[b_1, ..., b_R, s_{11}, ..., s_{1K_1}, ..., s_{R1}, ..., s_{RK_R}]$	$\sum_r \sum_k s_{rk} = S$
PS, DPS, GPS, INF	$[s_{11}, ..., s_{1K_1}, ..., s_{R1}, ..., s_{RK_R}]$	None

Here, by phase we mean the number of states of a distribution of class `Markovian`. If the distribution is not Markovian, then there is a single phase. With these definitions, the table below illustrates how to specify in LINE a valid state for a station depending on its scheduling strategy. There, S is the number of servers of the queueing station. All state variables are non-negative integers. The `SchedStrategy.EXT` policy is used for the `Source` node, which may be seen as a special station with an infinite pool of jobs sitting in the buffer and a dedicated server for each class $r = 1, ..., R$.

States can be manually specified or enumerated automatically. LINE library functions for handling and generating states are as follows:

- `State.from_marginal`: enumerates all states that have the same marginal state $[n_1, n_2, ..., n_R]$.
- `State.from_marginal_and_running`: restricts the output of `State.from_marginal` to states with given number of running jobs, irrespectively of the service phase in which they currently run.
- `State.from_marginal_and_started`: restricts the output of `State.from_marginal` to states with given number of running jobs, all assumed to be in service phase $k = 1$.
- `State.from_marginal_bounds`: similar to `State.from_marginal`, but produces valid states between given minimum and maximum value of the number of resident jobs.
- `State.from_marg`: the class-summed counterpart of `State.from_marginal`. It fixes only how many jobs the station holds in TOTAL, and returns the union of the per-class enumeration over every split of that total the station can hold. Classes disabled at the station are excluded from the split enumeration, so a disabled class cannot silently absorb a job.
- `State.from_marg_and_started`: the same for the pair (total number of jobs, total number of started jobs), taking the union of `State.from_marginal_and_started` over every pair of per-class vectors consistent with both totals.
- `State.to_marginal`: extracts marginal statistics from a state, such as the total number of jobs in a given class that are running at the station in a certain phase.

Note that if a function call returns an empty state (`[]`), this should be interpreted as an indication that no valid state exists that meets the required criteria. Often, this is because the state supplied in input is invalid.

Example

We consider the example network in `cqn_multiserver`. We examine station 2 (the `Queue1` station), which is a three-server FCFS station. All four classes have exponential service times except class 2, which has Erlang-2 service times. We are interested in states with two running class-1 jobs and one running class-2 job, together with two buffered jobs belonging to classes 3 and 4, respectively. We can automatically generate this state space, which we store in the `space` variable, as:

```
space = State.from_marginal_and_running(model, 1, [2,1,1,1], [2,1,0,0])
```

Here, each row of `space` corresponds to a valid state. The argument `[2,1,1,1]` gives the number of jobs in the node for the 4 classes, while `[2,1,0,0]` gives the number of running jobs in each class. This station has four valid states, differing on whether the class-2 job runs in the first or in the second phase of the Erlang-2 and on the relative position of the jobs of class 3 and 4 in the waiting buffer.

To obtain states where the jobs have just started running, we can instead use

```
space = State.from_marginal_and_started(model, 1, [2,1,1,1], [2,1,0,0])
```

We see that the above state space restricted the one obtained with `State.from_marginal_and_running` to states where the job in class 1 is always in the first phase.

If we instead remove the specification of the running jobs, we can use `State.from_marginal` to generate all possible combinations of states depending on the class and phase of the running jobs. In the example, this returns a space of 20 possible states.

```
space = State.from_marginal(model, 1, [2,1,1,1])
```

Assigning a prior to an initial state

It is possible to assign the initial state to a station using the `set_state` function on that station's object. LINE offers the possibility to specify a prior probability on the initial states, so that if multiple states have a non-zero prior, then the solver will need to solve an independent model using each one of those initial states, and then carry out a weighting of the results according to the prior probabilities. The default is to assign a probability 1.0 to the *first* specified state. The functions `set_state_prior` and `get_state_prior` can be used to check and change the prior probabilities for the initial states specified for a station or stateful node.

7.1.2 Network states

A collection of states that are valid for each station is not necessarily valid for the network as a whole. For example, if the sum of jobs of a closed class exceeds the population of the class, then the network state would be invalid. To identify these situations, LINE requires to specify the initial state of a network using functions supplied by the `Network` class. These functions are `init_from_marginal`,

`init_from_marginal_and_running`, and `init_from_marginal_and_started`. They require a matrix `n` with elements (i, r) specifying the total number of resident class-r jobs at node i and the latter two require a matrix `s` with elements (i, r) with the number of running (or started) class-r jobs at node i. The user can also manually verify if the supplied network state is going to be valid using `State.is_valid`.

It is also possible to request LINE to automatically identify a valid initial state, which is done using the `init_default` function available in the `Network` class. This is going to select a state where:

- no jobs in open classes are present in the network;
- jobs in closed classes all start at their reference stations;
- the servers at each reference station are occupied by jobs of in class order, i.e., jobs in the firstly created class are assigned to the server, then spare server are allocated to jobs in the second class, and so forth;
- service or arrival processes are initialized in phase 1 for each job;
- if the scheduling strategy requires it, jobs are ordered in the buffer by class, with the firstly created class at the head and the lastly created class at the tail of the buffer.

The `init_from_avg_q_len` method is a wrapper for `init_from_marginal` to initialize the system as close as possible to the average steady-state distribution of the network. Since averages are typically not integer-valued, this function rounds the average values to the nearest integer and adjusts the result to ensure feasibility of the initialization.

7.1.3 Initialization of transient classes

Because of class-switching, it is possible that a class r with a non-empty population at time $t = 0$ becomes empty at some positive time $t' > t$ without ever being visited again by any job. LINE allows users to place jobs in transient classes and therefore it will not trigger an error in the presence of this situation. If a user wishes to prohibit the use of a class at a station, it is sufficient to specify that the corresponding service process uses the `Disabled` distribution.

Certain solvers may incur problems in identifying that a class is transient and in setting to zero its steady-state measures. For example, the `JMT` solver uses a heuristic whereby a class is considered transient if it has fewer samples than jobs initially placed in the corresponding chain the class belongs to. For such classes, `JMT` will set the values of steady-state performance indexes to zero.

7.1.4 State space generation

The state space of a model can be obtained by either invoking `model.state_space()` or `solver.state_space()` on an instant of the CTMC solver, where the latter returns the state space cached during the solution of the CTMC.

LINE supports two state space generation methods, configurable using the option `options.config.state_space_gen` of the CTMC solver. Details may be found in Table 8.2.3.1.

Table 7.2: State probability accessors

Method	Returns
`get_prob_sys`	Probability of each global state of the network
`get_prob_sys_aggr`	The same, aggregated by class over all stations, so that phase and buffer-order detail is summed out
`get_prob`	Probability mass function of the state of one station
`get_prob_aggr`	The same, aggregated by class: the joint per-class queue-length distribution at that station
`get_prob_marg`	Marginal queue-length distribution $P(n_{i,r} = k)$ of a single class r at station i, other classes marginalized out (`MVA`, `MAM`). Under `NC` the method takes only a station and returns the class-aggregated distribution at it
`get_prob_sys_marg`	Joint law of the per-station TOTAL queue lengths, $P(n_1 = k_1, \ldots, n_M = k_M)$, with the classes summed out (`NC` only). This is the joint counterpart of `getProbMarg` and the class-aggregated counterpart of `getProbSysAggr`; see §7.1.5

7.1.5 State probability analysis

Besides mean values, solvers that build an explicit state representation can return the stationary probability of the network states. Table 7.2 lists the available accessors, which differ in whether the state is global or local to a station, and in whether the phase information of non-exponential service is retained or aggregated away.

Reading these outputs requires the state encoding of Section 7.1.4: a state is a vector of the number of jobs of each class at each station, extended with the phase of each in-service job when the service process is non-exponential, and with the buffer order under FCFS scheduling, which is why FCFS state spaces are larger than the corresponding processor-sharing or infinite-server ones. The example files `statepr_aggr` and `statepr_allprobs_fcfs` show how to aggregate over classes and stations and how to enumerate all states of a FCFS network, from which quantities such as $P(\text{station empty})$ or $P(\text{total population} > k)$ follow by summation.

For example, the marginal queue-length distribution of class `cclass1` at `queue1` can be obtained as follows:

```
solver = MVA(model)
Pmarg = solver.get_prob_marg(queue1, cclass1)
# Pmarg[k] = P(n_{queue1, cclass1} = k)
```

Joint law of the per-station total queue lengths

The methods above answer either a per-class question at one station or a per-class question about the whole network. The remaining question, the JOINT law of the per-station TOTALS with the classes summed out, is answered by `get_prob_sys_marg` under the `NC` solver. The four state-probability methods answer questions on four different index sets:

Each value returned by `getProbSysMarg` is therefore the sum of `getProbSysAggr` over the whole set of per-class tables with the prescribed row sums, and that set grows combinatorially, so summation is not a route to it. For a closed multiclass product-form network with load-independent single-server queues

Method	Random variable	States at 3 stations, 3 classes, populations 2, 2 and 1
`getProbAggr`	the per-class counts at one station	6 / 6 / 3
`getProbSysAggr`	the whole station-by-class table of counts	108
`getProbMarg`	the total count at one station	6
`getProbSysMarg`	the vector of per-station totals, classes summed out	21

Table 7.3: The four state-probability methods answer questions on different index sets. Each `getProbSysMarg` value is the sum of `getProbSysAggr` over every per-class table with the given row sums.

plus infinite servers, the sum has instead a closed form. It is the permanent of a square demand matrix, built by repeating each class column as many times as that class has jobs and each station row as many times as that station holds, divided by the normalizing constant, by the factorial of each class population, and by one factorial per infinite-server station of the jobs it holds. The queueing stations need no such factorial of their own, the permanent supplying it. The permanent is evaluated with Ryser's inclusion-exclusion formula exploiting the column multiplicities [135], at a cost that grows with the product of the class populations rather than with the factorial of their sum.

The method takes an optional second argument selecting the estimator of the permanent: `'exact'` (the default and the only exact one), `'bethe'` [155], `'heur'` [143], `'huberlaw'` [79] and `'adapart'` [91]. The four approximations require a demand matrix with full support and are refused, rather than applied to a floored matrix, whenever a class has zero demand at a station that holds jobs: Sinkhorn scaling does not converge without full support, and the Bethe estimate is a state-dependent lower bound whose gap does not cancel when the values are normalized against one another. The exact estimator has no such restriction, since a station holding no jobs contributes no row, a class with no jobs contributes no column, and a zero demand is an ordinary zero entry.

The closed form relies on one factorial per queueing station, which neither a multiserver nor a load-dependent station provides. Such models, as well as open and mixed ones, are refused by name rather than approximated. The following example sweeps the whole lattice of total states of a closed network, so that the reported probabilities sum to one:

```
solver = NC(model)
p, logp = solver.getProbSysMarg([1, 1, 1])
pb, _ = solver.getProbSysMarg([1, 1, 1], 'bethe')
```

Both the probability and its logarithm are returned, the latter being formed first so that a state whose probability underflows in double precision still reports a finite logarithm. A state whose total does not match the closed population is not an error but has probability zero, which is what allows the lattice sweep above to be written without a feasibility test. The example files `statepr_sys_marg` show the sweep in each language together with the check that the first moments of the law reproduce the mean queue lengths returned by the `CTMC` solver.

Transient state probabilities

For solvers that expose the underlying state space, LINE also provides transient counterparts of the steady-state probability methods, giving the distribution over network states at a finite time horizon (controlled by the `timespan` option). These are useful for analyzing warm-up behavior, ramp-up of arrival rates, or short-term reliability questions. Per-station transient probabilities are returned by `get_tran_prob` (full state) and `get_tran_prob_aggr` (aggregated by class), while system-level joint transient probabilities are returned by `get_tran_prob_sys` and `get_tran_prob_sys_aggr`. All four methods are currently supported by the `CTMC` solver. `JMT` additionally provides the aggregated form (`get_tran_prob_aggr`).

The following example computes the per-class transient probabilities at `queue1` from the default initial state up to time $t = 1$:

```
solver = CTMC(model, timespan=[0, 1])
Pi_t, SSnode_a = solver.get_tran_prob_aggr(queue1)
# Pi_t[k, :] gives the probability vector at the k-th time step
# SSnode_a lists the corresponding aggregated marginal states
```

7.2 Transient analysis

So far, we have seen how to compute steady-state average performance indexes, which are given by

$$E[n] = \lim_{t \to +\infty} E[n(t)]$$

where $n(t)$ is an arbitrary performance index, e.g., the queue-length of a given class at time t.

We now consider instead the computation of the quantity $E[n(t)|s_0]$, which is the *transient average* of the performance index, conditional on a given initial system state s_0. Compared to $n(t)$, this quantity averages the system state at time t across all possible evolutions of the system from state s_0 during the t time units, weighted by their probability. In other words, we observe all possible stochastic evolutions of the system from state s_0 for t time units, recording the final values of $n(t)$ in each trajectory, and finally average the recorded values at time t to obtain $E[n(t)|s_0]$.

7.2.1 Computing transient averages

The computation of transient metrics proceeds similarly to the steady-state case. We first obtain the handles for transient averages:

```
model = Gallery.gallery_cqn(2)  # closed single class queueing network
handles = model.get_tran_handles()
Qt = handles.Qt
Ut = handles.Ut
Tt = handles.Tt
```

After solving the model, we will be able to retrieve *both* steady-state and transient averages as follows

```
results = CTMC(model, timespan=[0,1]).tran_avg(Qt, Ut, Tt)
QNt = results.QNt
UNt = results.UNt
TNt = results.TNt
# matplotlib.pyplot.plot(QNt.t, QNt.metric)
```

The transient average queue-length at node i for class r is stored within `QNt` in row i and column r.

Note that the above code specifies a maximum time t for the output time series. This can be done using the `timespan` solver option. This applies also to average metrics. In the following example, the first model is solved at steady-state, while the second model reports averages at time $t = 1$ after initialization

```
# Steady-state analysis
CTMC(model).avg_table().print()

# Transient analysis at t=1
CTMC(model, timespan=[0, 1]).avg_table().print()
```

7.2.2 First passage times into stations

When the model is in a transient, the average state seen upon arrival to a station changes over time. That is, in a transient, successive visits by a job may experience different response time distributions. The function `get_tran_cdf_resp_t`, implemented by `JMT` offers the possibility to obtain this distribution given the initial state specified for the model. As time passes, this distribution will converge to the steady-state one computed by solvers equipped with the function `get_cdf_resp_t`.

However, in some cases one prefers to replace the notion of response time distribution in transient by the one of *first passage time*, i.e., the distribution of the time to complete the *first visit* to the station under consideration. The function `get_tran_cdf_pass_t` provides this distribution, assuming as initial state the one specified for the model, e.g., using `set_state` or `init_default`. This function is available in `FLD` and `JMT` and has a similar syntax as `get_cdf_resp_t`. A steady-state passage CDF is currently provided only by `FLD` (as `get_cdf_pt`); the generic `get_cdf_pass_t` dispatcher is reserved for future implementations and currently raises an error on all solvers.

7.3 Advanced metrics

This section collects analyses that go beyond the standard per-station and system-wide performance indexes. They include metrics specific to fork-join structures and finite capacity regions, the Age of Information for status-update systems, and user-defined reward models evaluated over the underlying Markov chain.

7.3.1 Fork-join metrics

Two metrics report a fork-join section as a whole rather than station by station. `FJQLen` is the mean number of jobs in the section, counted from dispatch at the fork to synchronization at the join, and `FJRespT` is the corresponding mean sojourn time, that is the time for all sibling tasks of a job to complete and be joined. Both therefore aggregate over every station lying between a fork and its join, which the ordinary per-station metrics cannot express. They are appended to the `get_avg_table` output whenever the model contains a fork-join section, and are computed by the `MVA`, `JMT` and `LDES` solvers.

7.3.2 Finite capacity region queue length

When a station is part of a Finite Capacity Region (FCR) with `Queue` drop strategy, the reported queue length (`QLen`) includes both jobs currently inside the station and jobs that are blocked waiting to enter the FCR. This semantic ensures consistency with blocking queueing network models where blocked jobs are logically associated with their destination queue. For example, if an FCR with capacity 2 contains a queue with 2 jobs, and 1 additional job is blocked waiting to enter, the reported queue length will be 3. This convention is consistent with JMT's handling of FCR blocking.

7.3.3 Age of Information analysis

`FLD` supports the computation of Age of Information (AoI) metrics for single-queue open systems with capacity constraints. AoI measures the freshness of information in status update systems, defined as the time elapsed since the last successfully received update was generated. When the model has a valid AoI topology (Source → Queue → Sink with one open class, one queue, one server, and queue capacity 1 or 2), the `mfq` method automatically switches to the AoI MFQ analysis and computes exact AoI and Peak AoI distributions. The native Python solver exposes the same AoI API:

```
solver = FLD(model, method='mfq')
solver.runAnalyzer()
AoI, PAoI, aoi_table = solver.getAvgAoI()
AoI_cdf, PAoI_cdf = solver.getCdfAoI()
```

7.3.4 Reward models

LINE supports the definition of custom reward functions on network models, enabling the computation of application-specific performance metrics beyond the standard measures of queue length, utilization, and response time. Rewards associate a numerical value with each state of the underlying continuous-time Markov chain, allowing analysts to define metrics that capture domain-specific objectives such as energy consumption, revenue, or composite performance indicators. The reward framework supports both steady-state analysis, which computes the expected reward in equilibrium, and transient analysis, which tracks the evolution of

Table 7.4: Reward function templates in the `Reward` class (MATLAB method names shown, with the Python-native name in parentheses where it differs). In Java, the same metrics are written directly as `RewardFunction` lambdas.

Template	Arguments	Description
`queueLength` (`queue_length`)	`node`, [`jobclass`]	Number of jobs at `node`, totalled across classes or restricted to `jobclass` when supplied.
`utilization`	`node`, [`jobclass`]	Fraction of busy servers at `node`, computed as min(jobs, **`nservers`**); per-class when `jobclass` is supplied.
`blocking`	`node`	Indicator equal to 1 when `node` is at capacity (buffer full) and 0 otherwise.
`custom`	`fn`	Returns the user-defined reward function `fn` unchanged, documenting the intent that it is a custom reward.

reward values over time. This feature is available through the `CTMC` solver, which constructs the Markov chain representation of the network and evaluates reward functions over the state space.

Reward functions are defined using the `set_reward` method on the `Network` object. This method takes two arguments: a string name for the reward, which is used to identify it in the results, and a function that maps a state accessor object to a scalar reward value. The state accessor provides the method `at`, which takes a node and a job class as arguments and returns the number of jobs of that class at that node in the current state.

```
model.set_reward('QueueLength', lambda state: state.at(queue, oclass))
```

This example defines a reward equal to the number of jobs of class `oclass` at the `queue` node. More complex reward functions can be constructed by combining state information from multiple nodes and classes, or by applying arbitrary mathematical operations to state variables.

To simplify common use cases, LINE provides pre-built reward templates through the `Reward` class. The `Reward.utilization` method creates a reward function that returns an indicator variable, taking the value 1 when the specified node is serving a job of the specified class and 0 otherwise. The `Reward.blocking` method creates a reward function that returns an indicator for buffer-full conditions, taking the value 1 when the node's buffer is at capacity and 0 otherwise.

```
model.set_reward('Utilization', Reward.utilization(queue, oclass))
model.set_reward('BlockingProb', Reward.blocking(queue))
```

The complete set of pre-built templates is summarized in Table 7.4. Each template is a factory that returns a reward function suitable for `set_reward`.

Once reward functions are defined on the model, they can be evaluated by creating a `CTMC` solver instance and calling the appropriate analysis methods. For steady-state analysis, the `get_avg_reward` method returns a vector of expected reward values in equilibrium, along with a list of reward names. For transient analysis, the `get_tran_reward` method returns, for each defined reward, the time-indexed trajectory of the expected reward $E[r(X(t))]$, together with the vector of time points and the reward names. Here $X(t)$ is the system state at time t and the expectation is taken over the transient distribution of the Markov chain, starting

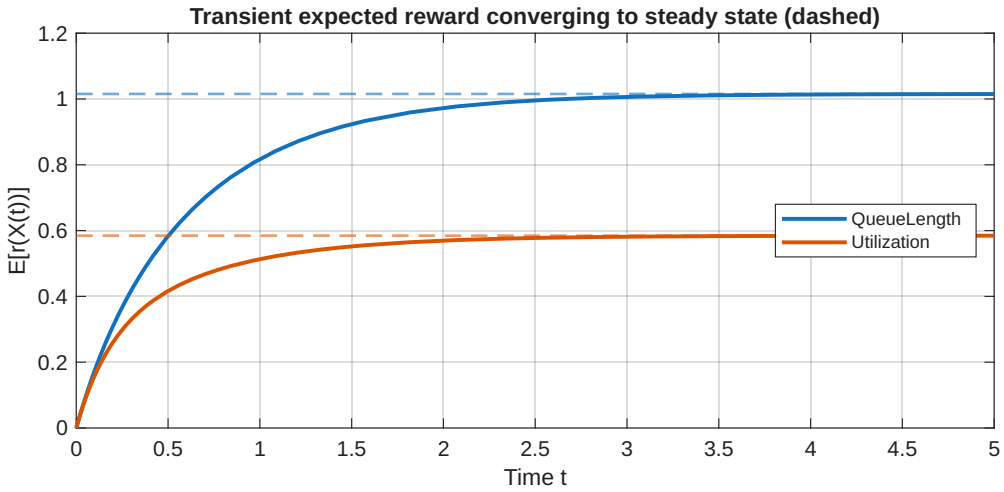


Figure 7.1: Transient expected reward $E[r(X(t))]$ for the queue length and utilization of an $M/M/1/3$ queue, converging over time to the steady-state values (dashed) computed by `get_avg_reward`.

from the model's initial state; a finite `timespan` must therefore be supplied to the solver.

```
solver = CTMC(model, timespan=[0, T])
R, names = solver.get_avg_reward()        # steady-state expected rewards
Rt, t, names = solver.get_tran_reward()   # transient E[r(X(t))] trajectories
```

The transient trajectories can be plotted to visualize how each reward relaxes from its initial value towards the steady-state equilibrium. For example, the following snippet overlays the transient expected rewards on their steady-state values, producing Figure 7.1.

The reward framework is particularly useful for defining objective functions in optimization problems, tracking custom performance indicators that combine multiple system states, or analyzing transient behavior of derived metrics. The examples `rewardModel_mm1k`, `rewardModel_templates`, and `rewardModel_aggregation` demonstrate various applications of reward models, including simple state-based rewards, the use of template functions, and the aggregation of reward values across multiple nodes or classes.

7.4 Sample path analysis

Transient analysis returns the expected value $E[X(t)]$ of a performance index, whereas sample path analysis returns a single realization of the process $X(t)$, that is one trajectory produced by a specific sequence of random events. A different random seed yields a different sample path, and averaging many realizations at a fixed time recovers the transient average. Four methods extract such a trajectory from the stochastic process underlying the network:

- `sample`: the time-varying state of a given stateful node, labelled with the events that changed it.
- `sample_aggr`: the same, with the state aggregated to the number of jobs of each class at the node.
- `sample_sys`: the state of every stateful node in the model.
- `sample_sys_aggr`: the aggregated state of every stateful node.

The `JMT` solver supports only the aggregate variants, since the simulator does not export phase-level information; the `SSA` solver supports all four. For example, the following extracts ten samples for an $APH(2)/M/1$ queue, where the first column is the time at which the station enters the state shown next to it:

```
model = gallery_aphm1()
sample_path = JMT(model).sample_aggr(model.get_nodes()[1], 10)
print(sample_path.t, sample_path.state)
```

7.4.1 Confidence intervals for steady-state quantiles

A tail service-level agreement is written against a quantile rather than a mean, and that estimate needs an interval of its own. LINE provides two automated procedures for this purpose, `sim.fquest` (FQUEST), which works from a single sample path of arbitrary length [100], and `sim.firquest` (FIRQUEST), which works from a set of independent replications of equal length [101]. Both act on a sequence of observations rather than on a model, so they apply to any output series, and neither requires the user to choose a batch size, a batch count or a warm-up length. The two hypothesis tests they use internally are also exposed on their own as `sim.vonneumann` and `sim.shapirowilk`.

The output series must be continuous, that is a response, waiting or sojourn time: the method must not be applied to integer-valued output such as a queue length, whose marginal law has no density, nor to heavy-tailed service times. When the sample is too small to support the underlying asymptotics the procedure returns a conservative fallback interval and sets the `heuristic` field of the result, which should be inspected before an interval is quoted. As an illustration we estimate the 0.9-quantile of the waiting time in an $M/M/1$ queue with $\lambda = 0.8$ and $\mu = 1$:

```
rng = np.random.default_rng(1)
A = rng.exponential(1/0.8, 200000)
S = rng.exponential(1.0, (200000, 1))
W = qsys.qsys_tandem_lindley(A, S)['W'][:, 0]
r = sim.fquest(W, 0.9, 0.05)
print(r['lower'], r['estimate'], r['upper'])
# 9.9825 10.5930 11.2035
```

7.4.2 Conditional waiting times along a sample path

Useful for validating a simulated trajectory one customer at a time, LINE implements the conditional moments of Lindley's recursion [96, 118]. Given the waiting time of a customer, `qsys.qsys_mm1_lindley`

returns the exact conditional moments of the waiting time of the next customer in an $M/M/1$ queue, and `qsys.qsys_hh1_lindley` does the same for hyperexponential interarrival and service times. Being conditional on the current state rather than stationary, they remain finite at any load, including a saturated queue.

For stations in tandem, `qsys.qsys_tandem_lindley` propagates the recursion over an arbitrary number of stations and arbitrary interarrival and service times, reproducing a direct event-driven simulation of the tandem to machine precision, while `qsys.qsys_mm1_tandem_lindley` gives the conditional mean waiting time at the downstream station of a two-station $M/M/1$ tandem.

7.4.3 Tail bounds for a tandem of two queues

End-to-end response times in a network are far harder to characterize than single-queue ones, since the waiting times a job accumulates along its path are neither independent nor identically distributed. For a tandem of two single-server FCFS stations fed by a renewal arrival process, LINE provides the polynomial-exponential upper bounds of [43] on the tails of the end-to-end waiting time and sojourn time, through `qsys.qsys_tandem_ub_ciucu`. Both stations serve the same hyperexponential law, a single phase giving exponential service, and the interarrival time enters only through its Laplace-Stieltjes transform, supplied as a function handle, so any light-tailed renewal input is admissible.

The bounds decay exponentially, at the rate fixed by the positive root of the transform equation that balances a service time against an interarrival time, multiplied by a polynomial of degree one; that polynomial factor is what lets them follow the concave bend of the tail a purely exponential bound cannot. They are attained on an `M/M/1` tandem, where they return the exact tails of both quantities. The bound on the waiting time is available for exponential service only and is returned as `NaN` otherwise. As an illustration we bound the sojourn time of a `D/M/1` tandem at three quarters utilization with unit service rate, at horizons 5, 10 and 20:

```
lst = lambda s: exp(-4 * s / 3)
dlst = lambda s: (4 / 3) * exp(-4 * s / 3)
r = qsys.qsys_tandem_ub_ciucu([5, 10, 20], lst, 1.0, 1.0, dlst)
print(r['theta'], r['S'])
# 0.4544 [0.4937 0.0912 0.0018]
```

The bound is sharp where the input is not too variable. Against an exact reference for an `Erlang(2)/M/1` tandem it exceeds the true sojourn-time tail by 2% where that tail is one in a hundred, and by 0.6% where it is five in ten billion, and it carries the correct asymptotic slope; under hyperexponential service with coefficient of variation between two and four it loosens to about a factor of two.

7.4.4 Multiserver queues with phase-type service

A queue with several servers is usually approximated by a single fast server, which reproduces the mean service time but neither the shape of the service law nor the correlation of the arrival stream. LINE instead solves two multiserver families exactly. `qsys.qsys_mapmc` solves the $MAP/M/c$ queue [66, 122], and

`qsys.qsys_mapphc` solves $MAP/PH/c$, returning the queue length distribution, the per-server utilization, the probability of waiting, the moments of the waiting time and its complementary distribution function.

The state of the $MAP/PH/c$ chain is the *multiset* of the phases occupied by the busy servers: since the servers are identical, their identities carry no information, and a configuration $n = (n_1, \ldots, n_{m_s})$ with $\sum_i n_i = k$ records how many of the k busy servers sit in phase i. There are $\binom{m_s+k-1}{k}$ such configurations rather than m_s^k [5]; for $m_s = 5$ and $c = 6$ that is 210 against 15625, which is what keeps the exact solution within reach. Levels $0, \ldots, c-1$ form the boundary and levels c and above repeat, so the queue-length tail is matrix-geometric.

The waiting time is phase type and is obtained without approximation. A customer that finds j others waiting must wait for exactly $j+1$ service completions, so its delay is the $(j+1)$-st event time of the Markovian arrival process formed by the configuration chain; folding the matrix-geometric level distribution over j leaves a linear matrix differential equation, from which each moment follows by one solve of a generalized Sylvester equation. With a single service phase the whole construction collapses onto the Erlang-C formulae, which it reproduces to fourteen digits.

```
r = qsys.qsys_mapphc(D0, D1, alpha, S, 3)
print(r.meanQueueLength, r.meanWaitingTime, r.probWait)
```

Both functions are also reached automatically by `SolverMAM` whenever a station in an open model matches them, so a multiserver queue fed by a correlated stream or serving a phase-type law is answered exactly rather than by the surrogate.

7.4.5 Per-class waiting times under general service

When customers of different classes need different service laws, and those laws are not phase type, the matrix-analytic machinery of the previous section no longer applies: no finite set of phases carries a deterministic or a uniform service time. `qsys.qsys_mmapgk1` solves this case through transforms, returning the waiting time moments, transform and distribution function of *each* class of a marked multiclass `MMAP/G/1` FCFS queue [72]. The arrival process is marked, so the classes may be correlated with each other and with the arrival epochs.

Under FCFS a customer waits exactly the workload it finds on arrival, so the joint transform of workload and arrival phase satisfies a linear system whose coefficients are the arrival blocks and the service transforms. The vector of idle probabilities it needs is recovered without searching for the roots of a determinant, from the generator of the arrival phase observed when the queue empties by one customer. Each class then sees that workload biased by its own arrival block, so classes with identical service laws still differ in delay whenever their arrival epochs do.

```
r = qsys.qsys_mmapgk1([D0, D1 + D2, D1, D2], [Det(0.5), Uniform(0.1, 0.9)])
print(r.meanWaitingTime)
```

With one class, Poisson arrivals and exponential or deterministic service the results reduce to the classical `M/M/1` and Pollaczek-Khinchine formulae, which they reproduce exactly.

Chapter 8

Solvers and solution methods

This chapter documents how each solver is used: the class of models it accepts, the method names its `options.method` recognises, its configuration options, and the interfaces that exist only for that solver. The algorithms behind those method names, the internal architecture of each solver and the bibliographic sources are documented in the LINE internals manual (`LINE-internals-python.pdf`), which devotes one chapter to each solver and is not summarized here.

8.1 Choosing a solver

8.1.1 Loading a solver by name

We now describe the solution methods available within the solvers. Table 8.1 provides a global summary, while Figure 8.1 organizes the same methods by class of stochastic model. Some listed methods (e.g., `mg1`) are not associated with a specific solver because they do not fall within one of the reference formalisms. A solver that runs these methods can be instantiated as follows:

```
solver = LINE.load('mg1', model)
solver.avg_table().print()
```

Note that the `LINE.load` notation can also be used to instantiate a custom solver pre-configured with the specified method. For example

```
solver = LINE.load('ctmc', model)
```

runs the `CTMC` solver with default options. Solver-specific methods can be specified by appending their name to the method option, e.g. this command creates the `CTMC` solver with `gpu` method enabled:

```
solver = LINE.load('ctmc.gpu', model)
```

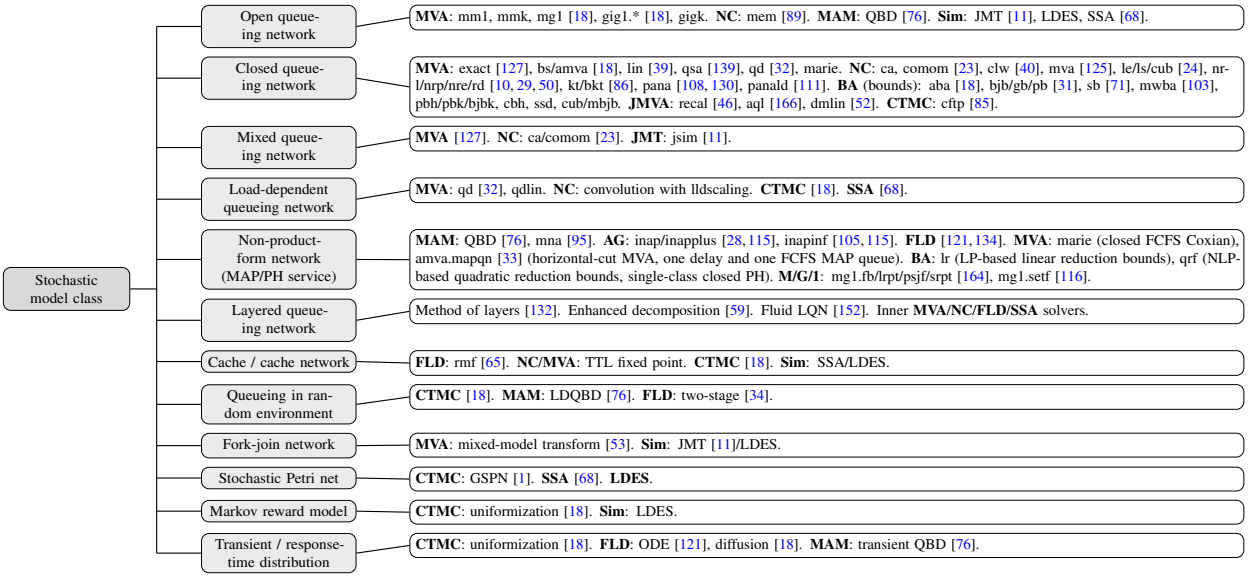


Figure 8.1: Taxonomy of solution techniques available in LINE for each class of stochastic model. Each leaf lists the short method name (as accepted by `options.method`) with its bibliographic reference; entries without a reference denote simulation backends or aliases. See Table 8.1 for the solver exposing each method.

8.1.2 Technique taxonomy and methods table

Figure 8.1 organizes the methods by class of stochastic model, and Table 8.1 lists, for each solver, the method names it accepts.

Table 8.1: Solution methods for `Network` solvers.

Solver	Method	Description	Refs.
`CTMC`	`default`	Solution based on global balance	[18, § 2.1.2]
`CTMC`	`exact`	Explicit alias of `default`: it pins the state-space path at the call site so that a later change of what `default` selects cannot re-baseline the result	–
`CTMC`	`cftp`	Perfect sampling of the stationary distribution by monotone coupling from the past; no state space is generated, so the memory gate does not apply	[85]
`CTMC`	`cftp.approx`	`cftp` with a deterministic cost per sample in place of the unbounded worst case, at the price of a residual bias	[85]
`CTMC`	`mdd`	Multi-valued decision-diagram aggregation of a closed single-class chain: K coupled level-CTMCs instead of the $\|S\|$-state generator, with an exactness certificate reported per solve	[110]
`LDES`	`default`	Discrete-event simulator using the SSJ library	–
`FLD`	`default`	ODE-based mean field approximations	[121, 134]
`FLD`	`matrix`	Alias for the `default` method	[121, 134]

Continued on next page

Table 8.1 – Solution methods for `Network` solvers. *Continued from previous page*

Solver	Method	Description	Refs.
FLD	closing	Fluid with closing method for open classes	[18, p. 507]
FLD	statedep	Kurtz's mean field ODEs for closed models	[121]
FLD	pnorm	ODE approximation using p-norm smoothing for the processor-share constraint	[134]
FLD	softmin	Smoothed `statedep` with *softmin* replacing *min* functions	–
FLD	diffusion	Diffusion approximation via Euler-Maruyama SDEs	[18, § 10.1.1]
FLD	mfq	Markovian fluid queue for single-queue systems	[75]
FLD	rmf	Refined mean field approximation for multi-list cache analysis	[65]
FLD	tbi	Cell-decomposed solution of the closing ODEs for large closed models	[142]
FLD	minnormal	Second-order (min-normal) moment closure, whose error stays bounded as $\rho \to 1$; the only fluid method serving `GPS` and the one reporting queue-length variances	[69]
FLD	refined	$O(1/N)$ refinement term added to the mean-field fixed point, closed models only; not available in the JLINE JAR	[64]
FLD	kp	Fluid *and* diffusion limits of the open time-inhomogeneous $(\mathrm{MAP}_t/\mathrm{Ph}_t/\infty)^N$ network, the mean and the covariance integrated jointly	[87]
FLD	dae	The min-normal closure stated and solved as one differential-algebraic system: in steady state the drift residual, the population conservation and the closure consistency are solved simultaneously by damped Newton, and over a finite horizon the index-1 DAE is integrated by RODAS, so the variance is the one the trajectory actually had. Finite capacity regions and station buffers are carried as algebraic constraints	[69, 70]
JMT	default	Alias for the `jsim` method	–
JMT	jmva	Alias for the `jmva.mva` method	–
JMT	jmva.mva	Exact MVA in JMVA	[127]
JMT	jmva.recal	Exact RECAL algorithm in JMVA	[46]
JMT	jmva.comom	Exact CoMoM algorithm in JMVA	[23]
JMT	jmva.amva	Approximate MVA, alias for `jmva.bs`.	–
JMT	jmva.aql	AQL algorithm in JMVA	[166]
JMT	jmva.bs	Bard-Schweitzer algorithm in JMVA	[18, § 9.1.1]
JMT	jmva.chow	Chow algorithm in JMVA	[41]
JMT	jmva.dmlin	De Souza-Muntz Linearizer in JMVA	[52]
JMT	jmva.lin	Linearizer algorithm in JMVA	[39]
JMT	jmva.ls	Logistic sampling in JMVA	[24]
JMT	jsim	Exact discrete-event simulation in JSIM	[11]
JMT	replication	Transient simulation via independent replications	–
AG	inap	Iterative Numerical Approximation Procedure via RCAT	[28, 115]
AG	inapplus	Improved INAP with weighted rates (no normalization)	[28, 115]
AG	inapinf	INAP with matrix-geometric infinite-state components	[105, 115]
MAM	default	Matrix-analytic solution of structured QBDs	[76]
MAM	dec.source	Decomposition with arrivals as from the source	–
MAM	dec.poisson	Decomposition based on Poisson arrival flows	–
MAM	dec.mmap	ETAQA departure decomposition: the departure process of each queue is approximated by a MAP and fed to the next	[129]
MAM	dec.source.mmap	`dec.source` with the per-class (marked) flows retained; selected automatically on general fork-join topologies	–
MAM	ldqbd	Level-dependent QBD: the queue length is solved from the rate matrices R_n of a level-dependent quasi-birth-death process, which admits load-dependent rates and, for a multiserver station with phase-type service, the exact multiset-of-phases chain	[5, 124]
MAM	mna	Decomposition based on MNA method	[95]

Continued on next page

Table 8.1 – Solution methods for `Network` solvers. *Continued from previous page*

Solver	Method	Description	Refs.
MAM	bgchain	Mixed and closed networks: the closed classes are solved exactly as a background modulating chain and each open station as a level-dependent QBD driven by it; with no open class the chain alone answers, exactly	[37]
MVA	default	Approximate MVA, same as qd option	–
MVA	amva	Approximate MVA, same as qd option	–
MVA	aql	Aggregate queue length approximate MVA, single-server product-form only	[166]
MVA	bs	Bard-Schweitzer approximate MVA	[18, § 9.1.1]
MVA	lin	Linearizer approximate MVA	[39]
MVA	dmlin	de Souza e Silva-Muntz improved Linearizer, same estimator at a reduced cost	[52]
MVA	lcp	Bard large-customer-population approximate MVA, the arrival-instant queue length estimated by the time-averaged one	[9]
MVA	chow	Chow second approximation, a θ-correction taken off the Bard LCP solution	[42]
MVA	pamb, pami, pamt	Hsieh-Lam noniterative proportional approximation methods: a proportional seed of the queue lengths followed by a fixed number of unrolled MVA steps	[78]
MVA	clust	de Souza e Silva-Lavenberg-Muntz clustering approximation for models with a large number of chains	[51]
MVA	qli, fli	Wang-Sevcik queue-line and fraction-line arrival-instant estimators	[158]
MVA	ab, schmidt, schmidt-ext	Akyildiz-Bolch and Schmidt multiserver corrections of approximate MVA	[147]
MVA	tay	Tay-Suri arrival-instant approximate MVA, single-server stations only	[150]
MVA	sqni	Single-queue numerical iteration, closed form for one queueing station with a delay; offered only when the model has that shape	–
MVA	marie	Marie aggregation-decomposition, closed FCFS non-exponential (Coxian) service	–
MVA	amva.mapqn	Horizontal-cut mean value analysis, closed model of one exponential delay and one FCFS queue with a MAP service per class	[33]
MVA	qd	Queue-dependent approximate MVA	[32]
MVA	qdlin	Queue-dependent Linearizer approximate MVA	–
MVA	qsa	Queue-shift approximate MVA, single-server product-form only	[139]
MVA	exact	Exact solution, method depends on model	–
MVA	mva	Alias for the mva.amva method	[127], [21]
MVA	oi	Exact order-independent analyzer for closed networks with order-independent (OI) or pass-and-swap stations; auto-selected under default/exact and reported as the actual method	–
MVA	sqd	Smith Queue Decomposition, single-chain closed blocking-after-service (BAS) networks; offered only when the model is BAS	–
MVA	qna	Queueing Network Analyzer (two-moment decomposition) - open	[161]
MVA	rqna	Robust Queueing Network Analyzer (indices of dispersion) - open	[162]
MVA	rqt	Robust Queueing Theory, worst-case system time bound - single-class open	[8]
MVA	scat	Neuse-Chandy SCAT approximate MVA, single-server product-form only	[114]
MVA	sum	Summation method (SUM/ESUM); closing method for open and mixed models	[18, § 9.2, § 10.1.4.4, § 10.1.5]
MVA	esum	Alias for the sum method	[18, § 10.1.4.4]
BA	auto.upper	Tightest feasible noniterative upper bound, selected automatically	–
BA	auto.lower	Tightest feasible noniterative lower bound, selected automatically	–
BA	aba.upper	Asymptotic bound analysis (upper bounds)	[18, § 9.4]

Continued on next page

Table 8.1 – Solution methods for `Network` solvers. *Continued from previous page*

Solver	Method	Description	Refs.
BA	`aba.lower`	Asymptotic bound analysis (lower bounds)	[18, § 9.4]
BA	`mwba.upper`	Majumdar-Woodside robust box bounds, multiclass NBUE (upper bounds)	[103]
BA	`mwba.lower`	Majumdar-Woodside robust box bounds, multiclass NBUE (lower bounds)	[103]
BA	`bjb.upper`	Balanced job bounds (upper bounds)	[31, Table 3]
BA	`bjb.lower`	Balanced job bounds (lower bounds)	[31, Table 3]
BA	`gb.upper`	Geometric square-root bounds (upper bounds)	[31]
BA	`gb.lower`	Geometric square-root bounds (lower bounds)	[31]
BA	`pb.upper`	Proportional bounds (upper bounds)	[31, Table3]
BA	`pb.lower`	Proportional bounds (lower bounds)	[31, Table3]
BA	`sb.upper`	Simple bounds (upper bounds, Thm. 3.2, $n = \min(3, N)$)	[71, Table3]
BA	`sb.lower`	Simple bounds (lower bounds, Eq. 1.6)	[71, Table3]
BA	`harel.upper/lower`	Harel-Namn-Sturm bounds extrapolated from the exact throughput at small populations, distinct from the power-sum `sb` family of the same paper	[71]
BA	`looping.upper/lower`	Eager multiclass queue-length bracket built from the unaccounted-congestion heaps, the initial estimate of the multiclass bound hierarchy	[55]
BA	`pbh.upper/lower`	Performance bound hierarchy, level-parameterized (Eager-Sevcik)	–
BA	`pbk.upper/lower`	Iterative proportional bounds PB(k) with delay	–
BA	`bjbk.upper/lower`	Iterative balanced job bounds BJB(k) with delay	–
BA	`cbh.upper/lower`	Convolutional bound hierarchy, level-parameterized (Dowdy et al.)	–
BA	`ssd.upper/lower`	Multiserver disaggregation bounds (Suri-Dallery)	–
BA	`cub.upper`	Composite upper bound, multiclass, upper-only (Kerola)	–
BA	`mbjb.lower`	Multiclass balanced job bounds lower (Kerola eq. 10; seeds `cub`)	–
BA	`sib.upper/lower`	Successively-improving bounds, level-parameterized (Srinivasan; delay-free)	–
BA	`ldbcmp.lower`	Load-dependent BCMP closed-open equivalence lower bound (Anselmi-Cremonesi)	–
BA	`lr.upper/lower`	Linear Reduction bounds, pure LP (`linprog`/HiGHS/simplex), single-class closed; bare `lr` means `lr.upper`	–
BA	`qr` / `qrf.mmi`	Quadratic Reduction Framework, NLP-based (mutual-information objective), single-class closed PH service; single-server stations only, delay stations rejected	–
BA	`qrf.mmi.linear`	QRF with an explicit linear constraint representation; the objective is still the nonlinear MEM entropy, so this is an NLP and is *not* the same as `lr`	–
BA	`qrf.mem/mmi.ld`	Further QRF reduction sub-methods (maximum entropy, load-dependent); NLP-based	–
BA	`qrf.bas.*/rsrd`	QRF blocking-after-service and RS-RD reduction bounds; LP-based	–
BA	`qrf.bethe`	QRF with the tree-reweighted (Bethe) free entropy at $\lambda = 1/M$ in place of the mutual-information objective	[26], [157]
BA	`qrf.bas.bethe`	The same Bethe free entropy maximized over the blocking-after-service polytope	[26], [157]
BA	`mapamva.upper/lower`	MAP-AMVA bounds: a linear program over the exact mean-value balances of a closed MAP queueing network in the per-phase variables, the only `BA` family that consumes the CORRELATION between successive services rather than the service mean alone. Single-class closed, single-server, no delay, FCFS at the MAP station	[33]
BA	`scb.upper/lower`	Single-class bounds bracketing the aggregated multiclass system; `scb.lower` is the exact single-class throughput	[54]

Continued on next page

Table 8.1 – Solution methods for `Network` solvers. *Continued from previous page*

Solver	Method	Description	Refs.
BA	bgt.upper	Piecewise-linear-Lyapunov queue-length upper bound, open multi-class Markovian networks	[13]
BA	bpt.lower	Achievable-region sojourn-time lower bound, open multiclass Markovian networks	[14]
BA	snc.upper	Stochastic network calculus: MGF envelope bound on the response-time and queue-length TAIL, feed-forward open networks, any work-conserving policy	[57]
BA	spnlp.upper/lower	Petri-net moment relaxation: the uniformized evolution equation written for the first two marking moments and their covariances, plus the behavioural, probabilistic and Little's-law families, gives a polytope over which each measure is minimized and maximized. Exponential firing times; offered only on a model holding `Transition` nodes, and the only `BA` family offered there	[98]
BA	spnlp.op.upper/lower	The same relaxation without its second-moment, covariance and Little's-law families, so it rests on flow balance, the place invariants and the state equation alone: it needs only a mean firing time and admits any phase-type law, at the price of a much looser bracket	[98]
MVA	gig1.allen	Allen-Cunneen formula - GI/G/1	[18, § 6.3.4]
MVA	gig1.heyman	Heyman formula - GI/G/1	–
MVA	gig1.kingman	Kingman upper bound- GI/G/1	[18, § 6.3.6]
MVA	gig1.klb	Kramer-Langenbach-Belz formula - GI/G/1	[18, § 6.3.4]
MVA	gig1.kobayashi	Kobayashi diffusion approximation - GI/G/1	[18, § 10.1.1]
MVA	gig1.marchal	Marchal formula - GI/G/1	[18, § 10.1.3]
MVA	gigk	Kingman approximation - GI/G/k	
MVA	mg1	Pollaczek–Khinchine formula - M/G/1	[18, § 3.3.1]
MVA	mg1.fb	Feedback/LAS scheduling - M/G/1/FB	[164]
MVA	mg1.lrpt	Longest remaining PT - M/G/1/LRPT	[164]
MVA	mg1.psjf	Preemptive SJF - M/G/1/PSJF	[164]
MVA	mg1.srpt	Shortest remaining PT - M/G/1/SRPT	[164]
MVA	mg1.setf	Shortest elapsed time first - M/G/1/SETF	[116]
MVA	mm1	Exact formula - M/M/1	[18, § 6.2.1]
MVA	mmk	Exact formula - M/M/k (Erlang-C)	
NC	default	Automated choice of solution method	–
NC	exact	Automated choice of exact solution method.	–
NC	ca	Multiclass convolution algorithm (exact): Buzen's recursion generalized to multiple closed chains	[22], [126]
NC	clw	Choudhury-Leung-Whitt generating function inversion (exact); limited load-dependent stations handled via Bertozzi-McKenna transforms	[40], [12]
NC	comom	Class-oriented method of moments for homogeneous models (exact)	[23]
NC	comomld	Class-oriented method of moments for load-dependent models (exact), via factorization of the load-dependent normalizing constant	[29]
NC	cub	Grundmann-Moeller cubature rules (alias `gm`)	[24]
NC	ger	Exact normalizing constant in closed form by iterated residues of the rational generating function, one class per elimination	[67]
NC	gleint	Gauss-Legendre evaluation of the normalizing-constant integral	–
NC	mva	Product of throughputs on MVA lattice (exact)	[125, Eq. (47)]
NC	imci	Improved Monte carlo integration sampler	[159]
NC	is	Importance sampling over job orderings; closed models only, load-independent, load-dependent, order-independent and pass-and-swap	[44]
NC	kt	Knessl-Tier asymptotic expansion	[86]

Continued on next page

Table 8.1 – Solution methods for `Network` solvers. *Continued from previous page*

Solver	Method	Description	Refs.
NC	bkt	Knessl-Tier asymptotic expansion minus the exact Stirling remainder $\log N_r! - (N_r \log N_r - N_r + \frac{1}{2}\log(2\pi N_r))$ that steepest descent drops in each Laplaced class direction (BKT, not part of the reference); with a think time it coincides with `ble`	[86]
NC	lekt	The estimator `ble` and `bkt` both compute, evaluated on the cheaper side (R-dimensional saddle point when $R \leq M$, M-dimensional logistic mode otherwise), with the $Z = 0$ constant $M(1 - \log(2\pi)/2) - r(N + M)$ so that both sides agree there too (not part of the references)	[24, 86]
NC	le	Logistic asymptotic expansion	[24]
NC	ble	Logistic asymptotic expansion with an empirical $(M - 1)(1 - \log(2\pi)/2)$ correction of the $\epsilon \to 0$ evaluation (BLE, not part of the reference); default for normally-loaded closed models	[24]
NC	aghq	Adaptive Gauss–Hermite quadrature of the simplex factor of the McKenna–Mitra integral, q nodes per simplex direction (`options.config.aghq_nodes`, default 3) at cost q^{M-1}; $q = 1$ reproduces `le` exactly	[97]
NC	ls	Logistic sampling	[24]
NC	bk	Birman-Kogan saddle-point generating-function evaluation, exact pole treatment for bottlenecked dedicated chains	[16]
NC	bkue	Birman-Kogan single-chain uniform expansion, pole and saddle combined via erfc	[16]
NC	lc	Birman-Kogan load concealment, Gauss-Seidel over single-chain solves	[16]
NC	lc.ue	`lc` with the `ue` inner single-chain solver	[16]
NC	oi	Exact balanced-fairness normalizing constant for closed order-independent (OI) networks, with the OI functional-server identity for the mean queue lengths; auto-selected under `default`/`exact` and reported as the actual method	–
NC	nrl	Norlund-Rice integral with logit transformation	[29]
NC	nrp	Norlund-Rice integral with probit transformation	[29]
NC	nre	Norlund-Rice integral on a saddle-tilted contour, with a second-order Edgeworth correction	[29], [50], [10]
NC	sdr	Product form of a closed multiclass network with state-dependent routing, evaluated exactly by state enumeration (eq. 16); any branch topology	[90]
NC	sdr.mva	Same model solved by the Section 4 mean value analysis and convolution instead of by enumeration; single-center branches only	[90]
NC	morrison	Heavy-usage asymptotic expansion of the generating function of a closed think+DPS network: one infinite-server station and one single-server discriminatory-processor-sharing station, exponential service, populations $K_j = Nb_j$ and usage $\rho = 1 - a/\sqrt{N}$. Non-product-form, so no normalizing constant is returned; the default (and only admissible method) on that shape	[113]
NC	pana	Panacea normal-usage asymptotic expansion of the normalizing constant	[107], [130]
NC	panald	Panacea asymptotic expansion, load-dependent	[111]
NC	rd	Reduction heuristic	[29]
NC	sampling	Automated selection of sampling method	–
NC	mem	Maximum Entropy Method for open ($GE/GE/1$, $GE/GE/c$, $GE/GE/\infty$ building blocks), closed (pseudo-open decomposition plus convolution) and mixed queueing networks; auto-selected by `default` for non-Markovian open models	[89]

Continued on next page

Table 8.1 – Solution methods for `Network` solvers. *Continued from previous page*

Solver	Method	Description	Refs.
NC	lossn.exact	Open loss network in a DROP finite capacity region: exact normalizing constant of the product form over the integer state space $\mathbf{An} \leq \mathbf{C}$ by contour integration and residues, with the region's job, memory and linear admission rows as constraints; the loss-network default when $\mathbf{A}, \mathbf{C}$ are integer	[104]
NC	lossn.erlangfp	Same models, Erlang fixed-point (reduced-load) approximation; the loss-network default for fractional class sizes or capacities	[83, 84, 131]
NC	lossn.mci	Same models, importance-sampling Monte Carlo summation of the normalizing constant with confidence intervals; `samples` and `seed` options	[133]
NC	mem.blocking	Finite station buffers in a single-class open network: censored $GE/GE/c/0$; N stations under the loss rule, plus the holding-node expansion of transfer blocking (BAS); explicit request only	[89, 148]
SSA	default	Alias for `nrm` if the model supports it, otherwise `serial`.	–
SSA	nrm	Next reaction method for population models (e.g., PS/INF scheduling).	[3]
SSA	serial	CTMC stochastic simulation on a single core	[68]
SSA	para	Parallel simulations (independent replicas)	–

8.1.3 Warm start

`init_from_solver` is available on every network solver and is not specific to simulation: an auxiliary solver supplies a steady-state job placement that becomes the initial state of the model, stored in `options.init_sol` as the station-major vector $[n_{1,1}, \ldots, n_{1,K}, \ldots, n_{M,K}]$. `FLD` starts the ODE integration from it, `SSA` and `LDES` start the simulated trajectory from it, `JMT` preloads the stations with it, and the AMVA submethods of `MVA` use it as the starting iterate in place of the uniform N/M guess. Because the placement is written into the model object, it is visible to any other solver instance constructed on the same model. The same mechanism is used by `SolverLN`, which forwards each layer's previous-iteration solution, and is also reachable by setting `options.init_sol` by hand.

With `CTMC` as the auxiliary solver the initial state is the mode of the exact stationary distribution over the aggregate state space; with any other solver the steady-state mean queue lengths are rounded to an integer placement that conserves each closed-class population. Under `LDES` the simulation then starts approximately in steady state, so warmup removal is disabled (`tranfilter` set to `fixed` with `warmupfrac` $= 0$) and every sample contributes to the estimators (see `examples/advanced/initState/ldes_warmstart.py`).

8.1.4 Where the algorithms are documented

Each solver of this chapter has a chapter of the same name in the internals manual, which gives its solution paradigm, its analyzers and handlers, a sequence diagram of a solution request, and the algorithm and reference behind every method name. The solvers documented in other chapters of this manual, namely `AUTO` (§1.3), `ENV` and `LN`, have internals chapters as well. When this manual and the internals manual disagree, the user-facing statements (option names, supported features) are authoritative here, and the algorithmic ones there.

8.2 Native solvers

The solvers described in this section are implemented inside LINE itself. Each one analyzes the model with its own algorithms, in the codebase the model was written in, and requires no external tool: what they need is the runtime of that codebase and nothing else. They are presented in alphabetical order.

8.2.1 AG: agent-based solver

`AG` is LINE's *agent-based* solver. It views the network as a collection of interacting agents rather than as a single stochastic process: every (station, class) pair becomes an isolated *agent*, solved exactly on its own state space, and the agents are coupled only through a small vector of scalars, one per synchronizing action, over which the analysis iterates to a fixed point. That vector is the entire interface between agents, so an agent is analyzed without ever forming the joint state space, and the cost of the analysis grows with the size of the largest agent rather than with the size of the network.

The decomposition is of the *model* rather than of the traffic: where `MAM` fits an arrival stream to each station and solves the station against it, `AG` solves each agent exactly and lets the agents meet only in those scalars. `AG` is also the only solver in LINE that reads the G-network signal marking, so a network with signals (§6.4.1) must be solved with it.

8.2.1.1 Methods and configuration options

The methods available in this release all realise the agent decomposition through the Reversed Compound Agent Theorem (RCAT), under which the coupling scalars are the reversed rates of the synchronizing actions: agent k carries the generator

$$Q_k(\mathbf{x}) \;=\; L_k \;+\; \sum_{c\,:\,\mathrm{passive}(c)=k} x_c\, P_c^{(b)},$$

and publishes, for every action it is active on, one such rate read off its own stationary vector. Other agent decompositions may be added in later releases; the interface described here – the option names, the truncation control and the execution backends below – is that of the solver and not of any one method.

- `options.method='inap'` (INAP, the default): Iterative Numerical Approximation Procedure [28]. Each agent is a quasi-birth-death process [115] whose level is the queue length and whose phase is the pair (arrival phase, service phase), so a phase-type arrival or service law is represented exactly rather than by its mean rate.
- `options.method='inapplus'` (Improved INAP) [28]: the reversed rate is estimated by rate conservation instead of by the mean of the state-wise ratios; recommended when `inap` converges slowly.
- `options.method='inapinf'` (Matrix-Geometric INAP) [105]: solves each open agent on its infinite state space through its geometric tail, avoiding the truncation of `maxStates`. Preferred for open signal networks with birth-death or catastrophe structure.

`options.config.maxStates` (default `100`) truncates the queue-length dimension of an *open* agent; a closed class uses its own population instead, and `inapinf` ignores the setting entirely.

Execution backends. Because the coupling between agents is one scalar per action, the sweep parallelises exactly rather than approximately. `options.config.exec` selects who evaluates an agent, which never changes what the agent evaluates to:

- `'serial'` (default): one agent after another, on the caller's thread.
- `'threads'`: the agents of a sweep are fanned out over a local pool of `options.config.nworkers` workers (0 selects one per processor). The sweep is a Jacobi iteration, so the agent order is immaterial and this backend is *bit-identical* to `serial`.
- `'cluster'`: the agents are partitioned over `ag-worker` processes listed in `options.config.endpoints` as `'host:port'`, which may be on other machines. Each worker receives the static half of its agents once and then exchanges only the reversed rates, one double per action, per sweep.

A worker is started with

```
java -cp jline.jar jline.cli.AgWorker -p 5870
```

A worker that is unreachable, slow or lost mid-run is not fatal: its agents are solved on the coordinator instead, because any agent can be solved anywhere given the reversed rates, so a distributed run degrades to a slower run and never to a wrong one. `options.config.worker_timeout` (default `30` s) bounds the wait on one.

8.2.2 BA: bound analysis solver

The `BA` class (alias: `SolverBA`) is the *bound analysis* solver: it computes performance *bounds* for queueing networks. It differs from the other solvers in what it returns: rather than a point estimate, each method returns one guaranteed side of an interval containing the exact solution, so that the `.lower` and `.upper` members of a family bracket the true value. This is the property that makes bounds useful inside optimization loops and admission tests, where an approximation of unknown sign cannot be used safely. Because every bound is a function of the service demands, think times and populations alone, and never of the service distribution beyond its mean, `BA` is inexpensive. Most families require a closed product-form-parameterized model; the `bgt`/`bpt` and `snc` families instead cover open networks (below), so the feature set actually admitted depends on which family is requested. One family, `snc.upper`, returns a *tail quantile* rather than a bound on a mean, and is described at the end of this section.

8.2.2.1 Methods and configuration options

The available families are listed in Table 8.1: the asymptotic families, the bound hierarchies parameterized by `options.level`, the reduction bounds `lr` and `qr`/`qrf.*`, the single-class bounds `scb.*`, the achievable-region bounds `bgt.upper`/`bpt.lower`, the stochastic network calculus bound `snc.upper`, the

MAP bounds `mapamva.*`, and the Petri-net moment-relaxation bounds `spnlp.*`. Their algorithms and bibliographic sources are given in the internals manual, Chapter `BA`.

Three methods are one-sided by construction: `cub`, `bgt.upper` and `snc.upper` yield an upper bound only, while `mbjb`, `ldbcmp` and `bpt.lower` yield lower bounds only. Several families carry structural restrictions beyond the feature set and reject a model rather than return a value that would not be a bound: `sb`, `sib` and `scb.*` do not admit infinite-server stations, `lr` and `scb.*` require a single-class closed model with single-server stations, and the whole `qrf.*` family requires a closed single-class model without delay stations. `mapamva.*` asks for that same shape with a MAP service law at one station, and `spnlp.*` is offered only on a model holding `Transition` nodes, where it is in turn the only family offered. The blocking variants `qrf.bas.mmi` and `qrf.bas.mem` are unavailable in native Python and raise an explicit error there. The linear programs of the reduction bounds are solved to about eleven digits in MATLAB and native Python but only about five in Java, so the three should not be compared digit for digit.

The `bgt.upper`/`bpt.lower` family, together with `snc.upper`, admits an *open* model: it requires a Source station, a single server per queueing station (no delay, no multiserver), and deterministic, non-merging per-class routing; on an open model `list_valid_methods` narrows to exactly these three method names, and `linprog`/HiGHS is required to solve the linear program underlying the first two. Both bounds are deliberately loose (by one to several orders of magnitude on small networks): the sharp content is the stability certificate and the tail-decay rate they establish, not the numerical constant, so they should not be used where a tight bracket is needed.

The stochastic network calculus family `snc.upper` answers a different question from every other family in this solver: its native output is a *tail quantile* with a certified violation probability, not a bound on a mean. Given a target violation probability, `get_delay_perc` returns, per station and class, the smallest delay whose exceedance probability is certified to stay below it, and `get_backlog_perc` the corresponding queue-length quantile in jobs; `get_perc_table` reports both in the layout of `getAvgTable`. The guarantee holds for *any* work-conserving scheduling policy at every station. The method describes each flow by a moment-generating-function envelope and each server by the counting process of its exponential service, composes them across the network with the min-plus operations (leftover service under blind multiplexing, concatenation of a tandem, departure envelope of a hop), and minimizes the resulting Chernoff bound over the envelope parameter. The underlying algebra is exposed as the `snc` API family in all four codebases.

The mean columns returned by `getAvgTable` under `snc.upper` are the integral of that tail bound, hence valid upper bounds on the mean response time, with the queue length following by Little's law from the exact open-network throughput. They are deliberately loose: on an `M/M/1` they exceed the exact mean response time by a factor of about 3.5 at 30% utilization and 8.8 at 90%, because an integral over the whole axis is dominated by the prefactor of the bound rather than by its decay rate. The decay rate itself is asymptotically exact, so the ratio of the bounded quantile to the exact one falls towards one as the violation probability is tightened. Use the quantiles, not the mean columns, when the tail is what matters. Beyond the open-model requirements above, `snc.upper` needs a feed-forward station graph, deterministic routing downstream of the Source (a probabilistic split is admitted only at a Poisson Source, where thinning is exact), exponential service, and equal service rates among the classes sharing a station; each is refused by name rather than approximated. Worked illustrations are `snc_delay_quantile` and `snc_tandem_multiclass`.

The single-class bounds `scb.upper`/`scb.lower` answer a different question from every other family: given a single-class demand vector and population, they do not bracket that model's own solution but the unknown multiclass system it aggregates. `scb.lower` is in fact the *exact* single-class throughput, not an approximation; `scb.upper` inflates it by a gap that shrinks with the multiplicity of the dominant demand and is capped by the one-server-per-station capacity constraint. Because its lower side already equals the exact answer, `scb` is excluded from the `auto.*` candidate set below, which would otherwise collapse onto it. The companion model-free functions `pfqn_scbgap`, `pfqn_usumbound` and `pfqn_minclasses` bound, respectively, the relative throughput error from merging classes, the sum of utilizations across stations, and the minimum number of classes consistent with a measured utilization sum; none requires a solved model. A worked illustration of all four, including the paper's own numeric examples, is given in `tut15_bound_analysis`.

The family `mapamva.upper`/`mapamva.lower` [33] is the only one in this solver that consumes the *correlation* between successive services rather than the service mean alone. Its linear program is written over the exact mean-value balances of a closed MAP queueing network, in the per-phase queue-length and utilization variables, so a workload whose burstiness moves the bottleneck between stations is bounded rather than averaged into a renewal process. It admits a single-class closed model with single-server stations and no delay, FCFS at the MAP station, and phase structure at one station only. A `MAP` or `MMPP2` service law reaches this family, the `qrf.*` chain and `snc.upper`, and no other.

The family `spnlp.*` [98] bounds a stochastic Petri net, and is the only `BA` family offered on one: every other family is parameterized by demands and a population, which a marking is not. It relaxes the stationary chain to a *moment polytope*, the uniformized evolution equation written for the mean marking, its second moments and the pairwise covariances, together with the behavioural, probabilistic and Little's-law families, and then minimizes and maximizes each reported measure over it. Every stationary point of the true chain satisfies every row, so the two optima bracket the exact value. `spnlp.upper` and `spnlp.lower` require exponential firing times. The operational pair `spnlp.op.upper`/`spnlp.op.lower` drops the second-moment, covariance and Little's-law families and rests on flow balance, the place invariants and the state equation alone, so it needs only a mean firing time and admits any phase-type law, at the price of a much looser bracket. A worked illustration is `spn_lpbounds`.

The composite family `auto.upper`/`auto.lower` evaluates every noniterative family that admits the model and keeps the tightest side, so its bracket is never wider than that of any single family; it requires a single-class closed model and reports an error when no candidate is feasible. The level-parameterized hierarchies are deliberately excluded from the candidates, their accuracy being set by `options.level` rather than being intrinsic. On the Eager-Sevcik hierarchy `pbh` the bracket width falls monotonically as `options.level` rises and reaches zero once the level equals the population; a worked illustration is given in `tut15_bound_analysis`.

8.2.2.2 Reading bounds

The normal way to read a bound is `get_bounds_table`, the counterpart of `avg_table`. It evaluates both sides of the family of the selected method and reports the bracket per station and class, with columns

`Qlower`, `Qupper`, `Tlower` and `Tupper`. It follows the labelling of `avg_table` and likewise omits disabled station-class pairs.

```
>>> SolverBA(model, 'gb.upper').get_bounds_table()
  Station JobClass   Qlower   Qupper   Tlower   Tupper
0   Think        C  1.13102  1.25571  0.56551  0.62785
1      Q1        C  0.70532  1.52127  0.56551  0.62785
2      Q2        C  1.50772  4.18702  0.56551  0.62785
```

For a one-sided family the absent side is reported as `NaN` rather than as zero, so that the absence of a bound stays distinguishable from a bound whose value happens to be zero. Requesting `cub`, which is upper-only, on the same model yields `NaN` throughout the `Qlower` and `Tlower` columns.

The programmatic accessor is `get_bounds`, which returns the same information as a raw bracket rather than a formatted table, and is the form to use inside a loop. The relationship between the two mirrors that between `get_avg` and `avg_table`.

```
b = SolverBA(model, 'bjb.upper').get_bounds()
b['Tlower'], b['Tupper']    # throughput bracket
b['Qlower'], b['Qupper']    # queue-length bracket
```

Both accessors inherit the option set of the solver they are called on, so `options.level` applies to the bracket as a whole.

8.2.3 CTMC: continuous-time Markov chain solver

The `CTMC` class solves the model by first generating the infinitesimal generator of the and then calling an appropriate solver. Steady-state analysis is carried out by solving the global balance equations defined by the infinitesimal generator. If the `keep` option is set to `True`, the solver will save the infinitesimal generator in a temporary file and its location will be shown to the user.

Figure 8.2 shows the two state-space structures involved.

The solver refuses by default to solve models whose state space is predicted not to fit in memory, comparing the predicted peak footprint against a fraction (by default 0.6) of the memory available on the host; the `force` option bypasses this control. In models with infinite states, such as networks with open classes, the `cutoff` option reduces the CTMC to a finite process: a scalar is the maximum number of jobs that a class can place at an arbitrary station, while a matrix gives in row i and column r the maximum number of jobs of class r that can be placed at station i. Transient analysis integrates Kolmogorov's forward equations over the interval given by `timespan`, with the ODE solver choice of `FLD`, or by uniformization with Fox-Glynn weights, exposed through the Markov-chain API as `ctmc_foxglynn`.

8.2.3.1 Methods and configuration options

The following methods are available for the `CTMC` solver:

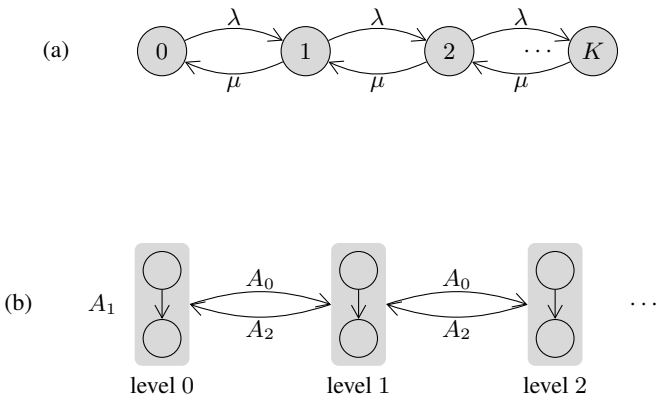


Figure 8.2: (a) The birth-death chain of an M/M/1/K queue, the kind of state space that `CTMC` enumerates. (b) The level structure of a QBD process, on which the matrix-analytic methods of `MAM` operate: A_0, A_1 and A_2 collect the transitions that raise, preserve and lower the level.

- `options.method='default'`: direct solution of the global balance equations, which also covers a reducible generator.
- `options.method='gpu'`: GPU-accelerated solution of the global balance equations.
- `options.method='cftp'` and `'cftp.approx'`: perfect sampling of the stationary distribution by coupling from the past, without generating the state space, so that the memory gate does not apply. The metrics are Monte Carlo estimates whose error decreases as `options.samples`$^{-1/2}$ (default 10^4), the exact variant having an unbounded worst-case cost per sample and the approximate one a deterministic cost with a residual bias. The model must be closed and single-class, built of `Queue`, `Delay` and `Router` nodes only, with exponential service under `INF`, `PS`, `FCFS`, `SIRO` or `LCFSPR` scheduling, infinite buffers and Markovian routing; anything else is refused rather than approximated.
- `options.method='mdd'`: decision-diagram aggregation of a closed single-class chain, whose memory cost is governed by the diagram size rather than by the state space. Every solve reports an exactness certificate, and phase-type service under `LCFSPR` or `FCFSPR` is rejected. Transient analysis, reward functionals and the symbolic export are unavailable under this method.
- `options.method='qrf.*'`: approximations from the Quadratic Reduction Framework, namely `qrf.mmi`, `qrf.mem`, `qrf.bethe`, `qrf.mmi.ld`, `qrf.mmi.linear`, `qrf.bas.mmi`, `qrf.bas.mem`, `qrf.bas.bethe`, `qrf.bas` and `qrf.rsrd`, available only for a single-class model with a finite total population. The two `bethe` variants replace the mutual-information objective by the tree-reweighted (Bethe) free entropy at $\lambda = 1/M$ [157], on the plain and on the blocking-after-service polytope respectively. The blocking variants require the blocking topology in `options.config.qrf_params` and the load-dependent ones read `options.config.qrf_alpha`; a missing mandatory field aborts the solver.

The two configuration entries read by the QRF dispatcher are:

- `options.config.qrf_params`: optional struct describing the blocking configuration of `qrf.bas.*` variants. Fields: `f` (blocking-event index), `MR` (number of resources), `BB` (per-station buffer capacities), `F` (capacity vector), `MM`, `MM1`, `ZZ`, `ZM` (auxiliary index sets needed by the BAS engine). Defaults to an empty struct.
- `options.config.qrf_alpha`: optional $M \times N$ matrix of load-dependent scaling factors used by `qrf.mmi.ld` and `qrf.mmi.linear`. Row i, column n scales the service rate of station i when n jobs are present. Defaults to a matrix of ones.

Solving an aggregated model. The state space of a multiclass model grows with the per-class populations, and the two model adaptations of §5.6 are the standard way of making an otherwise intractable model solvable. Both are reachable from the `CTMC` solver itself, as opt-in configuration entries, so that the reduction and the recovery of the original metrics happen inside a single solve:

- `options.config.chain_aggregation` (default false): collapse every chain onto a single class with `aggregateChains` (§5.6.1), solve the collapsed model, and split the answer back per class. It is applied only when the model has fewer chains than classes, and is exact on a product-form model and an approximation otherwise, because one aggregate service law replaces the per-class ones weighted by α.
- `options.config.fes_stations` (default unset): the indices of the stations to collapse into one load-dependent station with `aggregateFES` (§5.6.2). Indices are 0-based, must be distinct, must name at least two stations, and may not name every station. The surviving stations are answered by the reduced chain directly; a collapsed station is recovered by conditioning its queue length on the population held by the flow-equivalent server and averaging over that population [37], the isolated subnetwork's per-population queue lengths and utilizations being supplied by the API function `Fes_compute_metrics`, and the reduction leaves throughput unchanged. The decomposition is exact when the collapsed subnetwork is product-form.

When both entries are set, the flow-equivalent reduction is applied first. The method name reported back by `get_avg_table` records which path ran, as `default/chainaggr` or `default/fes`.

Immediate transitions introduced by `Router` nodes, for instance in a cache topology with state-dependent routing (§6.2.3), are eliminated automatically by stochastic complementation [109] before the chain is solved, so that per-class arrival and throughput metrics at downstream stations remain correct. The lower-level primitives are exposed as the API functions `Ctmc_stochcomp` and `Dtmc_stochcomp` and may be used directly on custom chains. The remaining configuration options are listed in the next table.

Table 8.2: `CTMC` configuration options (Python)

Option	Value	Description
`options.config.hide_immediate`	`bool`	If true, immediate transitions are hidden from the CTMC.
`options.config.state_space_gen`	`'reachable'`	Direct state space enumeration from initial state.
`options.config.state_space_gen`	`'full'`	Direct state space enumeration from all possible initial states.

Continued on next page

Table 8.2 – *Continued from previous page*

Option	Value	Description
`options.timestep`	`float`	Fixed time interval for transient analysis. If specified, generates equally-spaced time points instead of adaptive stepping.
`options.gen_method`	`'default'` | `'sync'`	Generator construction: `'default'` builds the generator monolithically, `'sync'` assembles it from the synchronization actions of the model. Python-native only; the MATLAB and Java engines always use the synchronization-based construction.

8.2.3.2 Generator and state space access

The CTMC solver also provides access to the underlying infinitesimal generator and its decomposition into per-event filtration matrices via the `get_generator` method:

```
solver = CTMC(model)
infGen, eventFilt = solver.get_generator()
```

Here `infGen` is the infinitesimal generator matrix and `eventFilt` is a list of sparse matrices, one per event type, encoding the transitions associated with each event (arrival, departure, phase change, etc.). The generator can also be built symbolically, with each event filtration normalized by its smallest positive rate and scaled by a symbol $x_1, \ldots, x_E$, one per event:

```
infGen, eventFilt, syncInfo, stateSpace, nodeStateSpace = ...
solver.getSymbolicGenerator()
```

No computer algebra is needed to assemble it, since the generator is linear in the symbols; substituting each symbol with its event's smallest positive rate recovers the numerical generator exactly. Solving with the symbolic generator does require a computer algebra system. The symbolic backend is described in §10.9.2.

A chain supplied directly by the user, rather than compiled from a network, is a model class of its own and is described in §6.1; the `CTMC` solver solves it as it stands, without generating a state space.

8.2.3.3 Reward analysis

The CTMC solver supports reward-based analysis through custom reward functions defined on network states. Users can define state-dependent reward functions that map a state vector and the network structure to a scalar reward value. Multiple rewards can be defined by calling `setReward` multiple times with different names. The solver computes rewards using value iteration with uniformization. Results are retrieved via `solver.getAvgReward()`, which returns steady-state expected rewards, or `solver.getTranReward()`, which returns the transient expected-reward trajectories $E[r(X(t))]$ over time. Example applications include computing expected queue lengths, utilization, blocking probabilities, or custom cost functions.

Reward functions are defined using `model.set_reward(name, reward_fn)`, where `name` is a string identifier and `reward_fn` is a callable that takes a state vector and the network structure and returns a scalar reward value.

8.2.4 FLD: fluid and mean-field solver

This solver implements fluid/mean-field approximation methods, based on the system of fluid ordinary differential equations for INF-PS queueing networks presented in [121]. The latter is based on Kurtz's mean-field approximation theory. The ODEs are integrated with SciPy's `solve_ivp`, using LSODA when `options.stiff` is set and RK45 otherwise; `options.tol` is passed as both the relative and the absolute tolerance.

ODE variables corresponding to an infinite number of jobs, as in the job pool of a source station, or to jobs in a disabled class are not included in the solution vector. These rules apply also to the `options.init_sol` vector.

Nodes with zero or near-zero service time (e.g., `Router`, `ClassSwitch`) are treated as immediate nodes and are eliminated from the fluid equations before integration; FCFS stations are mapped into equivalent PS stations with a mixture service process. Both transformations are described in the LINE internals manual (`LINE-internals-python.pdf`, Chapter `FLD`).

8.2.4.1 Methods and configuration options

The following methods are available for the `FLD` solver:

- `options.method='default'`: resolves to a concrete method from the model: `rmf` if the model contains `Cache` nodes, otherwise `minnormal` wherever that method applies, and failing that `closing` for models with `DPS` stations or `matrix` for the rest.
- `options.method='matrix'`: matrix formulation of the fluid ODEs [134], for closed and open networks.
- `options.method='closing'` or `'statedep'`: state-dependent fluid solver with closing approximations, useful for models with DPS scheduling.
- `options.method='pnorm'`: p-norm smoothing of the processor-share constraint [134].
- `options.method='diffusion'`: diffusion approximation, restricted to product-form closed networks.
- `options.method='mfq'`: exact steady-state analysis of a Source $\rightarrow$ Queue $\rightarrow$ Sink model with phase-type arrivals and service [76]; closed networks and multiple queues are not supported.
- `options.method='rmf'`: refined mean field for multi-list caches with RANDOM(m) replacement [65], in steady state and transient; selected by `default` when the model contains `Cache` nodes.
- `options.method='tbi'`: cell-decomposed solution of the closing ODEs for large closed models [142], the cells being reconciled by waveform relaxation [94]. Configured by `options.config.tbi_tol` (default 10^{-3}), `tbi_iter_max` (default 50), `tbi_sweep` (`'gauss-seidel'` or `'jacobi'`),

`tbi_parallel` (MATLAB only), and by the cell partition, given as `tbi_cells` or built automatically with `tbi_cellsize` stations per cell (default 5). Caching stations and immediate-node elimination are not supported.

- `options.method='minnormal'`: second-order moment closure [69, 145, 154], which keeps the error bounded as $\rho \to 1$ where the first-order closures do not. It is the only LINE fluid method supporting `GPS`. Open and mixed models are admitted when the arrival process is Poisson, a non-exponential inter-arrival time being refused; throughputs are then exact and the error falls on the mean queue lengths. The state-level covariance and the queue-length variances are reported by the `moments` dictionary of the result, and are reliable for closed classes only.
- `options.method='refined'`: refined mean field [64], adding the $O(1/N)$ term to the mean-field fixed point. Closed models only; it is not applied on top of `minnormal`, which already resums the same term. Where the drift is exactly affine on the reachable set – every station either an infinite server or one whose population bound cannot reach its server count, so that $\min(n, c)$ is the identity there – the correction is identically zero and the method returns the mean-field answer unchanged. Available in MATLAB, native Python and C++; the JAR implements `minnormal` only.
- `options.method='kp'`: fluid and diffusion limits of the open $(\text{MAP}_t/\text{Ph}_t/\infty)^N$ network of [87], i.e. of a network whose arrival and service processes are the time-inhomogeneous `MAPt`/`PHt` distributions. It integrates the mean and the covariance jointly, the second moment being exposed through `getTranAvgVar`, which is available under this method only. Closed classes are rejected.
- `options.method='dae'`: the same min-normal closure as `minnormal` – the same drift, the same rate factors, the same Lyapunov equation – stated as a single differential-algebraic system and solved as one, instead of by the outer substitution loop that alternates an ODE integration with a Lyapunov solve. See §8.2.4.2 below.

The algorithms behind these methods are documented in the internals manual, Chapter `FLD`.

FLD method selection. The default FLD method (`matrix`) formulates a set of ordinary differential equations (ODEs) that describe the mean-field dynamics of the network. The `statedep` method uses Kurtz's mean-field theorem, which is exact for state-dependent models in the large population limit. The `softmin` variant replaces the *min* function with a smooth approximation, improving numerical stability of the ODE integration at the cost of a small approximation error. The `mfq` method solves Markovian fluid queues and is applicable to single-queue systems with MAP arrivals. The `diffusion` method applies Euler-Maruyama discretization to model stochastic fluctuations around the mean field. The `rmf` method uses the refined mean field approximation [65] to analyze multi-list caches with RANDOM(m) replacement, providing $O(1/N)$ corrections to mean field fixed points via Lyapunov equations; it is automatically selected when the model contains Cache nodes.

8.2.4.2 The differential-algebraic formulation

The `minnormal` closure is already a differential system (the mean) coupled to an algebraic one (the covariance), and it is discharged by *successive substitution*: the mean ODE is integrated to its fixed point at a held variance, the Lyapunov equation is solved there, the variance is extracted and the cycle repeats. The `dae` method states the same closure as one system and solves it as one. Nothing about the closure changes – the drift, the rate factors and the Lyapunov equation are the same functions – only the way the coupled equations are discharged. The horizon decides which of two genuinely different problems is posed.

Steady state (an infinite `timespan`). One algebraic system collects three families of rows – the drift residual, one row per state coordinate; population conservation, one row per closed chain; and closure consistency, one row per closable station – and is solved simultaneously by a damped Newton iteration. Convergence is quadratic near the root instead of the linear rate of substitution, and the answer is converged to `options.tol` rather than to the coarse tolerance of the outer loop. The fixed point is *not* found by integrating to it: a seed trajectory is integrated once, cheaply and at the first-order closure, because Newton needs a starting point inside the basin, and everything after that is algebraic. This matters most on stiff models, where the cost of integrating to a standstill is set by the slowest mode of the model rather than by the accuracy wanted. Writing conservation as a constraint rather than leaving it a consequence of the drift is also what makes the Newton system solvable: the drift Jacobian is singular along exactly the conserved directions, and the constraint rows supply the missing rank. The covariance itself is not a Newton unknown – it is linear in itself for a held mean, so it is eliminated by one Lyapunov solve per residual evaluation and only the per-station variances join the mean in the unknown vector, which keeps the work cubic rather than quartic.

Transient (a finite `timespan`). The index-1 system that evolves the mean under the drift, holds the conservation constraint as an algebraic row and evolves the covariance by its Lyapunov equation, with a singular mass matrix, is integrated by RODAS, the Rosenbrock method of order (3)4 for differential-algebraic systems [70]. Together with `kp` this is the only fluid method producing a *time-varying* second moment: `minnormal` evaluates the whole transient at the single stationary variance, whereas here the variance is the one the trajectory actually had at each instant. It is reported alongside the transient means.

Capacity limits as algebraic constraints. A finite capacity region (§5.5) and a station buffer are both a linear inequality on the fluid state, which this formulation can carry beside the drift and no ODE method can carry at all. All the declared forms reduce to the same family of weighted-sum inequalities on the fluid state: a region-global job cap is the row with unit weights, a per-class cap weights one class, a memory budget weights each class by its size, and an explicit linear constraint supplies the weights directly; a station cap is the same row restricted to one station. What differs between them is not the row but where a blocked job goes. Each capped *region* therefore gains one waiting-room coordinate per class, and every admission into it is split into an unthrottled leg from the upstream station into the room and a throttled leg from the room into the region, so that a job blocked by the cap has completed its upstream service and waits somewhere else rather than piling up at a station it has already left. A *station* buffer gets no room, matching the reference semantics

in which the upstream departure is disabled and the blocked job is still counted where it was. Regions are accepted under the waiting-queue rule only; station buffers under either the waiting-queue or the drop rule. Blocking-after-service, blocking-before-service, RS-RD and the retrial rules each change the event set itself rather than constrain this drift, and are refused by name, as are per-class admission weights, which decide *which* job is admitted rather than how many.

Availability and limits. The method is present in all four codebases. It admits limited load dependence, like the rest of the closing family, and finite capacity regions, unlike every other fluid method; it rejects `SIRO`, `LCFS` and `LCFSPR`, which have no branch in the closing drift, and `DPS` and `GPS`, whose closures need covariance blocks between the class coordinates of a station that the DAE form carries no unknown for. Two size gates guard the cost: `options.config.dae_maxstate` (default 100) caps the phase-resolved state of the simultaneous solve and `options.config.dae_maxcov` (default 25) the transient covariance.

```
solver = FLD(model, 'dae')
avg_table = solver.getAvgTable()
```

8.2.4.3 Exporting the ODE system

The system of ODEs integrated by the mean-field methods (`default`/`matrix`, `pnorm`, `closing`, `statedep`, `softmin`) can be exported in symbolic form as a standalone LaTeX document using the `exportODEs` method. The export is both human readable, since the compiled document shows the equations together with a legend of the state variables, and machine readable, since the document header carries the state-variable mapping and event rates as structured comment lines (`% STATE` and `% EVENT`). Each state variable x_s represents the mean number of jobs of a class in a service phase at a station. Two notations are available:

- `'scalar'` (default): one expanded ODE per state variable, written in terms of station-level auxiliary functions such as the total station mass $n_i(x)$ and the capacity scaling $g_i(x) = \min(n_i(x), S_i)/n_i(x)$;
- `'matrix'`: the compact matrix form, $d\mathbf{x}/dt = W^\top \theta(\mathbf{x}) + \boldsymbol{\lambda}$ for the `matrix`/`pnorm` methods (per Ruuskanen et al. [134], with θ the smoothed service vector and $\boldsymbol{\lambda}$ the exogenous arrival rates), and $d\mathbf{x}/dt = J\, r(\mathbf{x})$ for the `closing`/`statedep`/`softmin` methods, where J is the stoichiometry (jump) matrix and $r(\mathbf{x})$ the vector of event rate functions.

The exported document also reports the initial condition $\mathbf{x}(0)$ and remarks on the numerical regularizations applied by the solver. The exported system reflects the method set in the solver options; methods that are not ODE-based (`diffusion`, `mfq`, `rmf`) are not supported by this export.

```
solver = FLD(model, 'closing')
tex = solver.exportODEs('model_odes.tex', 'scalar')
```

Passing an empty file name returns the LaTeX source without writing a file. The method is also available under the alias `export_odes`.

8.2.5 LDES: discrete-event simulation engine

The `LDES` solver is LINE's own native discrete-event simulation engine, built on the SSJ library and shipped inside the canonical JAR. All three codebases drive the same engine, so a given model and seed produce identical sample paths and hence identical estimates. Python-native drives it as a fully JSON-mediated subprocess: `save_model` writes `model.json`, the engine is invoked as `solve model.json -o result.json`, and the resulting JSON is parsed back. The engine is the native `common/ldes` binary, falling back to `java -jar common/ldes.jar`.

The model features admitted by the solver are those declared by its feature set (`SolverLDES.getFeatureSet()`), summarized in the tables of Appendix B. Finite-buffer stations are honoured on arrival according to the per-station `DropStrategy`, as are network-level `FiniteCapacityRegion` constraints. A `BMAP` assigned as a service process is interpreted as a batch service process (§5.3.6). The accuracy of the estimates is governed by the simulation length and by the transient filter and confidence-interval estimators of Table 8.3; the solver is validated against `JMT` by a two-sample t-test at the 1% significance level. Its initial state can be seeded from another solver as described in §8.1.3.

8.2.5.1 Methods and configuration options

Table 8.3 lists the configuration options. Defaults are those of the engine and are shared by all codebases.

Table 8.3: `LDES` configuration options

Option	Default	Description
`options.samples`	$2 \cdot 10^5$	Service-completion events to simulate (engine flag `-s`).
`options.events`	`None`	Explicit service-completion budget; when set it overrides `samples`.
`options.seed`	23000	Random number generator seed (`--seed`).
`options.method`	`'default'`	Simulation method; only `default` is defined.
`options.timeout`	`inf`	Wall-clock budget in seconds (`--maxtime`).
`options.tranfilter`	`mser5`	Warmup detection: `mser5` (MSER-5), `fixed` (fixed fraction), `none`.
`options.mserbatch`	5	MSER batch size; used only when `tranfilter` is `mser5`.
`options.warmupfrac`	0.2	Fraction discarded; used only when `tranfilter` is `fixed`.
`options.cimethod`	`obm`	Estimator: `obm` (overlapping batch means), `bm` (batch means), `spectral` (Heidelberger-Welch), `none`.
`options.obmoverlap`	0.5	Overlap fraction of `obm`; 0 reduces it to `bm`.
`options.ciminbatch`	10	Minimum batch size; the batch size used is $\max(\texttt{ciminbatch}, \sqrt{n})$ on n observations.
`options.ciminobs`	100	Minimum post-warmup observations below which no interval is reported.
`options.spectral_low_freq_frac`	0.25	Fraction of lowest frequencies in the log-periodogram regression; used only when `cimethod` is `spectral`.
`options.cnvgon`	`false`	Stop as soon as the relative precision of every metric falls below `cnvgtol`.
`options.cnvgtol`	0.05	Target relative precision (CI half-width over mean).
`options.cnvgbatch`	20	Minimum batches before the first convergence check.

Continued on next page

Table 8.3 – *Continued from previous page*

Option	Default	Description
`options.cnvgchk`	0 (auto)	Events between checks; 0 selects `samples/50`.
`options.slotted`	`false`	Advance time on a discrete lattice instead of continuously.
`options.slot_length`	1.0	Slot length in model time units; meaningful only when `slotted` is set.
`options.replications`	1	Independent replications, each with its own RNG stream; intervals then use the cross-replication variance.
`options.numthreads`	$\lceil \text{cores}/2 \rceil$	Threads used when `replications` > 1.
`options.init_sol`	empty	Warm-start placement, station-major vector (`--initsol`); set by `initFromSolver`.
`options.busy_period_orders`	0	Highest busy period order measured (`--busyperiod`); 0 disables the measurement.
`options.busy_period_subnets`	empty	Station sets whose joint busy period is measured (`--busyperiod-subnet`).
`options.rest_url`	empty	Base URL of an LDES REST server; when set the model is POSTed to `<url>/api/v1/solve` instead of being solved locally.

Setting `options.slotted` switches the engine from its default continuous time scale to a discrete (slotted) one, in which time advances in integer multiples of `options.slot_length`. Every sampled interarrival and service time must then fall on the slot lattice, a sample that does not lie on it being reported as an error rather than rounded; within a slot, simultaneous events are ordered by a fixed phase (service completions, internal movement, external arrivals, bookkeeping); and non-homogeneous Poisson arrivals and the processor-sharing family are rejected. The canonical discrete-time model is the Geo/Geo/1 queue, a `Source` with `Geometric` interarrivals feeding an `FCFS Queue` with `Geometric` service and a slot length of one, whose analytical counterpart is `qsys_geogeo1`; with a batch source (§5.3.6) the stream is Geo^X and the references are `qsys_geoxgeo1` and `qsys_geoxgeo1_moments`.

8.2.5.2 Busy period measurement

Setting `options.busy_period_orders` to K makes the engine measure, along the simulated sample path, the mean busy periods of orders $1, \ldots, K$ of every station, of every station-class pair, and of every station set declared through `options.busy_period_subnets`. A busy period of order n runs from the instant an arrival raises the jobs held by the target to n up to the instant the target falls back below n; the per-class variant counts only the jobs of one class. Periods still in progress when the warmup ends are discarded, their start not being observable, and a period of zero length is treated as the artifact of two simultaneous events rather than as an observation. The measurement is off by default and costs nothing when off.

```
b, count = LDES(model, samples=2000000, seed=23000).getAvgBusyPeriod([0, 1], -1, [1, 2, 3])
```

The exact counterpart on product-form models is §8.2.8.2, against which the measurement agrees to within the simulation error.

8.2.5.3 Command-line interface

The `LDES` engine is also distributed as a standalone executable, either as the native binary `common/ldes` or as `common/ldes.jar`. The MATLAB and Python wrappers are thin JSON-mediated clients of this executable: they serialize the model to `model.json`, invoke `solve model.json -o result.json`, and parse the returned metrics. The same interface can be driven directly. The principal flags are summarized below.

Table 8.4: `LDES` command-line flags

Flag	Description
`--samples`	Number of simulation events to execute.
`--maxevents`	Hard cap on the number of simulated events.
`--maxtime`	Hard cap on simulated time.
`--seed`	Random number generator seed.
`--method`	Simulation algorithm selector.
`--cnvgon`, `--cnvgtol`	Enable convergence-based stopping and set its relative tolerance.
`--tranfilter`	Transient (warmup) filter: `mser5`, `fixed` or `none`.
`--warmupfrac`	Warmup fraction discarded under the `fixed` filter.
`--cimethod`	Confidence interval estimator.
`--replications`	Number of independent replications; drives the parallel analyzer.
`--numthreads`	Number of worker threads used for replications.
`--slotted`	Run on a discrete (slotted) time scale.
`--slotlength`	Slot length in model time units; implies `--slotted`.
`--timespan`	Time span for transient analysis.
`--initsol`	Warm-start initial state, as a station-major population vector.
`--trajectory`	Export the simulated sample path.
`--export-histogram`	Export the state histogram, which backs reward, `getProb` and sampling queries.
`--busyperiod`	Measure the busy periods of orders $1..K$ for every station and every station-class pair.
`--busyperiod-subnet`	Also measure the joint busy period of the given station set; repeatable.

8.2.6 MAM: matrix-analytic methods solver

This is a basic solver for some Markovian open queueing systems that can be analyzed using matrix analytic methods. The core solver is based on the BU tools library for matrix-analytic methods [76]. The solution of open queueuing networks is based on traffic decomposition methods that compute the arrival process at each queue resulting from the superposition of multiple source streams.

The `getCdfRespT` method computes the response time distribution using matrix-analytic techniques. For processor-sharing (PS) queues with MAP arrivals, it uses the algorithm of [106]. The number of CDF points computed can be controlled via `options.config.num_cdf_pts` (default: 200).

8.2.6.1 Methods and configuration options

The following methods are available for the `MAM` solver:

- `options.method='default'` or `'dec.source'` (Arrival Stream Decomposition): the default decomposition, applicable to open, closed, and mixed queueing networks.

- `options.method='mna'` (Matrix Network Analyzer) [95]: more accurate than `dec.source`; recommended for networks with Markovian arrival processes, and restricted to FCFS and HOL scheduling. A departure stream split deterministically by round-robin scheduling, either at the station itself or at a downstream `Router` it feeds exclusively, is recognised as such and has its variance propagated by the exact round-robin formula rather than by the Bernoulli-thinning formula used for a probabilistic split.
- `options.method='dec.poisson'` (Poisson Arrival Approximation): the fastest but least accurate decomposition, suitable for exploratory analysis or as a baseline.
- `options.method='dec.mmap'` (ETAQA Departure Decomposition) [129]: computes MAP approximations of the departure processes; when the required state space exceeds `space_max` or the queue is unstable, the solver falls back to a scaled service process.
- `options.method='bgchain'` (Background Modulating Chain): for *mixed* and *closed* networks, see below.

The algorithms behind these methods are documented in the internals manual, Chapter `MAM`.
The agent-based methods (`inap`, `inapplus`, `inapinf`) are *not* `MAM` methods: they belong to the `AG` solver of §8.2.1. `MAM` refuses them by name and says so.

The `dec.mmap` method exposes additional configuration parameters via `options.config`:

- `etaqa_trunc` (default: 8): QBD truncation level for the ETAQA computation; higher values yield more accurate departure processes at increased computational cost.
- `space_max` (default: 128): maximum state-space size $(\texttt{etaqa\_trunc} + 1) \times n_a \times n_s$ beyond which ETAQA is skipped and a scaled service process is used instead.
- `merge` (default: `'super'`): strategy for merging multiple marked MAP streams via superposition.
- `compress` (default: `'mixture.order1'`): compression method to reduce the order of superposed MMAPs before downstream analysis.

For models containing fork-join sections, `options.config.fj_accuracy` sets the numeric accuracy parameter of the fork-join approximation (default `100`) and `options.config.fj_tmode` selects how the fork-join T matrix is computed, either `'NARE'` (default) or `'Sylves'`.

When an FCFS station declares a finite capacity K (§5.1.1), `MAM` analyses it as a loss system and reports the blocking probability as a reduction of the class throughputs, in preference to the infinite-buffer closed forms for PH/M/1, D/M/c and MAP/D/c stations, which are defined for $K = \infty$ only. An M/M/c/K station is solved exactly; a single-server station with marked arrivals is handled exactly in the MATLAB implementation only; in the remaining cases the QBD of the infinite-buffer station is truncated at level K and renormalized, which yields one blocking probability shared by all classes.

MAM method selection. The MAM solver offers several decomposition approaches for networks with non-exponential service. The `dec.mna` method uses the MNA decomposition, which is generally the most accurate for networks with Markovian arrival processes. The `dec.source` method approximates arrivals at each station using the distribution observed at the source, while `dec.poisson` replaces all arrival flows

with Poisson processes. The `inapplus` method provides an improved iterative approximation that avoids the normalization step of the standard `inap` method, and is recommended for models where `inap` converges slowly. The `inapinf` method [105] solves each isolated open component directly on its infinite state space by a matrix-geometric decomposition, yielding the exact marginal without state-space truncation – the scalar geometric with a single phase per level, and Neuts' rate matrix R [115] above it; it is preferred over `inap` for open G-networks with birth-death or catastrophe structure, and for heavy-tailed phase-type service, where truncation would otherwise bias the queue-length estimates. These methods accept any process with a Markovian (D_0, D_1) representation, and refuse the laws that have none (matrix-exponential and rational arrival processes), those that are time-inhomogeneous, discrete-time, or batch; they remain single-server only.

8.2.6.2 Background modulating chain for mixed and closed networks

The `bgchain` method applies to *mixed* and *closed* networks: the closed population vector of such a network is the sole part of the model whose state space is bounded, so it is a finite continuous-time Markov chain in its own right. The method solves that chain exactly, given the mean open occupancy, and hands each open station a station-local Markovian environment read off it, so the station becomes a level-dependent QBD whose phase carries the number of closed jobs competing for its server. The two halves meet at a fixed point on the mean open occupancy: the background chain is built from it, and the QBDs recompute it. A purely open model is refused by name, having no closed population vector to build the chain from, as are class priorities, which the per-station phase-type mixture cannot express, and fork-join sections, which do not conserve the closed population per station; `dec.source` is the fallback in all three cases.

A purely closed network is the degenerate case of the same construction rather than a separate algorithm. With no open class the fixed point has nothing to iterate: the capacity share the closed jobs hold never leaves $\min(e, c)$, which is exactly what they hold when no open work competes for the server, so the background chain alone answers, and it answers with the exact closed continuous-time Markov chain at chain granularity. On a product-form closed network it therefore reproduces exact MVA to machine precision, which is why it is what `default` resolves to on a closed model, ahead of `mna`. Two conditions gate that choice. The chain must fit within `bgstates_max`, since it enumerates the closed population vector over the stations the closed classes visit and so grows as $\binom{N+M-1}{M-1}$ per class; and the background chain reads only the *mean* service time, which is exact at a processor-sharing or infinite-server station by insensitivity and under any discipline when the law is exponential, but a first-moment surrogate otherwise. A non-exponential law at an FCFS station therefore defaults to `mna`, which carries the phase-type representation instead. Requesting `bgchain` by name overrides neither gate: the model is answered with the approximation stated.

With $R > 1$ closed chains the background chain would be exponential in R, so the method keeps one chain free at a time: the tagged chain is carried exactly and the remaining $R - 1$ are replaced by flow-equivalent aggregate classes whose population is their total and whose service times and routing are their throughput-weighted means [37]. Every chain takes its turn as the tagged one, reads its own metrics off the chain in which it is exact, and the open-class results are averaged over the passes. Which chains share an aggregate is decided by the similarity of their per-station service demands, so the aggregation stays where it is harmless.

At a processor-sharing station the open service law is replaced by the exponential of the same mean before the QBD is built. This is not a shortcut but the insensitivity property of PS: the QBD tracks one service phase for the whole station, so carrying a phase-type law there would make the open queue length inherit the coefficient-of-variation sensitivity of an M/PH/1 FCFS queue, which was measured 21% high on a HyperExp of squared coefficient of variation 4 where the exact answer is the exponential one to five digits. Measured against `CTMC` and exact MVA on mixed models of two to four stations, PS and INF stations agree to four or five significant digits under any service law, any number of servers and Poisson or MAP arrivals, where `dec.source` is 10 to 24% out; FCFS stations with class-independent rates agree to five digits, while class-dependent FCFS rates keep the closed queue lengths within about 1% and read the open queue length 14 to 20% low.

The method is controlled through `options.config`:

- `bgaggr` (default 1): the number G of aggregate classes, so that the background chain carries $1 + G$ classes. $G = 1$ is the classic tagged/aggregate pair and the cheapest option; $1 < G < R - 1$ splits the untagged chains into G groups; $G \geq R - 1$ aggregates nothing, which makes the background chain exact in the closed classes and lets a single pass answer for all of them.
- `bgstates_max` (default 20000): the cap on the number of background chain states, which is the product over the $1 + G$ classes of $\binom{N_b+M_c-1}{M_c-1}$. Exceeding it is an error naming both remedies, a smaller `bgaggr` or a larger cap.
- `qbdphases_max` (default 500): the cap on the number of environment phases carried by a station QBD.
- `bgenv` (default `'lump'`, MATLAB only): the environment axis handed to a station. The default lumps the background chain onto the closed occupancy of that station; `'full'` passes the whole chain instead, with no lumping, so its phase count grows as the chain does.

8.2.6.3 Level-dependent QBDs

The `MAM` solver embeds an exact engine for Level-Dependent Quasi-Birth-Death processes [124], whose truncation level is chosen automatically from the traffic intensity and whose convergence is controlled by `options.tol` and `options.iter_max`. It is invoked either explicitly,

```
solver = MAM(model, method='ldqbd')
avg_table = solver.getAvgTable()
```

or automatically. The explicit form is restricted to single-class networks with a single FCFS queue, in two regimes: a closed one, with an infinite-server delay station and a finite population, and an open one, with a `Source` feeding the queue and a level index truncated at `options.cutoff`. Multi-server and load-dependent stations are supported, and phase-type service is handled through matrix blocks. Multi-server phase-type service is solved *exactly*: the coordinate carried inside a level is then the multiset of the phases occupied by the $\min(n, c)$ busy servers, the same configuration space as the $MAP/PH/c$ queue of §7.4.4 and of order $\binom{\min(n,c)+m_s-1}{m_s-1}$ rather than $m_s^{\min(n,c)}$ [5], so level sizes grow over the boundary levels and repeat above them. A load-dependent factor scales the station's total service rate, shared over the busy

servers. The automatic invocation happens on retrial topologies (§5.3.5), where the orbit retrial rate makes the generator level-dependent, and no method selector is required; the metrics then include the mean orbit length, the throughput, the server utilisation and the orbit blocking probability. A transient counterpart integrates Kolmogorov's forward equation on the level-truncated generator and is invoked automatically when `get_tran_avg` is called on such a model.

8.2.7 MVA: mean value analysis solver

The solver offers approximate mean value analysis (AMVA) (`options.method='default'`), but also exact MVA algorithms (`options.method='exact'`). The default AMVA solver is based on Linearizer [39], unless there are two or less jobs in total within closed classes, in which case the solver runs the Bard-Schweitzer algorithm [138].

8.2.7.1 Methods and configuration options

Beyond the `default`/`exact` entry points, the solver exposes a catalogue of AMVA variants reported by `list_valid_methods`; the `amva.`-prefixed names are aliases of the corresponding short names. The most commonly used variants are:

- `amva.lin` (alias `lin`): Linearizer [39], the default AMVA for closed multi-class product-form models.
- `amva.bs` (alias `bs`): Bard-Schweitzer approximate MVA [138], automatically selected when the closed populations are small.
- `amva.qd` (alias `qd`) and `amva.qdlin` (alias `qdlin`): queue-dependent AMVA [32] and queue-dependent Linearizer, for limited load-dependent stations; `qd` is the engine behind `default` when load dependence is detected.
- `amva.qli` and `amva.fli`: Wang-Sevcik queue-line and fraction-line approximations [158].
- `amva.aql` (alias `aql`): aggregate queue-length approximation [166], which carries $R + 1$ population points and a correction term that removes the Schweitzer proportionality error. Restricted to strict product-form, load-independent, single-server models; it is listed only when the model qualifies.
- `amva.qsa` (alias `qsa`): the queue-shift approximation [139]. Where Linearizer extrapolates the *fractional* deviation of a per-class queue length, QSA extrapolates the *absolute* shift of the aggregate queue length seen when one customer is removed, so the unknowns are one per station rather than one per station-class. The core equation is imposed at the full population and at every population with one or two customers removed, the two-removal case through an affine extrapolation of the one-removal shifts, and the resulting system is solved as a whole by a damped Newton iteration. On the three stress networks published with the method it is between two and five times more accurate than Linearizer, and its advantage grows with the population. Because the shift is carried per station rather than per station-class, the gain holds when the service time at a queueing centre is the same for all classes, which is the regime the method was designed for and the one FCFS requires anyway; with strongly class-dependent demands Linearizer's per-class

deviations can be the more accurate choice. Restricted, like `aql`, to strict product-form, load-independent, single-server models.

- `priomva` (alias `amva.priomva`): the preemptive-resume priority arm, offered only when a station uses `SchedStrategy.FCFSPRPRIO`. It applies the Chandy-Lakshmi approximation [38] where that paper derives it – a job in service is preempted by a higher-priority arrival – so unlike the non-preemptive `HOL` arm it carries no residual for the preempted job and scales the tagged job's own service by $1/(1-\sigma_{k-1})$ as well as the queued work. `PRIOMVA` is Bolch's name for this method in PEPSY-QNS. Single-server stations only; multiserver PRS is refused by name. Accuracy degrades with population at high utilization, comparably to the `HOL` arm; see [56].
- `amva.scat` (alias `scat`): the Neuse-Chandy SCAT approximation [114], which shares the Bard-Schweitzer correction term $\Delta_{rit} = Q_{ir}(\mathbf{N}-\mathbf{e}_s)/(N-\mathbf{e}_s)_r - Q_{ir}(\mathbf{N})/N_r$ with Linearizer but refreshes it once per fixed-point pass rather than Linearizer's three, at roughly a third of the cost and with accuracy between plain Bard-Schweitzer and Linearizer. Restricted, like `aql` and `qsa`, to strict product-form, load-independent, single-server models.
- `amva.ab` (alias `ab`), `amva.schmidt` and `amva.schmidt-ext`: Akyildiz-Bolch and Schmidt approximations [147] for multi-server stations.
- `amva.tay` (alias `tay`): Tay-Suri arrival-instant approximate MVA [150], restricted to single-server stations.
- `egflin`, `gflin` and `sqni`: feedback Linearizer variants and single-queue numerical iteration, used internally for specialised topologies.
- `qna` and `rqna`: Whitt's Queueing Network Analyzer [18, § 6.7] for open networks with general distributions, and its robust counterpart [162] for single-class open networks with bursty non-renewal arrivals. Under `default`, `rqna` is auto-selected for multi-station bursty open networks, while a single Source-Queue-Sink model is routed to the queueing-system formulas; in either case the arrival autocorrelation is not silently discarded.
- `rqt`: Robust Queueing Theory [8], restricted to single-class open networks with a Source station. Rather than an expected value, it reports a worst-case system time over polyhedral uncertainty sets that replace the stochastic primitives, composing the robust-Burke queue-passage, superposition and thinning theorems of the reference along the visit-ratio path and closing with a heavy-traffic-regressed worst-case bound (`options.config.rqt_regime`, one of `'independent'`, `'normal'` or `'pareto'`; `rqt_exact` evaluates the exact worst case instead of the closed-form bound). Tail coefficients default to $\alpha = 2$ (finite variance) and are set per-stream by `rqt_alpha_a`/`rqt_alpha_s` in $(1, 2]$ to declare heavier tails, which LINE's distributions cannot express directly since they carry finite moments. Accuracy is heavy-traffic-biased: on M/M/1 the error is about 8% at $\rho = 0.9$ but exceeds 50% at $\rho = 0.5$, so `qna` or an exact method should be preferred at low utilization.
- `oi`, `sqd` and `sjn.mva`/`sjn.amva`: conditional MVA for order-independent and P&S stations, Smith Queue Decomposition for closed single-chain blocking-after-service models (§5.3.2), and shortest-job-next analysis of closed models with `SchedStrategy.SJF` stations [82]. None of the three is

requested by the user: a model declaring the corresponding feature is routed to them automatically and the method name is reported back as the actual method used. The SJN route is configured by `options.config.sjn_lattice_max` (default 10^5, above which the Schweitzer fixed point replaces the population lattice), `sjn_ns` (quadrature panels, default 32), `sjn_lfactor` (grid length, default 8) and `sjn_umax` (utilization cap, default 0.999); every other station must then be single-server FCFS, PS, LCFS-PR, SIRO or a delay, and open models are rejected.

For single-class closed models the solver additionally exposes the bound families `aba.*`, `bjb.*`, `gb.*`, `pb.*` and `sb.*`, and for multiclass closed models the Majumdar-Woodside robust box bounds `mwba.upper`/`mwba.lower` [103], also exposed for layered networks by `SolverLN`. For two-station open single-class networks it dispatches the `mm1`/`mmk`/`mg1`/`gig1.*`/`gigk.*` formulas; see Table 8.1 for the complete list. Queues with `POLLING` scheduling are routed to a dedicated polling analyzer implementing the `stationtime` method for cyclic polling with exhaustive, gated and limited service. The algorithms behind all these methods, and the policy by which `default` selects among them, are documented in the internals manual, Chapter `MVA`.

Extended features are handled through the configuration options of Table 8.5: `options.config.highvar` selects the correction applied to non-exponential service at FCFS stations, which is ignored by default and available in the single-server case only; `multiserver` selects the multi-server approximation; `np_priority` the non-preemptive priority approximation; and `fork_join` the transformation applied to fork-join sections, which are assumed to form a directed acyclic graph between each fork and its join. DPS stations, limited load dependence and class dependence are handled by built-in correction factors and need no option. The algorithms behind these settings are described in the internals manual, Chapter `MVA`.

8.2.8 NC: normalizing constant solver

The `NC` class implements a family of solution algorithms based on the normalizing constant of state probability of product-form queueing networks. Contrary to the other solvers, this method typically maps the problem to certain multidimensional integrals, allowing the use of numerical methods such as MonteCarlo sampling and asymptotic expansions in their approximation. The `NC` solver is the recommended back-end for the analysis of load-dependent stations specified through `set_load_dependence` (see §5.2.1); the `nrl`, `nrp`, `nre`, and `rd` methods listed below are designed for this case and remain compatible with the probability metrics `get_prob`, `get_prob_aggr`, and `get_prob_norm_const_aggr`.

The `NC` solver is also the only one to answer `get_prob_sys_marg`, the joint law of the per-station total queue lengths described in §7.1.5. That metric is not a method and is not requested through `options.method`: it is a permanent of the demand matrix replicated once per job, normalized by the same constant the solver already computes, and its optional second argument selects only the estimator of that permanent. Being a state probability it inherits the scope of the product form, so a multiserver station, a load-dependent station and an open or mixed model are refused by name rather than approximated.

When the model declares order-independent (`SchedStrategy.OI`) or pass-and-swap (P&S, `SchedStrategy.PAS`) stations, the `NC` solver dispatches automatically to a dedicated normalizing-constant

Table 8.5: `MVA` configuration options (Python)

Option	Value	Description
`options.config.multiserver`	`'default'`	Equals `'softmin'` at PS queues and `'seidmann'` at FCFS queues.
`options.config.multiserver`	`'seidmann'`	Seidmann's decomposition [140].
`options.config.multiserver`	`'softmin'`	QD-AMVA's softmin approximation [32].
`options.config.multiserver`	`'suri'`	Suri et al.'s multiserver approximation [147].
`options.config.np_priority`	`'default'`	Non-preemptive priority handling. Equals `'cl'`.
`options.config.np_priority`	`'cl'`	Chandy-Lakshmi [38], in the arrival-instant utilization form of Eager-Lipscomb [56].
`options.config.np_priority`	`'shadow'`	Sevcik's shadow server [141].
`options.config.highvar`	`'default'`	Ignored - no correction applied.
`options.config.highvar`	`'interp'`	Diffusion-M/G/k interpolation from [25].
`options.config.highvar`	`'hvmva'`	High-variance MVA as in [20], extended to multiclass as [62, Eq. 3.21].
`options.config.fork_join`	`'default'`	Equals `'mmt'`.
`options.config.fork_join`	`'mmt'`	Mixed-model transformation [53]
`options.config.fork_join`	`'ht'`	Heidelberger-Trivedi [73]
`options.config.rqt_regime`	`'independent'`	Default Table 1 adaptation regime of `rqt` [8].
`options.config.rqt_regime`	`'normal'/'pareto'`	Alternative `rqt` adaptation regimes [8].
`options.config.rqt_exact`	`False`	`rqt` evaluates the exact worst case instead of the closed-form bound when `True`.
`options.config.rqt_alpha_a/s`	2	Tail coefficients of the `rqt` arrival/service streams, in $(1, 2]$.

analyzer, which is reported back as the actual method `oi`. It evaluates the balance functions of the OI stations by a convolution recursion instead of the product-form load terms, and it verifies the permutation-invariance condition that the declared station must satisfy. As with `MVA`, `oi` is not a method name the user requests: it is selected under `default` and `exact`. The importance-sampling method `is` provides a Monte Carlo alternative for OI and P&S networks whose state space makes the exact recursion impractical.

8.2.8.1 Methods and configuration options

The exact algorithms are `ca` (multiclass convolution [18, § 9.1]), `comom` (class-oriented method of moments [23], polynomial in the number of classes), `rgf` (recursion by generating functions [48], single-class closed models only) and `clw` (Choudhury-Leung-Whitt numerical inversion [40], exact up to a controllable aliasing error and efficient when a few chain populations are large). The method name `exact` selects among them automatically, dispatching load-dependent closed models to the load-dependent back-end.

The sampling estimators are `imci` [159], `ls` [24] and `is`, the last sampling job orderings rather than the integral representation, which is what makes it applicable to order-independent and P&S stations [44]; `samples` and `seed` control it. The selector `sampling` maps to `is` on such models and otherwise chooses among `imci`, `ls` and the McKenna-Mitra sampler.

The asymptotic and integral methods are `le` and `kt` [24, 86], `pana` and `panald` [108, 111, 130] for normally-loaded closed models, `cub`, `gm` and `gleint` (cubature and quadrature of the normalizing-constant integral [24]), `mmint2` (McKenna-Mitra integral [108], two-station closed models), `aghq` (quadrature of the simplex factor of that integral [97]), `nrl`, `nrp`, `nre` and `rd` (saddle-point and reduction approximations for

load-dependent models [29]), and `propfair`, a fast proportionally-fair throughput heuristic. `ble` is a bias-corrected variant of `le` that adds the empirical term $(M-1)(1-\log(2\pi)/2)$ before taking the $\epsilon \to 0$ limit of the logistic expansion, which the reference itself requires to stay bounded away from zero; `le` evaluates the limit directly and keeps the resulting $O(1)$ bias, so it is retained unmodified as the published form while `ble` is what `default` now routes normally-loaded closed models through, having measured a lower median error across the whole example suite. The correction cancels in throughput ratios $G(\mathbf{N}-\mathbf{e}_r)/G(\mathbf{N})$, so it changes only the reported normalizing constant, not the mean-performance metrics derived from it. `bkt` plays the same role for `kt`: steepest descent on the generating function carries Stirling's approximation of $\log N_r!$ in place of $\log N_r!$ in every class direction, and `bkt` subtracts that remainder, $\log N_r! - (N_r \log N_r - N_r + \frac{1}{2}\log 2\pi N_r)$, exactly, for each class the saddle point expands. With a think time `bkt` and `ble` coincide to the accuracy of their saddle-point solvers, being the same estimator computed in R and in M dimensions, so the cheaper of the two is whichever of R and M is smaller; without a think time they differ by a constant depending only on the total population. `lekt` exposes that identity as a method: it computes the common estimator on whichever side is cheaper, the Knessl–Tier saddle point when $R \le M$ (or when a class self-loops, which that side extracts exactly) and the logistic mode otherwise, and without a think time it carries the constant $M(1-\log(2\pi)/2) - r(N+M)$ on the logistic side, r the Stirling remainder, so that the two sides agree there as well.

The method `aghq` acts on the simplex factor of the McKenna-Mitra integral rather than on the logistic transform of the whole integrand, and reads the mode and curvature that `le` already computes as a quadrature rule instead of as a closure. It evaluates that factor by adaptive Gauss-Hermite quadrature [97] with q nodes per simplex direction, set by `options.config.aghq_nodes` (default 3) at a cost of q^{M-1} integrand evaluations. The factor is where the error of `le` sits: without think times the integrand is homogeneous, so the radial direction separates and integrates exactly, leaving the simplex integral as the only approximated quantity. At $q = 1$ the single node is the mode and the method reproduces `le` exactly, so raising q buys accuracy in integrand evaluations rather than in a different approximation.

The `nre` method evaluates the same Norlund-Rice integral as `nrl` and `nrp`, but by steepest descent rather than by a Gaussian fit on the contour of unit radii. Two properties of the integrand are exploited. First, the integrand is invariant under a common shift of the class variables, since it is homogeneous of degree $\sum_r N_r$ and that degree cancels against the coefficient-extraction kernel; the redundant direction is therefore quotiented out and the integral is $(R-1)$-dimensional, whereas `nrl` and `nrp` integrate over R dimensions and let the logit or probit Jacobian supply a curvature term along the null direction that belongs to the change of variables rather than to the model. Next, the contour radii are tilted per class to the saddle point, i.e., to the $\mathbf{X}$ solving $X_r \partial \log h / \partial X_r = N_r$, obtained by a damped Newton iteration on the convex cumulant generating function of the tilted distribution; on the untilted contour the origin is not a stationary point of the phase, which is the leading source of bias of the two earlier methods. A second-order Edgeworth correction built from the third and fourth cumulants at the saddle [10, 50] is then applied, giving a relative error of order $O(1/(\sum_r N_r)^2)$. Every evaluation of the integrand sits at real positive demands, so, unlike `nrl` and `nrp`, the method requires no complex arithmetic and remains in the log domain throughout; its cost is $O(IR^2 + R^4)$ evaluations of a single-class load-dependent normalizing constant, hence polynomial in the number of classes.

The Birman-Kogan family [16] covers `bk` (saddle-point evaluation of the generating function, with an

exact pole treatment for chains whose dedicated station is a bottleneck), `bkue` (the single-chain uniform-expansion variant that keeps the pole and the saddle together through an erfc term when they are close, falling back to `bk` on a multiclass model) and `lc`/`lc.ue` (Algorithm 2 of the reference, a Gauss-Seidel sweep that solves each chain against the residual capacity left by the others, with `mva` or `ue` as the inner single-chain solver).

The remaining method is `mem`, the Maximum Entropy Method [89] for open, closed and mixed networks with general (GE-type) arrival and service processes, exact for $M/G/1$, $M/M/c$ and infinite-server stations. It admits Source, Queue, Delay and Sink nodes with FCFS, PS, SIRO, LCFS, LCFS-PR and INF scheduling, rejecting priority disciplines, class switching and reducible routing; multiserver stations are supported in open models, and mixed models are restricted to single-server and infinite-server stations. It is used only on explicit request: `default` never selects it. Convergence is controlled by `options.config.mem_tol` (default 10^{-6}), `mem_maxiter` (default 1000) and `mem_verbose`. It is also the only `NC` method aware of a finite station buffer, every other method being product-form, so that a model carrying a binding `set_capacity` is rejected outright rather than solved as if the buffer were unbounded. That path is reported as `mem.blocking` and is restricted to single-class open models with FCFS capacitated stations whose arrival and service processes have a squared coefficient of variation of at least one.

For cache models, the importance sampling method `sampling` estimates hit probabilities and miss rates by Monte Carlo, which is useful for large cache configurations where the exact methods become expensive:

```
print(NC(model, method='sampling', samples=100000).avg_table())
```

The algorithms behind these methods are documented in the internals manual, Chapter `NC`.

NC method selection. The normalizing constant solver provides both exact and approximate methods. For exact computation, `ca` (convolution algorithm) and `comom` (class-oriented method of moments) are available, with `comom` being generally faster for models with many classes. Among the approximate methods, `nr.logit` and `nr.probit` use Norlund-Rice integrals with different variable transformations and are suitable for large populations, while `nre` evaluates the same integral on a saddle-tilted contour with a second-order Edgeworth correction, which is markedly more accurate on load-dependent models at the same asymptotic cost. The `rd` method applies a reduction heuristic that recursively simplifies the network. The `cub` method employs cubature rules for numerical integration of the normalizing constant. The `mem.blocking` variant extends the same building block to finite station buffers, which no other `NC` method can represent: a capacitated station becomes a censored $GE/GE/c/0; N$ queue, exact for $M/M/1/N$ and $M/M/c/N$, and blocking after service is handled by the holding-node transformation of [148] that restores work conservation. For open networks, the `mem` method applies the Maximum Entropy Method [89] to estimate the joint queue-length distribution via a queue-by-queue decomposition into $GE/GE/1$, $GE/GE/c$ and $GE/GE/\infty$ building blocks; it is exact for $M/G/1$, $M/M/c$ and infinite-server stations and captures the propagation of non-exponential variability (including immediate feedback) through the network. The `default` method automatically selects `mem` through a benchmark-calibrated heuristic: open networks within its feature set whenever any arrival or service process has a non-unit squared coefficient of variation (except bursty external

arrivals at low load, where the GE bound is loose), and closed networks that are non-product-form due to hypoexponential services or class-dependent FCFS rates; purely exponential models and hyperexponential closed FCFS models are instead solved by the native normalizing-constant path. For closed networks, `mem` implements the two-stage pseudo-open decomposition and convolution algorithm [89] on $G/G/1$ and $G/G/\infty$ building blocks; it conserves class populations exactly and reduces to the exact BCMP solution on Markovian single-class networks, but it is selected only on explicit request since the default path is exact for product-form closed models. Mixed open/closed models compose the two algorithms with product-form-style conditioning (capacity reduction of the closed subnetwork by the open load, inflation of the open queue lengths by the closed occupancy), exactly recovering mixed MVA in the product-form limit. The `oi` method analyzes closed order-independent (OI) networks, in which one or more stations serve jobs at a total rate that depends only on the multiset of classes present (the balanced-fairness rank rate [19]), possibly mixed with ordinary BCMP product-form stations: infinite-server (delay), processor sharing, LCFS-PR, SIRO and class-independent-rate FCFS, single- or multi-server. The normalizing constant is assembled exactly over the population lattice, convolving the order-independent balanced-fairness recursion for the OI stations with the load-dependent BCMP weight tables of the remaining stations. The per-class mean queue lengths are then obtained without recourse to Little's law, from the order-independent functional-server identity $E[n_{i,r}] = G^{+}_{i,r}/G - 1$, the OI generalization of the load-dependent functional server, and the per-class throughputs from ratios of normalizing constants. The method is selected automatically, in place of the maximum-entropy heuristic above, whenever the closed model contains at least one order-independent station and every other station is BCMP product-form; a permutation-invariance check on the rate function guards the order-independence assumption at model-linking time. The `morrison` method is the one `NC` route that computes no normalizing constant at all, and it is the clearest illustration of the solver's wider remit. It analyzes the closed two-station network of [113]: one infinite-server (think) station and one single-server discriminatory-processor-sharing station, exponential service, each class alternating between the two. Unequal DPS weights destroy the product form, so there is no constant to expand; instead the substitution $P(\mathbf{n}) = \langle \mathbf{w}, \mathbf{n} \rangle f(\mathbf{n})$ clears the $1/\langle \mathbf{w}, \mathbf{n} \rangle$ denominators of the balance equations, whose generating function then satisfies a linear partial differential equation with affine coefficients. Rescaling about $z = 1$ and expanding in inverse powers of $\sqrt{N}$, in the moderately-heavy regime $\rho = 1 - a/\sqrt{N}$ with populations $K_j = Nb_j$, leaves a degenerate leading operator whose kernel is spanned by the functions of a single similarity variable; the solvability condition along its characteristic reduces the p-dimensional problem to one ordinary differential equation, whose bounded solution is a family of parabolic-cylinder integrals. The solver returns the resulting two-term approximations to the mean queue lengths and, by Little's law so that the reported table is self-consistent, to the mean sojourn times. Accuracy improves as the populations grow with the usage held near saturation, which is precisely the regime where a Markov chain is intractable and simulation converges slowly; the expansion also remains valid above saturation, where its leading term recovers the fluid approximation. It is selected automatically on that shape and is the only method admitted there, since every other `NC` route would silently drop the weights and answer with the egalitarian processor-sharing network; DPS models of any other shape are refused rather than approximated, and `logNormConstAggr` is returned as `NaN`.

8.2.8.2 Busy periods of a subnetwork

The busy period of a given order of a set of stations is the time from the instant an arriving job raises the jobs held by the set to that order, up to the next instant when the set falls back below it. At order one it is the ordinary busy period of the subnetwork, and at a high order it is a heavy-traffic period, so the family describes how long congestion of a given depth persists. Comparing idle and busy periods of a station yields its utilisation, and the busy periods of a high-priority class are the blocking times of the lower-priority ones.

The `NC` method `getAvgBusyPeriod(stations, n)` evaluates it exactly by the mean value analysis of [49]. The closed and the open forms are both a ratio: the numerator sums the normalizing constant of the subnetwork over every population at or above the order, weighted in the closed case by the normalizing constant of the complement at the matching residual population; the denominator takes the same constants one population lower and multiplies them by the total rate at which jobs enter the set from outside, which in the open case also collects the external arrivals. Both are evaluated in the logarithmic domain, which keeps the individual normalizing constants from overflowing. The result is exact on single-chain product-form models and, being a function of the stationary distribution and of an ergodic flow rate alone, it is insensitive to the service time distributions beyond their means. A closed subnetwork must be a proper subset of the stations, since a busy period is started from outside it; an open subnetwork may be the whole network.

```
b, lG, lH = NC(model).getAvgBusyPeriod([0, 1], [1, 2, 3])
```

The same quantity can be measured by simulation, which is the recommended cross-check on models outside the product-form class: see §8.2.5.2.

8.2.9 SSA: stochastic simulation algorithm solver

The `SSA` class is a stochastic simulator for continuous-time Markov chains. It reuses parts of the `CTMC` machinery to generate network states and then samples the state dynamics by selecting among enabled events according to their rates. The sampling algorithms and state representation are described in the LINE internals manual (`LINE-internals-python.pdf`, Chapter `SSA`).

The state space is not generated upfront, but typically stored during the simulation, starting from the initial state. If the initialization of a station generates multiple possible initial states, `SSA` initializes the model using the first state found. The list of initial states for each station can be obtained using the `get_state` function of the `Network` class.

8.2.9.1 Methods and configuration options

The `SSA` solver offers two general-purpose methods: `'serial'` and `'para'`. The serial method runs on a single core, while the parallel methods run on multicore The `'default'` method resolves to `'nrm'` on models that support it (see below) and to `'serial'` otherwise.

`SSA` solver also implements the much faster next reaction method (`'nrm'`, see [3]). This is available for closed queueing models with exponential service under the `INF` and `PS` disciplines, as well as buffered `FCFS`

and `LCFS` stations through integer (inverse-CDF) routing that preserves the per-buffer job order. The `'nrm'` method is selected by default on such models. Moreover, by setting `options.config.state_space_gen` to `'full'` it is possible to explicitly generate during the simulation the simulated state space and its steady-state probability.

`SSA` accepts a warmup-discard option `options.config.warmupfrac` (default `0.0`, disabled) that drops the leading fraction of collected samples before computing per-station means, mitigating transient bias on open networks; values of 0.1–0.3 typically bring open-network averages within $\sim$1% of `LDES` (which uses MSER-5 truncation). `SSA` also supports the full set of state-dependent dispatching strategies via a routing callback invoked at each event firing — `RAND`, `PROB`, `RROBIN`, `WRROBIN`, `JSQ`, `SQ`, with the `SQ` marginals matching the with-replacement, first-occurrence tie-break used by `LDES` and cached per $(d, \text{ndest}, n[\cdot])$ to amortise enumeration cost.

SSA method selection. The `serial` method implements Gillespie's stochastic simulation algorithm on a single core, while `para` runs independent replicas in parallel for faster convergence. The `nrm` (next reaction method) offers significantly faster simulation for population models with processor-sharing or infinite-server scheduling, as it avoids recomputing all transition rates after each event.

8.3 Metasolvers

A metasolver does not analyze a model directly. It transforms the model, or decomposes it into a family of simpler ones, and delegates each resulting subproblem to one of the solvers of §8.2 or §8.4, which it may call many times; the features it admits and the accuracy it attains are therefore largely those of the solver it is given. Four are available in LINE, and only the last is documented here:

- `AUTO` chooses the back-end solver itself from structural features of the model, and falls back along a list of feasible solvers when the one it picked does not implement the requested operation (§1.3).
- `LN` solves a `LayeredNetwork` by decomposing it into layers and iterating a fixed point over their artificial delays, each layer being solved by a solver of the caller's choosing (§6.5).
- `ENV` solves a model embedded in a random environment by analyzing one sub-model per environment stage and coupling the stages through their transition process (§6.6).
- `UQ` propagates uncertainty in the model parameters, solving one realization of the model per sample or quadrature node.

8.3.1 UQ: uncertainty quantification meta-solver

The `UQ` (uncertainty quantification) solver is a meta-solver that extends LINE's capabilities to handle models with parameter uncertainty. In many practical scenarios, model parameters such as service rates or routing probabilities are not known precisely, but are instead characterized by a set of plausible alternatives with associated likelihoods. Rather than requiring the analyst to manually explore each alternative, the `UQ` solver automates this process by accepting parameters specified as probability distributions over discrete alternatives.

When a model contains parameters defined using the `Prior` distribution class, which encodes a discrete set of alternative parameter values along with their prior probabilities, the `UQ` solver detects these uncertain parameters and expands the model into a family of concrete instances. Each instance corresponds to one combination of parameter values from the alternatives. The solver then applies a user-specified inner solver to each instance, obtaining performance metrics for all alternatives. Finally, it aggregates the results using the prior probabilities as weights, producing expected performance metrics that account for the parameter uncertainty.

8.3.1.1 Methods and configuration options

The constructor for the `UQ` solver takes two arguments: the model containing uncertain parameters, and a solver factory that specifies which analysis method should be applied to each concrete instance. The solver factory can be any of LINE's standard solvers such as `MVA`, `CTMC`, or `JMT`.

```
post = UQ(model, MVA)
```

The `UQ` solver provides several methods for retrieving results. The `get_avg_table` method returns the prior-weighted expected values of all performance metrics, effectively averaging over the parameter uncertainty. The `get_posterior_table` method returns the complete set of results for each alternative, allowing the analyst to examine how performance varies across different parameter values. Additionally, the `get_posterior_dist` method takes a metric type, node, and class as arguments and returns the posterior distribution of that metric as an `EmpiricalCDF` object, which can be used to compute quantiles or probabilities.

8.3.1.2 Priors

To specify parameter uncertainty, the analyst creates a `Prior` distribution by providing a list of alternative distributions and their corresponding probabilities. This prior distribution can then be used in place of a standard distribution when configuring model parameters such as service times.

```
prior = Prior([Exp(1), Exp(2), Exp(3)], [0.3, 0.5, 0.2])
queue.set_service(job_class, prior)
```

In this example, the service time at the queue is uncertain, with three possible mean service rates having probabilities 0.3, 0.5, and 0.2 respectively. The `UQ` solver will analyze all three scenarios and provide both individual results and weighted averages. This capability is particularly useful for sensitivity analysis, robust design under uncertainty, and understanding the range of possible performance outcomes when input parameters cannot be measured precisely.

8.3.1.3 Interval parameters

Assigning probabilities to the alternatives is not always possible. An analyst may be able to bound a service demand between a best and a worst case, without being willing to say how the values in between are distributed. For this situation the `UQ` solver offers the `get_interval` method, which discards the prior weights and keeps only the endpoints of the support, and its tabular form `get_interval_table`. Each metric is then returned as a pair of values, a lower and an upper endpoint, instead of a single expectation.

Two regimes are possible and the returned `exact` flag distinguishes them. If the model is a closed single-class network of load-independent single-server queues and delay stations, and the uncertainty affects service demands only, then the interval is computed by the algorithm of Lüthi and Haring [102], exposed at the API level as `pfqn_mva_interval`. Mean value analysis of such a network is monotone in each of its inputs: the throughput decreases in every demand and in the think time and increases in the population, while the queue length and the residence time of a station increase in the demand of that station and decrease in the demands of the others. The range of each output over the input box is therefore attained at a corner of the box, and the interval is obtained by a small number of ordinary MVA solutions, without expanding the model into a family of instances at all. The reported interval is exact: every value inside it is realized by some admissible choice of the parameters.

If the model does not meet these conditions, for instance because it has several job classes, the solver falls back on the range of the results across the instances it did solve, and reports `exact` as false together with the condition that was violated. This fallback covers the whole support when the uncertain parameters take finitely many values, but it is only an inner approximation when a parameter varies continuously, since the sampled values are interior points of its range rather than its endpoints.

Such an interval is the range of the model's own predictions over the set of admissible parameters, conditional on the true parameters lying inside the declared intervals. It says nothing about the accuracy of mean value analysis itself, and is therefore a different object from the bounds returned by the `BA` solver, which bracket the exact solution of a model whose parameters are known. The two must not be combined.

8.4 External solvers (wrappers)

The solvers in this section do not implement an analysis of their own: each drives a third-party tool as an external process. LINE transforms the model into the input format of that tool, runs it, and parses its output back into the LINE result tables, so that a wrapper is used exactly like a native solver and returns the same objects. Two consequences follow. First, the tool must be installed and reachable – on the system `PATH`, or at a location given through the options – and the solver is unavailable otherwise; the environment check of §2.1.2 reports which of them can be found. Second, the accuracy, the supported model features and the failure modes are those of the back end rather than of LINE.

8.4.1 JMT: Java Modelling Tools wrapper

The class is a wrapper for the `JMT` and consists of a model-to-model transformation from the data structure into the JMT's input XML formats (either `.jsimg` or `.jmva`) and a corresponding parser for JMT's results. Upon first invocation, the `JMT` JAR archive will be searched in the MATLAB path and if unavailable automatically downloaded.

8.4.1.1 Methods and configuration options

This solver offers several methods. The default method is the JSIM solver (`'jsim'` method), which runs JMT's discrete-event simulator. For parallel simulation, the solver supports both serial and parallel execution methods. The alternative method is the JMVA analytical solver (`'jmva'` method), which is applicable only to queueing network models that admit a product-form solution. This can be verified calling `model.has_product_form_solution` prior to running the JMVA solver.

For transient analysis, the `'replication'` method runs independent simulation replications and interpolates the resulting sample paths to compute transient averages. This method is automatically selected when the `timespan` option is set to a finite interval (e.g., `[0, T]` for some finite T) and the method is set to `'default'`.

In the transformation to JSIM, artificial nodes will be automatically added to the routing table to represent class-switching nodes used in the simulator to specify the switching rules. One such class-switching node is defined for every ordered pair of stations (i, j) such that jobs change class in transit from i to j.

8.4.2 LQNS: layered queueing network solver wrapper

The `LQNS` class wraps the `lqns` analytical solver and the `lqsim` simulator of the LQNS suite [60]. It accepts `LayeredNetwork` models only: the model is exported to the LQNS XML interchange format (`.lqnx`), the back-end command is run as an external process, and the resulting `.lqxo` file is parsed back into the layered result tables. It is, together with the native metasolver `LN`, one of the two ways of solving a layered model in LINE, and the two are directly comparable on the same model, as shown in §6.5.

8.4.2.1 Methods and configuration options

- `options.method='default'` or `'lqns'`: the `lqns` analytical solver with its default settings. The method name `lqnsdefault` is an alias retained for backward compatibility.
- `options.method='exactmva'`: `lqns` with exact MVA in each layer instead of the default approximate one.
- `options.method='srvn'` and `'srvn.exactmva'`: the same two, with SRVN layering.
- `options.method='sim'` or `'lqsim'`: the `lqsim` simulator, run for `options.samples` events (passed as `-A`).

Table 8.6: QNS solution methods

Method	Flag	Description
default	-mrolia or none	Sets config.multiserver to rolia, so a model with multi-server FCFS stations is solved by -mrolia; without such stations no flag is emitted and qnsolver runs its built-in product-form solution.
conway	-mconway	Conway's multi-server MVA approximation.
reiser	-mreiser	Reiser's multi-server MVA approximation.
rolia	-mrolia	Rolia-Sevcik multi-server approximation [132].
zhou	-mzhou	Zhou's multi-server approximation.
suri	none	Sets config.multiserver to suri for compatibility with the homonymous option of MVA. qnsolver has no flag for it, so the back end runs its built-in default.
schmidt	none	Sets config.multiserver to schmidt for compatibility with the homonymous option of MVA. qnsolver has no flag for it, so the back end runs its built-in default.

options.config.multiserver forwards a -Pmultiserver=... pragma that selects the LQNS multi-server approximation (conway, reiser, rolia, zhou, suri or schmidt; default maps to rolia), and is passed verbatim, so any pragma the installed LQNS understands is accepted. options.config.remote and options.config.remote_url dispatch the run to an lqns-rest service instead of a local binary, which is useful where the tools cannot be installed.
LINE ships no LQNS binary, names no container image holding one, and never pulls one: LQNS is distributed by its authors under an evaluation agreement that forbids providing the software to third parties, so the binary must be one the user has obtained and is licensed for. It is available from http://www.sce.carleton.ca/rads/lqns/, and lqns and lqsim must be on the system PATH. The wrapper, its result mapping and remote execution are covered in full in §6.5.

8.4.3 QNS: qnsolver wrapper

The QNS class is a wrapper around the standalone qnsolver command-line tool distributed with the LQNS suite. It performs a model-to-model transformation from the data structure into a JMVA-format input file (.jmva), invokes qnsolver as an external process, and parses the resulting per-station chain throughputs, queue-lengths, response times, and utilisations back into the LINE result tables. QNS is recommended for product-form closed networks with multi-server FCFS stations, where it offers several well-known multi-server approximations not implemented natively by MVA.

For non-product-form models, the MATLAB version of QNS performs an automatic conversion to a layered queueing network via QN2LQN and then dispatches the analysis to LQNS. The JAR and Python versions invoke qnsolver directly without this fallback, so for non-product-form workloads they may return less accurate results. The qnsolver executable is located on the system PATH; if it is missing, the call to QNS aborts.

8.4.3.1 Methods and configuration options

The methods available for the QNS solver are listed in Table 8.6. Each one maps to a multi-server approximation of qnsolver, selected through the flag shown in the second column.

The selected method is reflected in the actual back-end command line, e.g. `qnsolver -l model.jmva -mconway -o result.jmva`, so users can replicate `QNS` runs by hand if needed.

Chapter 9

Parameter Inference

LINE supports parameter inference, i.e., the estimation of queueing network model parameters from empirical data. Given observed system measurements such as response times, throughputs, queue lengths, and utilizations, the inference methods recover the model parameters—primarily service demands—that best explain the data. While the LINE solvers perform *forward analysis* (computing performance metrics from model parameters), the inference functions address the *inverse problem*: recovering model parameters from observed metrics.
To cite the parameter inference methods, we recommend referencing [99, 123, 144, 159, 160].

9.1 Quick start

The typical workflow for parameter estimation is:

1. Define a model: create a LINE `Network` with known structure but unknown service demands (set to `NaN`).
2. Collect measurements: wrap observed data as `SampledMetric` objects, specifying the metric type, timestamps, values, node, and (optionally) job class.
3. Configure the estimator: create a `ParamEstimator` with the model and desired method.
4. Add data and estimate: add samples to the estimator, interpolate, and call `estimateAt` to obtain the estimated service demands.

The following example estimates service demands at a PS queue in a closed network using the UBO method:

```
import numpy as np
from line_solver import (Network, Delay, Queue, Exp,
    ClosedClass, SchedStrategy, MetricType)
from line_solver.inference import ParamEstimator, SampledMetric
```

```
# Step 1: Define the queueing network model
model = Network('model')
delay = Delay(model, 'Delay')
queue = Queue(model, 'Queue1', SchedStrategy.PS)
class1 = ClosedClass(model, 'Class1', 1, delay, 0)
class2 = ClosedClass(model, 'Class2', 2, delay, 0)

# Set known service times at the delay station
delay.setService(class1, Exp.fitMean(1.0))
delay.setService(class2, Exp.fitMean(1.0))

# Mark queue service demands as unknown (to be estimated)
queue.setService(class1, Exp(float('nan')))
queue.setService(class2, Exp(float('nan')))

# Define routing: each class cycles between delay and queue
P = model.initRoutingMatrix()
P.set(class1, class1, delay, queue, 1.0)
P.set(class1, class1, queue, delay, 1.0)
P.set(class2, class2, delay, queue, 1.0)
P.set(class2, class2, queue, delay, 1.0)
model.link(P)

# Step 2: Generate synthetic measurement data from a ramping workload
# True demands are D1=0.1 and D2=0.3; the arrival rate ramps up over the
# observation window, driving utilization from ~0.2 to ~0.85
n = 40
ts = np.arange(1, n + 1, dtype=float)
ramp = np.linspace(0.8, 3.4, n)
arvr1 = ramp + (np.random.rand(n) - 0.5) * 0.10
arvr2 = 0.5 * ramp + (np.random.rand(n) - 0.5) * 0.05
util = 0.1 * arvr1 + 0.3 * arvr2  # U = D1*lam1 + D2*lam2
respt1 = 0.1 / (1 - util)  # M/G/1 response time formula
respt2 = 0.3 / (1 - util)

# Step 3: Configure the estimator with the UBO method
options = ParamEstimator.defaultOptions()
options['method'] = 'ubo'
se = ParamEstimator(model, options)

# Step 4: Add observed samples and run estimation
se.addSamples(SampledMetric(MetricType.ArvR, ts, arvr1, queue, class1))
se.addSamples(SampledMetric(MetricType.ArvR, ts, arvr2, queue, class2))
se.addSamples(SampledMetric(MetricType.RespT, ts, respt1, queue, class1))
se.addSamples(SampledMetric(MetricType.RespT, ts, respt2, queue, class2))
se.addSamples(SampledMetric(MetricType.Util, ts, util, queue))
se.interpolate()  # align all samples to common timestamps
estVal = se.estimateAt(queue)
```

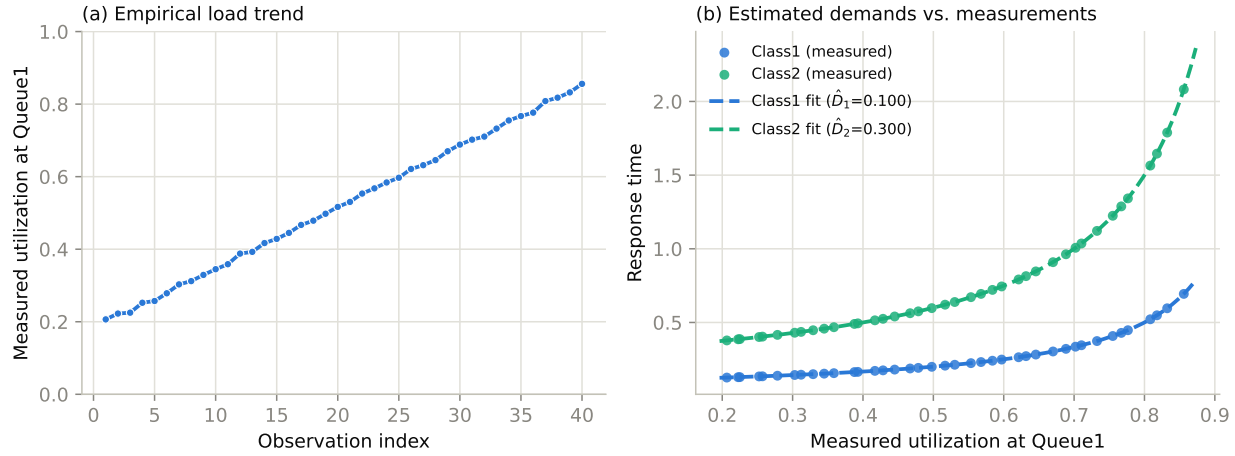


Figure 9.1: Quick-start example: (a) the empirical utilization trend as the workload ramps up over the observation window; (b) the measured response times against utilization, together with the M/G/1 curves $\hat{D}_r/(1-u)$ reconstructed from the demands $\hat{D}_1, \hat{D}_2$ estimated by the UBO method.

```
print('Estimated demands:', estVal)  # close to [0.1, 0.3]

# Step 5: Plot the empirical trend against the recovered M/G/1 fit
import matplotlib.pyplot as plt
D1_est, D2_est = np.asarray(estVal).flatten()
u_fit = np.linspace(util.min() * 0.95, util.max() * 1.02, 200)
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(9, 3.6))
ax1.plot(ts, util, marker='o')
ax1.set_xlabel('Observation index'); ax1.set_ylabel('Measured utilization')
ax2.scatter(util, respt1, label='Class1 (measured)')
ax2.scatter(util, respt2, label='Class2 (measured)')
label1 = f'Class1 fit (D1={D1_est:.3f})'
label2 = f'Class2 fit (D2={D2_est:.3f})'
ax2.plot(u_fit, D1_est / (1 - u_fit), '--', label=label1)
ax2.plot(u_fit, D2_est / (1 - u_fit), '--', label=label2)
ax2.set_xlabel('Measured utilization'); ax2.set_ylabel('Response time')
ax2.legend()
plt.show()
```

9.2 Specifying measurement data

Parameter estimation requires observed performance measurements to be provided as `SampledMetric` objects. Each `SampledMetric` encapsulates a single time series and is constructed from the following arguments:

1. A metric type (one of `MetricType.ArvR`, `MetricType.RespT`, `MetricType.Util`, `MetricType.QLen`, or `MetricType.Tput`).
2. A vector of timestamps `ts`, of length n.
3. A vector of observed values `data`, also of length n, where `data[k]` is the measurement taken at time `ts[k]`.
4. The node at which the measurements were taken (e.g., a `Queue` or `Delay` station).
5. Optionally, the job class to which the measurements refer. If omitted, the metric is treated as an aggregate measurement across all classes.

The five metric types are summarized below:

Metric type	Description	Unit / range
`ArvR`	Arrival rate at the node	jobs / time unit
`RespT`	Response time (waiting + service)	time units
`Util`	Utilization of server capacity	$[0, 1]$
`QLen`	Queue length (jobs present at the node)	jobs (non-negative)
`Tput`	Throughput (completions at the node)	jobs / time unit

Timestamps and values must be numeric vectors of the same length. The timestamps should be monotonically increasing and represent the instants at which each measurement was collected. The values represent the observed metric at each timestamp. The following example creates a `SampledMetric` containing 30 per-class arrival rate observations at a queue:

```
ts = np.arange(1, 31, dtype=float)          # 30 timestamps: [1, 2, ..., 30]
data = 2 * np.ones(30) + np.random.rand(30) # 30 observed arrival rates
sample = SampledMetric(MetricType.ArvR, ts, data, queue, class1)
```

When the job class argument is omitted, the metric is treated as an aggregate measurement across all classes. This is common for utilization data, which monitoring tools typically report at the station level rather than per class:

```
util_sample = SampledMetric(MetricType.Util, ts, util_data, queue)
```

Different estimation methods require different combinations of metric types (see Table 9.1). For example, UBR requires per-class arrival rates and station utilization, while MCMC requires per-class queue-length samples.

9.2.1 Periodic samples and trace data

By default, a `SampledMetric` represents periodic samples: the values are averages or snapshots collected at regular intervals. Some estimation methods (MLPS, FMLPS, Gibbs) instead require individual job traces, where each entry records a single job event. In a trace, timestamps are irregular (one per event rather than one per interval), and values represent instantaneous observations for that event. To indicate that a `SampledMetric` contains trace data, call `set_trace()`:

```
# arrival_times[k]: time of the k-th arrival
# rates[k]: instantaneous arrival rate at that time
arvr_trace = SampledMetric(MetricType.ArvR, arrival_times, rates, queue, class1)
arvr_trace.set_trace()
```

9.2.2 Conditional metrics

Some estimation methods require measurements that were recorded conditionally upon a specific event. For example, the ERPS estimator uses queue-length observations collected only at the instants when a job of a given class arrives. This conditional relationship is expressed using the `Event` class:

```
from line_solver.inference import Event
evt = Event(EventType.ARV, queue, class1)    # arrivals of class1
ql = SampledMetric(MetricType.QLen, ts, ql_data, queue)
ql.set_conditional(evt)                       # queue length seen by arriving ...
class1 jobs
```

9.2.3 Configuring and running the estimator

The `ParamEstimator` takes a LINE `Network` model in which the unknown service demands are set to `Exp(NaN)`. After adding all measurement samples, a call to `interpolate()` aligns the time series onto common timestamps (using linear interpolation so that all metrics share the same time grid), and `estimateAt()` runs the estimation:

```
options = ParamEstimator.defaultOptions()
options['method'] = 'ubo'                    # choose estimation method
se = ParamEstimator(model, options)
se.addSamples(arvr_sample)                   # add each SampledMetric
se.addSamples(respt_sample)
se.addSamples(util_sample)
se.interpolate()                             # align timestamps
estVal = se.estimateAt(queue)                # returns estimated demands
```

The `estimateAt` method returns the estimated service demands and also updates the model's service distributions in-place. If the best method is not known in advance, `auto_method()` selects one automatically based on the types of data that have been added.

The estimator accepts several configuration options through `ParamEstimator.defaultOptions()`. The `method` option selects the estimation algorithm (see Section 9.3; default: `'ubr'`). The `solver` option specifies the forward solver used during estimation (default: `SolverMVA`). Iterative methods respect `iter_max` (default: 1000) and `tol` (default: 10^{-3}). For open networks, several methods automatically construct a closed equivalent with population controlled by `openPopulation` (default: 100).

9.3 Estimation methods

LINE provides twelve estimation methods, each suited to different data availability scenarios. Table 9.1 summarizes all methods along with their required input data and references.

Table 9.1: Estimation methods for `ParamEstimator`.

Method	Required Data	Description	Refs.
`ubr`	ArvR, Util	Utilization-based non-negative least squares regression	[144]
`ubo`	ArvR, RespT, Util	Utilization-based constrained quadratic optimization	[99]
`erps`	RespT (trace), QLen (cond.)	Extended regression for processor sharing stations	[123]
`ekf`	ArvR, RespT, Util	Sequential estimation via Extended Kalman Filter	[144]
`mle`	ArvR, RespT, Util	Maximum likelihood via bounded optimization with forward solver	[123]
`mcmc`	QLen (aggregate)	Gibbs sampling with product-form queueing network likelihood	[159]
`qmle`	QLen (per-class)	Closed-form quick maximum likelihood estimation	[160]
`mlps`	ArvR (trace), RespT (trace)	Exact CTMC-based maximum likelihood for PS stations	[123]
`fmlps`	ArvR (trace), RespT (trace)	Fluid-limit maximum likelihood for PS stations	[123]
`gibbs`	ArvR (trace), RespT (trace), Tput	Gibbs sampling from individual job trace data	[159]
`vi`	QLen (all stations, trace)	Variational inference from noisy queue-length readings; returns a conjugate Gamma posterior per service rate	[119]

9.3.1 Variational inference

The `vi` method estimates service rates from noisy queue-length readings taken over time, following Perez and Casale [119]. It differs from the other estimators in two respects. It reads the queue lengths of *every* station rather than only the estimated one, because the transition counts the method is written in are pinned by the whole picture; and it returns a conjugate Gamma *posterior* per estimated rate, not only a point estimate, with the mean service time under that posterior reported as the estimate and the posterior itself left in `options.posterior` as $[\alpha\ \beta]$ rows.

The network is translated into the transition set $\eta = (i, j, c)$ of the paper, with transition rate $\lambda_\eta = \mu_{i,c}\, p^c_{i,j}$; the routing probabilities are taken as known from the model and every rate other than those of the requested nodes is held at its model value. The trajectory is reparameterised by the transition counts Y^η, so that the station marginals decouple, and the variational family is a product of inhomogeneous pure-birth processes, one per transition, times a product of Gamma densities over the unknown rates. The state space is expanded by adding δ to every feasible rate, which lets queue lengths go negative and keeps the approximating measure mutually absolutely continuous with the target; the original model is recovered as $\delta \to 0$. Each iteration runs,

per transition, a backward pass for the Lagrange multipliers with multiplicative jumps at the observation epochs, a rate update, and a forward pass of the master equation, after which the Gamma posteriors are refreshed from the expected number of firings and the expected exposure time of each station-class pair. Expectations over the other transitions are taken on a deterministic Halton lattice mapped through the inverse marginal c.d.f., so that the estimator carries no random-number stream and is reproducible across the four implementations. Class switching and caches are refused.

Beside `iter_max` and `tol`, the method reads the following options:

- `epsilon` (default 0.05): the probability that a reading is faulty. Each observation is treated as exact with that complementary probability and as uniform over the remaining feasible values otherwise.
- `prior_shape`: the shape of the Gamma prior on each estimated rate; its rate parameter is set from the model, so that the prior mean is the rate the model starts from.
- `ngrid`: the number of points of the time grid on which the backward and forward passes are integrated.
- `nsamples`: the number of lattice points per marginal.
- `ymax`: the truncation level of the transition counts.
- `delta`: the expansion parameter δ described above.

The example `est_vi_closed` estimates the service rate of a queue in a closed loop from ten readings of which one in ten is faulty.

9.4 Inference examples

The table below lists the inference example scripts available under the `examples/inference/` folder.

Table 9.2: Inference examples

Example	Problem
Utilization-Based Regression (UBR)	
`est_ubr_closed`	UBR estimation in a closed queueing network
`est_ubr_open`	UBR estimation in an open queueing network
`est_ubr_changepoint`	UBR estimation with workload change-point detection
Utilization-Based Optimization (UBO)	
`est_ubo_closed`	UBO estimation in a closed queueing network
`est_ubo_open`	UBO estimation in an open queueing network
`est_ubo_changepoint`	UBO estimation with workload change-point detection
`est_ubo_variants_closed`	Comparison of UBO and MLE estimators in a closed network
Extended Kalman Filter (EKF)	
`est_ekf_closed`	EKF estimation in a closed queueing network
`est_ekf_open`	EKF estimation in an open queueing network
`est_ekf_changepoint`	EKF estimation with workload change-point detection
Maximum Likelihood (MLE)	
`est_mle_open`	MLE estimation in an open queueing network
Markov Chain Monte Carlo (MCMC)	
`est_mcmc_closed`	MCMC estimation in a closed queueing network
Quick Maximum Likelihood (QMLE)	

Continued on next page

Table 9.2 – Inference examples. *Continued from previous page*

Example	Problem
`est_qmle_closed`	QMLE estimation in a closed queueing network
Extended Regression for PS (ERPS)	
`est_erps_closed`	ERPS estimation in a closed queueing network
`est_erps_open`	ERPS estimation in an open queueing network
Trace-Based Estimation	
`est_trace_closed`	Trace-based estimation (MLPS/FMLPS) in a closed network

9.5 Layered network parameter identification

The methods described so far estimate service demands of flat queueing networks. LINE additionally provides `infer_lqn`, which identifies hidden parameters of a *layered* queueing network—activity host demands and task think times—from measured performance data (response times, utilizations, throughputs). This addresses the inverse problem for `LayeredNetwork` models, using the Extended Kalman Filter (EKF) tracking scheme of [167].

The unknown parameter vector $\mathbf{a}$ is modelled as a zero-mean random walk $\mathbf{a}_k = \mathbf{a}_{k-1} + \mathbf{w}_k$, and the measurement as $\mathbf{z}_k = \mathbf{h}(\mathbf{a}_k) + \mathbf{v}_k$, where $\mathbf{h}$ is the layered solver (`SolverLN` by default, configurable). At each step the filter forms the prediction error $\mathbf{e}_k = \mathbf{z}_k - \mathbf{h}(\mathbf{a}_{k-1})$ and updates $\mathbf{a}_k = \mathbf{a}_{k-1} + \mathbf{K}_k\mathbf{e}_k$, where the gain $\mathbf{K}_k = \mathbf{P}'_k\mathbf{H}_k^\top(\mathbf{H}_k\mathbf{P}'_k\mathbf{H}_k^\top + \mathbf{R})^{-1}$ uses the sensitivity matrix $\mathbf{H}_k = \partial\mathbf{h}/\partial\mathbf{a}$ obtained by finite differences. The drift covariance $\mathbf{Q}$ and measurement covariance $\mathbf{R}$ are diagonal, tuned by the factors `QFac` and `RFac` respectively.

Each estimated parameter is named by a specification giving its kind (`hostdem` for an activity host demand, `think` for a task think time) and the element name; each observation by a metric (`RespT`, `Util`, `Tput` or `QLen`) and an element name. The measurements are passed as a matrix `Z` of size (number of observations) $\times$ (number of time steps), one column per step. The estimated parameter trajectory is returned in `ahat`, whose last column is the final estimate; the model is updated in place. In the following, `model` is a `LayeredNetwork` whose parameters are set to an initial guess.

```
from line_solver.inference import infer_lqn
# hidden parameters: a reference-task think time and an activity host demand
param_spec = [{'type': 'think',   'name': 'T1'},
              {'type': 'hostdem', 'name': 'AS3'}]
# observed metrics: a response time and two processor utilizations
obs_spec   = [{'metric': 'RespT', 'name': 'E1'},
              {'metric': 'Util',  'name': 'P1'},
              {'metric': 'Util',  'name': 'P2'}]
# Z: an (nobs x nsteps) array of measurements, one column per step
opt = {'QFac': 0.1, 'RFac': 0.2, 'gammaT': 1.0}
model, info = infer_lqn(model, param_spec, obs_spec, Z, opt)
print(info['ahat'][:, -1])            # final parameter estimate
```

A complete, runnable version that synthesises a measurement sequence with a workload change and recovers the parameter trajectory is provided as `examples/basic/layeredModel/lqn_paramident.py`.

Chapter 10

Optimization

10.1 Overview

Besides evaluating a given model, LINE can solve design problems on queueing networks. While the solvers answer evaluation questions (given a model, what are its performance metrics?), the optimization layer answers design questions: how many servers, what service rates, which routing probabilities, or what job populations optimize a cost or performance objective subject to service-level constraints.

An optimization problem couples a LINE `Network` model with three ingredients:

- *Decision variables*, declaring which model parameters are free to change and over what ranges;
- An *objective*, such as cost minimization or performance maximization;
- *Constraints*, bounding performance metrics such as response times or utilizations.

Each candidate configuration is evaluated by applying the decision variable values to a copy of the model and solving it with `SolverAuto`, which automatically selects an appropriate analysis algorithm (e.g., MVA). The optimization layer never inspects solver internals: all metrics are read from the standard `getAvgTable` and `getAvgSysTable` result tables.

The optimization layer is available in the MATLAB, Java and Python codebases with the same class names, options and default values. The differential evolution solver reproduces the same pseudo-random sequence in all three, so a seeded run explores an identical trajectory and returns identical results across languages.

10.2 Requirements

The optimization API ships with the LINE Python package and becomes available once it is installed. It relies on `numpy` and `scipy`, which are installed together with LINE by `pip install line-solver`. The optional `networkx` dependency enables dependency-ordered decomposition workflows and can be installed

separately, with `pip install networkx`, when needed. All optimization classes are exported from the top-level `line_solver` package alongside the modeling and solver classes, and are also available under the `line_solver.opt` subpackage.

10.3 Examples

In this section, we present some examples that illustrate the optimization features. The relevant scripts are included under the `examples/opt/` folder and each runs in seconds. The general structure of an optimization script consists of four blocks:

1. Definition of the LINE network model
2. Declaration of the decision variables
3. Specification of the objective and constraints
4. Solution

Every public method is offered both in CamelCase and snake_case, following LINE's dual naming convention (e.g., `addVariable` and `add_variable` are equivalent).

10.3.1 Tutorial 1: Sizing an M/M/c queue

The M/M/c queue is a classic model of a service center with c parallel servers fed by a common infinite-capacity buffer. In this example, we wish to find the minimum number of servers such that the utilization does not exceed 50%. Jobs arrive at rate $\lambda = 3$ job/s and each server works at rate $\mu = 1$ job/s, hence the utilization with c servers is $\rho = \lambda/(c\mu) = 3/c$. It is known from theory that the optimum is therefore $c^* = 6$; we wish to recover this value. The following script (`opt_server_sizing.py`) solves the problem with the default differential evolution solver, at a cost of 10 units per server:

```
from line_solver import Network, Queue, Source, Sink, OpenClass, Exp, SchedStrategy
from line_solver import (OptimizationProblem, ServerAllocation,
                         MinimizeCost, UtilizationConstraint)

## Block 1: LINE model (Source -> Queue -> Sink)
model = Network("MMc")
source = Source(model, "Arrivals")
queue = Queue(model, "Server", SchedStrategy.FCFS)
sink = Sink(model, "Departures")
jobs = OpenClass(model, "Jobs")
source.setArrival(jobs, Exp(3.0))    # lambda = 3
queue.setService(jobs, Exp(1.0))    # mu = 1 per server
model.addLink(source, queue)
model.addLink(queue, sink)
```

```
## Block 2: decision variables
problem = OptimizationProblem(model)
problem.add_variable(ServerAllocation(queue, bounds=(1, 10)))

## Block 3: objective and constraints
problem.set_objective(MinimizeCost(server_cost={queue: 10.0}))
problem.add_constraint(UtilizationConstraint(queue, max_value=0.5))

## Block 4: solution
result = problem.solve(seed=42)
```

Obtaining the following output:

```
Optimal servers : 6
Cost            : 60.0
Feasible        : True
LINE evaluations: 10
```

Note that only 10 distinct configurations required a LINE solve: the evaluator caches results at the level of the decoded configuration, so the hundreds of candidate vectors explored by differential evolution collapse onto the 10 possible server counts.

10.3.2 Tutorial 2: Exact sizing by bisection

The sizing problem of Tutorial 1 has a special structure: feasibility is monotone in the decision variable, since adding servers can only decrease utilization. In such cases `BisectionSolver` finds the exact optimum in $O(\log n)$ LINE evaluations. Model and problem definition are identical to Tutorial 1 (`opt_bisection_sizing.py`); only the solution block changes:

```
from line_solver import BisectionSolver

result = BisectionSolver(problem).solve()
```

Obtaining the following output:

```
Optimal servers : 6
Cost            : 60.0
LINE evaluations: 4 (vs hundreds for DE)
```

10.3.3 Tutorial 3: Continuous service rate tuning

We now choose the cheapest service rate, a continuous decision, meeting a response time SLA on an M/M/1 queue. With $\lambda = 3$, the response time is $R = 1/(\mu - 3)$, so $R \leq 0.5$ requires $\mu \geq 5$; the rate is billed at 20

cost units per unit, hence the optimum is $\mu^* = 5$ at cost 100. In the script (`opt_service_rate.py`), the model is an M/M/1 queue built as in Tutorial 1 with initial rate $\mu = 4$, and the problem is declared as:

```
problem = OptimizationProblem(model)
problem.add_variable(ServiceRate(queue, jobs, bounds=(3.5, 8.0)))
problem.set_objective(MinimizeCost(rate_cost={queue: 20.0}))
problem.add_constraint(ResponseTimeConstraint(queue, jobs, max_value=0.5))

result = problem.solve(seed=42, max_iterations=60)
```

Obtaining the following output:

```
Optimal rate    : 5.001 (theory: 5.0)
Cost            : 100.0
Feasible        : True
```

10.3.4 Tutorial 4: Load balancing across heterogeneous servers

An arrival stream of rate $\lambda = 2$ must be split between a fast queue ($\mu_1 = 4$) and a slow queue ($\mu_2 = 2.5$) so that the mean end-to-end response time is minimized. Modeling each queue as M/M/1, the mean number of jobs in the system is $N(p) = \frac{\rho_1}{1-\rho_1} + \frac{\rho_2}{1-\rho_2}$ with $\rho_1 = p\lambda/\mu_1$ and $\rho_2 = (1-p)\lambda/\mu_2$, and by Little's law the system response time is $N(p)/\lambda$. Setting $dN/dp = 0$ yields $p^* \approx 0.743$ with response time 0.4264: the optimal policy does not saturate the fast server. The script (`opt_load_balancing.py`) declares the routing split as the decision variable:

```
problem = OptimizationProblem(model)
problem.add_variable(RoutingProbabilities(jobs, source=source,
                                          targets=[fast, slow]))
problem.set_objective(MinimizeSystemResponseTime(jobs))

result = problem.solve(seed=42, max_iterations=60)
probs = result.getVariableValue('Jobs_routing_from_S')
```

Obtaining the following output:

```
Fraction to Fast: 0.736 (theory: 0.743)
Fraction to Slow: 0.264
System RT       : 0.4248 (theory: 0.4264)
```

The system response time is computed by Little's law from the chain queue lengths and throughput, which remains correct on parallel topologies (Section 10.5).

10.3.5 Tutorial 5: Population sizing in a closed network

An interactive system is modeled as a closed network with a Delay station (mean think time 1) and a processor sharing server (rate 2). We seek the largest number of concurrent users the system can sustain while the server response time stays within an SLA of 2 time units. Feasibility is monotone non-increasing in the population, so bisection applies with the `max_feasible` direction (`opt_population_sizing.py`):

```
model = Network("Interactive")
think = Delay(model, "Think")
server = Queue(model, "AppServer", SchedStrategy.PS)
users = ClosedClass(model, "Users", 1, think)
think.setService(users, Exp(1.0))
server.setService(users, Exp(2.0))
model.addLink(think, server)
model.addLink(server, think)

problem = OptimizationProblem(model)
problem.add_variable(JobPopulation(users, bounds=(1, 50)))
problem.set_objective(MinimizeCost())    # pure feasibility sizing
problem.add_constraint(ResponseTimeConstraint(server, users, max_value=2.0))

result = BisectionSolver(problem, direction='max_feasible').solve()
```

Obtaining the following output:

```
Max users       : 5
Feasible        : True
LINE evaluations: 6
```

10.3.6 Tutorial 6: Robust sizing under workload scenarios

Capacity decisions should often hold under load uncertainty. Here the server pool of Tutorial 1 is sized so that the utilization SLA holds both under the average load ($\lambda = 3$) and under a peak-load scenario ($\lambda = 4.5$). Scenario models share the node and class names of the base model and are registered with `add_scenario`; the solver enforces every constraint on all scenarios (`opt_robust_sizing.py`):

```
base_model, queue = build(3.0)    # average load
peak_model, _ = build(4.5)        # peak load scenario

problem = OptimizationProblem(base_model)
problem.add_variable(ServerAllocation(queue, bounds=(1, 12)))
problem.set_objective(MinimizeCost(server_cost={queue: 10.0}))
problem.add_constraint(UtilizationConstraint(queue, max_value=0.5))
problem.add_scenario(peak_model)

result = BisectionSolver(problem).solve()
```

Obtaining the following output:

```
Robust servers  : 9 (6 for average load only)
Cost            : 90.0
Feasible        : True
```

The average load alone needs 6 servers; the peak scenario pushes the answer to $\lceil 4.5/0.5 \rceil = 9$.

10.3.7 Tutorial 7: Cost-performance tradeoff frontier

How much does tightening the utilization SLA cost? `ParetoSweep` re-solves the sizing problem for a sweep of utilization bounds (epsilon-constraint method) and reports the non-dominated cost frontier, here with each point solved exactly by bisection (`opt_pareto_frontier.py`):

```
sweep = ParetoSweep(
    problem,
    constraint_factory=lambda eps: UtilizationConstraint(queue, max_value=eps),
    epsilons=[0.3, 0.4, 0.5, 0.6, 0.75, 0.9],
    solver='bisection')
sweep.solve()
frontier = sweep.getFrontier()
```

Obtaining the following output:

```
Utilization bound -> optimal cost (servers)
  util <= 0.3  ->  100.0  (10)
  util <= 0.4  ->   80.0  (8)
  util <= 0.5  ->   60.0  (6)
  util <= 0.6  ->   50.0  (5)
  util <= 0.75 ->   40.0  (4)
```

The bound 0.9 is absent from the frontier: it requires the same 4 servers as 0.75, so its point is dominated. The frontier can be plotted with `sweep.plot(xlabel='utilization bound', ylabel='optimal cost')`, which draws only the non-dominated points as a piecewise-constant cost staircase (the epsilon-constraint cost is constant between sampled bounds, so a straight-line interpolation would misrepresent it) and marks the dominated sample points faintly.

10.3.8 Tutorial 8: Decomposing a multi-variable problem

Jointly choosing the number of servers and the service rate couples two variable types. The decomposition workflow splits them into blocks (servers first, then rates), solves the blocks in sequence with the other block's values held fixed, and cycles until the full penalized objective stabilizes, i.e., block coordinate descent (`opt_decomposition.py`):

```
problem = OptimizationProblem(model)
```

```
problem.add_variable(ServerAllocation(queue, bounds=(1, 10)))
problem.add_variable(ServiceRate(queue, jobs, bounds=(1.0, 4.0)))
problem.set_objective(MinimizeCost(server_cost={queue: 10.0},
                                   rate_cost={queue: 20.0}))
problem.add_constraint(ResponseTimeConstraint(queue, jobs, max_value=0.5))

workflow = DecompositionWorkflow(problem)
workflow.auto_decompose()
workflow.set_solver_options(max_iterations=30, popsize=10, seed=42)
result = workflow.solve_sequential(max_cycles=4, tolerance=1e-3)
```

Obtaining the following output:

```
Converged       : True after 1 cycle(s)
Servers         : 10
Rate            : 2.045
Objective       : 140.90
```

As with any coordinate descent scheme on a non-convex problem, the workflow converges to a block-wise optimum that may differ from the joint optimum; a joint solve on this instance finds cost 76. Decomposition remains preferable when the joint dimensionality is intractable.

10.4 Decision variables

Decision variables encode a model parameter into one or more continuous values in $[0, 1]$, which the solvers explore; each variable decodes candidate values back to its native domain and applies them to a per-evaluation model copy, resolving stations and classes by name.

Variable	**Meaning**
`ServerAllocation`	Integer number of servers of a station
`StationReplicas`	Number of identical station replicas, represented as a pooled multiserver station
`ServiceRate`	Exponential service rate of a class at a station
`RoutingProbabilities`	Outgoing routing probabilities of a class at a node (stick-breaking encoding, probabilities sum to one)
`ClassServiceMapping`	Which station among a candidate set serves a class
`JobPopulation`	Population of a closed class
`ClassPriority`	Priority levels or a priority permutation over classes

Variables are constructed with the base model's objects, e.g. `ServerAllocation(queue, bounds=(1, 20))` or `ServiceRate(queue, jobclass, bounds=(0.5, 5.0))`. Integer variables use rounding of the continuous encoding; the solver caches evaluations at the level of the decoded configuration, so candidate vectors that round to the same configuration cost a single LINE solve. Each variable exposes its decoded value in the result under a canonical name, such as `Server_servers`, `Server_Jobs_rate`, `Users_population` or `Jobs_routing_from_S`.

10.5 Objectives and constraints

Three objective families are provided. `MinimizeCost` minimizes a linear cost in the decision variables, with per-server, per-rate and per-replica coefficients (`server_cost`, `rate_cost`, `replica_cost`). `MaximizePerformance` maximizes a weighted combination of throughput and inverse response time, optionally under a budget. `MinimizeSystemResponseTime` minimizes the mean end-to-end response time of a chain, which is the natural objective for load balancing and routing problems. Constraints may be attached to the objective (its last argument) or added to the problem with `add_constraint`; both are enforced identically through a penalty method with weight `penalty_weight` (default 10^6).

Constraint	**Meaning**
`ResponseTimeConstraint`	Station response time $\leq$ bound
`SystemResponseTimeConstraint`	End-to-end (chain) response time $\leq$ bound
`ThroughputConstraint`	Throughput $\geq$ bound
`UtilizationConstraint`	Utilization $\leq$ bound
`BudgetConstraint`	Linear cost of the variables $\leq$ budget

Station-level constraints reference stations and classes by object or name. `SystemResponseTimeConstraint` and `MinimizeSystemResponseTime` use the mean end-to-end response time of the chain; for open chains this is computed by Little's law (jobs in system divided by system throughput), which coincides with the chain sojourn time on serial paths and remains correct on parallel topologies.

10.6 Solvers

10.6.1 Differential evolution

The default solver, `LineOptSolver`, implements differential evolution through `scipy.optimize.differential_evolution`. Options are passed to `problem.solve(...)` directly or through `LineOptSolverOptions`:

Option	**Default**	**Meaning**
`strategy`	`'best1bin'`	DE mutation strategy
`popsize`	`15`	Population size multiplier
`max_iterations`	`100`	Maximum DE generations
`tol`	`0.01`	Convergence tolerance
`time_limit`	`300.0`	Wall-clock limit in seconds, enforced per evaluation; the best solution found so far is returned
`seed`	`None`	Random seed for reproducibility
`penalty_weight`	`1e6`	Constraint violation penalty
`scenario_aggregation`	`'worst'`	Objective aggregation across scenarios
`verbose`	`False`	Progress reporting

10.6.2 Analytic gradients for product-form networks

The `optimizer` option selects the backend: `'evolution'` (the default, differential evolution), `'gradient'` (`scipy.optimize.minimize` with L-BFGS-B and a deterministic multistart controlled

by `gradient_restarts`, default 4), or `'auto'`, which takes the gradient path only when every decision variable is continuous, that is of type service rate or routing. The gradient path minimizes the same penalized scalar objective as the evolutionary path, so constraints need no separate handling.

The Jacobian first tries an analytic assembly and otherwise falls back to central finite differences in the encoded space with step `fd_step` (default 10^{-6}), reverting to a one-sided difference at a bound where the two-sided value is not finite. The analytic path costs no extra LINE solves: it composes three exact pieces, namely the derivative of each performance measure with respect to the service rates, the derivative of the penalized scalar with respect to each metric (obtained by arithmetic-only finite differences in metric space, with no solving), and the derivative of the rate with respect to the encoded variable, which is linear in the decode.

The performance derivatives come from `compute_model_sensitivities` in `line_solver/opt/sensitivity.py`, which is invoked once per model evaluation and attached to the `EvaluationResult` as its sensitivities field. It maps each metric kind (`'RespT'`, `'QLen'`, `'Tput'`, `'Util'`) and metric key (a station-class pair, a station name alone for `'Util'`) to the derivative with respect to each parameter key, the rate of a class at a station. Two model families are covered:

- Open product-form networks of single-server queueing stations and infinite-server delay stations, which decouple. The derivatives follow in closed form from $R_r = D_r/(1-U)$ and $Q_r = \rho_r/(1-U)$ by the quotient rule, and the routine declines when a station is unstable or multiserver.
- Closed single-server product-form networks with unit visit ratios, through the exact differentiated MVA primitive `pfqn_sens`, which yields the full cross-station Jacobian since closed networks couple the stations. Utilization derivatives are aggregated per station to match the utilizations reported by the evaluation result.

Mixed, multiserver, and non-unit-visit topologies are declined, as is any failure of the sensitivity computation, in which case the optimizer silently uses finite differences. The analytic assembly is additionally declined, and finite differences used instead, when the problem carries more than one workload scenario or when any decision variable is not a service-rate variable.

10.6.3 Exact sizing by bisection

For the common sizing pattern (a single integer variable whose feasibility is monotone, such as adding servers to meet an SLA), `BisectionSolver` finds the exact optimum in $O(\log n)$ LINE solves instead of hundreds of DE evaluations:

```
from line_solver import BisectionSolver
result = BisectionSolver(problem).solve()    # smallest feasible value
result = BisectionSolver(problem, direction='max_feasible').solve()
```

The `min_feasible` direction assumes feasibility is non-decreasing in the variable (server sizing); `max_feasible` assumes it is non-increasing (population sizing).

10.6.4 Workload scenarios

Robust designs can be obtained by registering scenario models, i.e., copies of the network with the same node and class names but different parameters (e.g., a peak-load arrival rate). Constraints are enforced on the base model and on every scenario; the objective is aggregated across scenarios by worst case (default) or weighted mean:

```
problem.add_scenario(peak_load_model, weight=1.0)
result = problem.solve(scenario_aggregation='worst')
```

10.6.5 Cost-performance tradeoffs

`ParetoSweep` characterizes the tradeoff between cost and a service-level bound using the epsilon-constraint method: the problem is re-solved for a sweep of bounds and the feasible, non-dominated points form the frontier. The constraint factory maps a bound to the constraint to enforce at that bound, and the solver argument selects bisection or differential evolution for each point, as illustrated in Tutorial 10.3.7.

10.7 Decomposition workflows

Problems mixing several variable types can be decomposed into blocks solved in sequence, with the values fixed by previously solved blocks propagated into later ones (block coordinate descent). Cycling repeats the sequence until the full penalized objective stabilizes:

```
from line_solver import DecompositionWorkflow
workflow = DecompositionWorkflow(problem)
workflow.auto_decompose()              # group variables by type
workflow.set_solver_options(max_iterations=30, seed=42)
result = workflow.solve_sequential(max_cycles=5, tolerance=1e-3)
print(result.converged, result.final_variable_values)
```

The automatic decomposition orders blocks as: server allocation, station replicas, service rates, job populations, class priorities, routing, class mapping. Custom blocks and dependencies can be declared with `add_subproblem(name, variables, after=[...])`. As with any coordinate descent scheme on a non-convex problem, the workflow converges to a block-wise optimum that may differ from the joint optimum; the joint solve remains preferable when its dimensionality is tractable.

10.8 Results and metrics

All solvers return an `OptimizationResult` carrying the decoded variable values (accessed by name with `getVariableValue`), the objective value, the feasibility flag, the per-constraint violations, and statistics on the number of model evaluations, the solution time and the termination rea-

son. Model evaluations are exposed through `EvaluationResult`, whose accessors include per-station metrics (`getResponseTime`, `getThroughput`, `getUtilization`, `getQueueLength`) and end-to-end metrics (`getSystemResponseTime`, `getSystemThroughput`). Decomposition workflows return a `WorkflowResult` instead, with the final variable values (`final_variable_values`), the final objective, the convergence flag and the number of cycles completed.

10.9 Sensitivity analysis and fast parameter updates

Performance and reliability studies routinely vary one or more model parameters, either to measure the sensitivity of the results or to drive an optimization loop. This section collects the facilities that make such sweeps cheap: exact sensitivities where they are available in closed form, a symbolic backend, and the refresh and low-level update routines that avoid recompiling the whole internal representation at every iteration.

10.9.1 Performance sensitivities

For product-form models the derivative of a performance measure with respect to a service rate is available in closed form, so a numerical sweep is not needed. The `get_sensitivity_table` method returns one row per station and class with the columns `dTput_dRate`, `dRespT_dRate`, `dQLen_dRate` and `dUtil_dRate`, and a second output returns the underlying structure. Solvers that cannot differentiate analytically fall back to a finite difference, whose scheme is selected by the `scheme` argument.

```
sens_table = SolverMVA(model).get_sensitivity_table()
sens_table = SolverCTMC(model).get_sensitivity_table(scheme='central')
```

10.9.2 Symbolic analysis backend

Some analyses are exact only in symbolic form, such as the stationary distribution of a Markov chain as a rational function of its transition rates. The backend is selected by `line_solver.api.sym.set_backend`, whose accepted values are `'auto'` (the default: the native engine, and an external service only where there is none), `'sage'` (always the service) and `'native'`. A service is located from an explicit URL, from the `LINE_SAGE_URL` environment variable, from a service already listening on port 8085 or 8080, or from a container started by LINE on a free port. Expressions are exchanged as plain infix strings with exact rational coefficients, so the exchange introduces no rounding.

```
solver = SolverLN(model, lambda layer: SolverMVA(layer))
sens_table = solver.get_sensitivity_table()
sens_table.attrs['method']   # 'exact', 'fd', or 'mixed'
```

Exact parametric sensitivities follow from the symbolic solution by the chain rule, which leaves only the map from the parameter to the affected rates to be differenced; a parameter that reshapes an event rather than scaling it invalidates the chain rule and is refused with an explanatory error rather than approximated silently.

Table 10.1: Functions for direct modification of `NetworkStruct` objects

Function	Description
`sn_set_service`	Sets service rate at a specific station and class. Optional parameters: squared coefficient of variation (default 1.0) and auto-refresh flag.
`sn_set_arrival`	Sets arrival rate for a class at the Source station. Automatically locates the Source and updates its rate.
`sn_set_population`	Sets the number of jobs for a closed class. Automatically recalculates `nclosedjobs`.
`sn_set_servers`	Sets the number of servers at a station.
`sn_set_priority`	Sets the priority for a class.
`sn_set_routing_prob`	Sets a routing probability between two stateful node-class pairs.
`sn_set_routing`	Sets the full routing matrix.
`sn_set_fork_fanout`	Sets fork fanout parameters.
`sn_set_service_batch`	Sets service parameters for batch processing.

The mean-field drift can likewise be exported as expressions and differentiated to give the Jacobian, provided the drift is differentiable: the methods that scale service rates by $\min(n_i, S_i)$ are not, and a smoothed variant is selected through `options.config.pstar`.

10.9.3 Fast parameter update

Successive calls to `get_struct()` return a cached copy of the internal representation, which must be refreshed after a parameter change. Recomputing all of it is wasteful, since several fields require a dedicated algorithm, so LINE exposes one refresh routine per group of fields: `refresh_capacity` after a capacity change, `refresh_chains` after a routing change, `refresh_priorities`, `refresh_scheduling` and `refresh_processes` after a change of service or arrival process.

```
queue.set_service(class1, Exp(1.0))
model.refresh_struct()
```

10.9.4 Direct modification of NetworkStruct objects

For maximum performance in an optimization loop, the low-level `sn_set_*` functions modify a `NetworkStruct` in place, bypassing the model object and the refresh routines altogether. They are listed in Table 10.1; because they write the structure directly, the caller is responsible for keeping the derived fields consistent.

10.9.5 Refreshing a network topology with non-probabilistic routing

The `reset_network` function removes the nodes that LINE adds automatically, such as the `ClassSwitch` nodes introduced when the model is linked, and returns the remaining user-defined nodes. It should be called before re-linking a model whose routing is not purely probabilistic.

10.9.6 Saving a network object before a change

A model object is a handle, so assigning it to another variable does not copy it and a later change is seen through both names. Use the copy method to take a snapshot before modifying a model in a sweep.

```
model_by_ref = model; model_by_ref.set_name('myModel1')
model_by_copy = model.copy(); model_by_copy.set_name('myModel2')
```

10.10 Current limitations

- `StationReplicas` represents N replicas as one multiserver station with N times the servers, pooling the replicas' queues; it does not create distinct nodes.
- `ClassServiceMapping` and `RoutingProbabilities` reconstruct default routing (uniform split over physical links) for the classes they do not control; models with hand-crafted non-default routing for other classes should apply routing variables last.
- Percentile (tail latency) constraints are not yet supported; they depend on response time distribution support in the LINE API.
- Evaluations are exact only up to the accuracy of the LINE solver selected by `SolverAuto` for the model at hand.
- Frontier plotting (`ParetoSweep.plot`) is available only in the Python codebase; the MATLAB and Java sweeps return the frontier points for the caller to plot.

Conclusion

LINE provides a common framework for specifying, solving, and validating queueing networks and related stochastic models across four language editions. The methods in this manual range from exact product-form and state-space algorithms to matrix-analytic approximations, fluid models, simulation, inference, and optimization. Their value depends on making solver assumptions and language support explicit, reporting reproducible numerical evidence, and selecting an analysis whose cost is commensurate with the research question. Continued development should therefore preserve cross-language parity while strengthening reproducibility, scalable state representations, and integration with empirical data. These objectives determine the long-term utility of LINE as both a research platform and an engineering tool for the performance analysis of networked and service-based systems.

Bibliography

[1] M. Ajmone Marsan, G. Conte, and G. Balbo. A class of generalized stochastic petri nets for the performance evaluation of multiprocessor systems. *ACM Trans. Comput. Syst.*, 2(2):93–122, May 1984.

[2] I. F. Akyildiz and J. C. Strelen. Moment analysis for load-dependent mixed product form queueing networks. *IEEE Trans. Commun.*, 39(6):828–832, 1991.

[3] D. F. Anderson. A modified next reaction method for simulating chemical systems with time dependent propensities and delays. *The Journal of chemical physics*, 127(21), 2007.

[4] S. Asmussen and M. Bladt. Point processes with finite-dimensional conditional probabilities. *Stochastic Processes and their Applications*, 82(1):127–142, 1999.

[5] S. Asmussen and J. R. Møller. Calculation of the steady state waiting time distribution in GI/PH/c and MAP/PH/c queues. *Queueing Systems*, 37(1):9–29, 2001.

[6] S. Balsamo. Product form queueing networks. In Günter Haring, Christoph Lindemann, and Martin Reiser, editors, *Performance Evaluation: Origins and Directions*, volume 1769 of *Lecture Notes in Computer Science*, pages 377–401. Springer, 2000.

[7] S. Balsamo, A. Marin, and I. Stojic. Computation of the normalising constant for product-form models of distributed systems with synchronisation. *Future Generation Computer Systems*, 111:475–490, 2020.

[8] C. Bandi, D. Bertsimas, and N. Youssef. Robust queueing theory. *Operations Research*, 63(3):676–700, 2015.

[9] Y. Bard. Some extensions to multiclass queueing network analysis. In *Proc. of the 3rd Int'l Symp. on Model. and Performance Evaluation of Comp. Syst.*, pages 51–62, 1979.

[10] O. Barndorff-Nielsen and D.R. Cox. Edgeworth and saddle-point approximations with statistical applications. *Journal of the Royal Statistical Society. Series B*, 41(3):279–312, 1979.

[11] M. Bertoli, G. Casale, and G. Serazzi. The JMT simulator for performance evaluation of non-product-form queueing networks. In *Proc. of the 40th Annual Simulation Symposium (ANSS)*, pages 3–10, 2007.

[12] A. L. Bertozzi and J. McKenna. Multidimensional residues, generating functions, and their application to queueing networks. *SIAM Review*, 35(2):239–268, 1993.

[13] D. Bertsimas, D. Gamarnik, and J. N. Tsitsiklis. Performance of multiclass markovian queueing networks via piecewise linear lyapunov functions. *The Annals of Applied Probability*, 11(4):1384–1428, 2001.

[14] D. Bertsimas, I. Ch. Paschalidis, and J. N. Tsitsiklis. Optimization of multiclass queueing networks: Polyhedral and nonlinear characterizations of achievable performance. *The Annals of Applied Probability*, 4(1):43–75, 1994.

[15] D. Bini, B. Meini, S. Steffé, J. F. Pérez, and B. Van Houdt. Smcsolver and q-mam: tools for matrix-analytic methods. *SIGMETRICS Performance Evaluation Review*, 39(4):46, 2012.

[16] Alexander Birman and Yaakov Kogan. Asymptotic evaluation of closed queueing networks with many stations. *Communications in Statistics. Stochastic Models*, 8(3):543–563, 1992.

[17] Mogens Bladt and Bo Friis Nielsen. *Matrix-Exponential Distributions in Applied Probability*. Springer, New York, 2017.

[18] G. Bolch, S. Greiner, H. de Meer, and K. S. Trivedi. *Queueing Networks and Markov Chains*. Wiley, 2006.

[19] T. Bonald and A. Proutiere. Insensitive bandwidth sharing in data networks. *Queueing Systems*, 44(1):69–100, 2003.

[20] A. B. Bondi and W. Whitt. The influence of service-time variability in a closed network of queues. *Perform. Eval.*, 6:219–234, 1986.

[21] S. C. Bruell, G. Balbo, and P. V. Afshari. Mean value analysis of mixed, multiple class BCMP networks with load dependent service stations. *Performance Evaluation*, 4:241–260, 1984.

[22] J. P. Buzen. Computational algorithms for closed queueing networks with exponential servers. *Communications of the ACM*, 16(9):527–531, 1973.

[23] G. Casale. CoMoM: Efficient class-oriented evaluation of multiclass performance models. *IEEE Trans. Software Engineering*, 35(2):162–177, 2009.

[24] G. Casale. Accelerating performance inference over closed systems by asymptotic methods. In *Proc. of ACM SIGMETRICS*. ACM Press, 2017.

[25] G. Casale. Integrated Performance Evaluation of Extended Queueing Network Models with Line. In *2020 Winter Simulation Conference (WSC)*, pages 2377–2388. IEEE, dec 2020.

[26] G. Casale, V. De Nitto Personé, and E. Smirni. QRF: An optimization-based framework for evaluating complex stochastic networks. *ACM Transactions on Modeling and Computer Simulation*, 26(3):15:1–15:24, 2016.

[27] G. Casale and N. Gast. Performance analysis methods for list-based caches with non-uniform access. *IEEE/ACM Transactions on Networking*, 29(2):651–664, 2021.

[28] G. Casale and P. G. Harrison. AutoCAT: Automated product-form solution of stochastic models. In *Matrix-Analytic Methods in Stochastic Models*, volume 27 of *Springer Proceedings in Mathematics & Statistics*, pages 57–85. Springer, 2013.

[29] G. Casale, P.G. Harrison, and O.W. Hong. Facilitating load-dependent queueing analysis through factorization. *Perform. Eval.*, 2021.

[30] G. Casale, N. Mi, L. Cherkasova, and E. Smirni. Dealing with burstiness in multi-tier applications: Models and their parameterization. *IEEE Transactions on Software Engineering*, 37(5):1040–1053, 2011.

[31] G. Casale, Richard R. Muntz, and Giuseppe Serazzi. Geometric bounds: A noniterative analysis technique for closed queueing networks. *IEEE Trans. Computers*, 57(6):780–794, 2008.

[32] G. Casale, J. F. Pérez, and W. Wang. QD-AMVA: Evaluating systems with queue-dependent service requirements. In *Proceedings of IFIP PERFORMANCE*, 2015.

[33] G. Casale and E. Smirni. MAP-AMVA: Approximate mean value analysis of bursty systems. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 409–418, 2009.

[34] G. Casale and M. Tribastone. Fluid analysis of queueing in two-stage random environments. In *Proceedings of QEST 2011*, 2011.

[35] G. Casale, M. Tribastone, and P. G. Harrison. Blending randomness in closed queueing network models. *Perform. Eval.*, 82:15–38, 2014.

[36] G. Casale, E. Z. Zhang, and E. Smirni. KPC-Toolbox: Best recipes for automatic trace fitting using Markovian arrival processes. *Performance Evaluation*, 67(9):873–896, 2010.

[37] K. M. Chandy, U. Herzog, and L. Woo. Parametric analysis of queuing networks. *IBM Journal of Research and Development*, 19(1):36–42, 1975.

[38] K. M. Chandy and M. S. Lakshmi. An approximation technique for queueing networks with preemptive priority queues. Technical report, Department of Computer Sciences, University of Texas at Austin, 1983.

[39] K. M. Chandy and D. Neuse. Linearizer: A heuristic algorithm for queuing network models of computing systems. *Commun. ACM*, 25(2):126–134, 1982.

[40] G. L. Choudhury, K. K. Leung, and W. Whitt. Calculating normalization constants of closed queuing networks by numerically inverting their generating functions. *J. ACM*, 42(5):935–970, 1995.

[41] W.-M. Chow. Approximations for large scale closed queueing networks. *Perform. Eval*, 3(1):1–12, 1983.

[42] W.-M. Chow. Approximations for large scale closed queueing networks. *Performance Evaluation*, 3(1):1–12, 1983.

[43] F. Ciucu and S. Mehri. On the distribution of sojourn times in tandem queues. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 9(2):1–34, 2025. Article 27, ACM SIGMETRICS 2025.

[44] C. Comte and J.-P. Dorsman. Pass-and-swap queues. *Queueing Systems*, 98(3):275–331, 2021.

[45] A. E. Conway. *Fast Approximate Solution of Queueing Networks with Multi-Server Chain-Dependent FCFS Queues*, pages 385–396. Springer US, Boston, MA, 1989.

[46] A. E. Conway and N. D. Georganas. RECAL - A new efficient algorithm for the exact analysis of multiple-chain closed queueing networks. *J. ACM*, 33(4):768–791, 1986.

[47] P. J. Courtois. *Decomposability: queueing and computer system applications*. Academic Press, New York, 1977.

[48] S. Coury and P. G. Harrison. Asymptotic properties of queuing networks. *IEE Proceedings - Computers and Digital Techniques*, 144(5):247–254, 1997.

[49] H. Daduna. Busy periods for subnetworks in stochastic networks: Mean value analysis. *Journal of the ACM*, 35(3):668–674, 1988.

[50] H.E. Daniels. Saddlepoint approximations in statistics. *The Annals of Mathematical Statistics*, 25(4):631–650, 1954.

[51] E. de Souza e Silva, S. S. Lavenberg, and R. R. Muntz. A clustering approximation technique for queueing network models with a large number of chains. *IEEE Trans. Computers*, C-35(5):419–430, 1986.

[52] E. de Souza e Silva and R. R. Muntz. A note on the computational cost of the linearizer algorithm for queueing networks. *IEEE Trans. Computers*, 39(6):840–842, 1990.

[53] R.-A. Dobre, Z. Niu, and G. Casale. Approximating fork-join systems via mixed model transformations. In *Companion of the 15th ACM/SPEC International Conference on Performance Engineering*, ICPE '24 Companion, page 273–280, New York, NY, USA, 2024. Association for Computing Machinery.

[54] L. W. Dowdy, B. M. Carlson, A. T. Krantz, and S. K. Tripathi. Single-class bounds of multi-class queuing networks. *Journal of the ACM*, 39(1):188–213, 1992.

[55] D. L. Eager. *Bounding Algorithms for Queueing Network Models of Computer Systems*. Tech. rept. csrg-156, University of Toronto, Toronto, Ontario, Canada, 1984.

[56] D. L. Eager and J. N. Lipscomb. The AMVA priority approximation. *Perform. Eval.*, 8(3):173–193, 1988.

[57] M. Fidler and A. Rizk. A guide to the stochastic network calculus. *IEEE Communications Surveys and Tutorials*, 17(1):92–105, 2015.

[58] B. L. Fox and P. W. Glynn. Computing Poisson probabilities. *Communications of the ACM*, 31(4):440–445, 1988.

[59] G. Franks, T. Al-Omari, M. Woodside, O. Das, and S. Derisavi. Enhanced modeling and solution of layered queueing networks. *IEEE Trans. Software Engineering*, 35(2):148–161, 2009.

[60] G. Franks, P. Maly, C. M. Woodside, D. C. Petriu, A. Hubbard, and M. Mroz. *Layered Queueing Network Solver and Simulator User Manual*, 2012.

[61] G. Franks, P. Maly, M. Woodside, D. C. Petriu, Alex Hubbard, and Martin Mroz. *Layered Queueing Network Solver and Simulator User Manual*. Carleton University, January 2013.

[62] R. G. Franks. *Performance Analysis of Distributed Server Systems*. PhD thesis, Department of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, Canada, December 1999.

[63] E. J. Friedman and S. G. Henderson. Fairness and efficiency in web server protocols. In *Proc. ACM SIGMETRICS*, pages 229–237, 2003.

[64] N. Gast. Expected values estimated via mean-field approximation are 1/n-accurate. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 1(1):17:1–17:26, 2017.

[65] N. Gast and B. Van Houdt. Transient and steady-state regime of a family of list-based cache replacement algorithms. *Queueing Syst*, 83(3-4):293–328, 2016.

[66] D. P. Gaver, P. A. Jacobs, and G. Latouche. Finite birth-and-death models in randomly changing environments. *Advances in Applied Probability*, 16(4):715–731, 1984.

[67] Alexander I. Gerasimov. On normalizing constants in multiclass queueing networks. *Operations Research*, 43(4):704–711, 1995.

[68] D. T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *J. Phys. Chem.*, 81(25):2340–2361, 1977.

[69] M. C. Guenther, A. Stefanek, and J. T. Bradley. Moment closures for performance models with highly non-linear rates. In *Computer Performance Engineering (EPEW/UKPEW 2012)*, volume 7587 of *Lecture Notes in Computer Science*, pages 32–47. Springer, 2013.

[70] E. Hairer and G. Wanner. *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*, volume 14 of *Springer Series in Computational Mathematics*. Springer, 2nd edition, 1996.

[71] A. Harel, S. Namn, and J. Sturm. Simple bounds for closed queueing networks. *Queueing Systems*, 31(1-2):125–135, 1999.

[72] Q.-M. He. The versatility of MMAP[K] and the MMAP[K]/G[K]/1 queue. *Queueing Systems*, 38(4):397–418, 2001.

[73] P. Heidelberger and K. Trivedi. Queueing network models for parallel processing with asynchronous tasks. *IEEE Trans. Computers*, 100(11):1099–1109, 1982.

[74] A. Heindl, G. Horvath, and K. Gross. Explicit inverse characterizations of acyclic maps of second order. In *European Performance Engineering Workshop (EPEW)*, pages 108–122, 2006.

[75] G. Horváth and M. Telek. Sojourn times in fluid queues with independent and dependent input and output processes. *Performance Evaluation*, 79:160–181, 2014.

[76] G. Horváth and M. Telek. Butools 2: A rich toolbox for markovian performance evaluation. In *Proc. of VALUETOOLS*, pages 137–142, ICST, Brussels, Belgium, Belgium, 2017. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[77] Gábor Horváth, Illés Horváth, and Miklós Telek. Numerical inverse laplace transformation using concentrated matrix exponential distributions. *Performance Evaluation*, 137:102067, 2020.

[78] C. T. Hsieh and S. S. Lam. PAM - a noniterative approximate solution method for closed multichain queueing networks. *ACM SIGMETRICS Performance Evaluation Review*, 16(1):261–269, 1988.

[79] M. Huber and J. Law. Fast approximation of the permanent for very dense problems. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 681–689, 2008.

[80] A. Jensen. Markoff chains as an aid in the study of markoff processes. *Scandinavian Actuarial Journal*, 1953(sup1):87–91, 1953.

[81] J. L. Jensen. Saddlepoint expansions for sums of Markov dependent variables on a continuous state space. *Probability Theory and Related Fields*, 89:181–199, 1991.

[82] K. Kant. MVA approximations for SJN scheduling. *Performance Evaluation*, 15(1):41–61, 1992.

[83] J. S. Kaufman. Blocking in a shared resource environment. *IEEE Transactions on Communications*, 29(10):1474–1481, 1981.

[84] F. P. Kelly. Loss networks. *Annals of Applied Probability*, 1(3):319–378, 1991.

[85] S. Kijima and T. Matsui. Approximate/perfect samplers for closed Jackson networks. In *Proceedings of the Winter Simulation Conference*, pages 862–868, 2005.

[86] C. Knessl and C. Tier. Asymptotic expansions for large closed queueing networks with multiple job classes. *IEEE Trans. Computers*, 41(4):480–488, 1992.

[87] Y. M. Ko and J. Pender. Diffusion limits for the $(\text{MAP}_t/\text{Ph}_t/\infty)^n$ queueing network. *Operations Research Letters*, 45(3):248–253, 2017.

[88] J. R. Koury, D. F. McAllister, and W. J. Stewart. Iterative methods for computing stationary distributions of nearly completely decomposable Markov chains. *SIAM Journal on Algebraic Discrete Methods*, 5(2):164–186, 1984.

[89] D.D. Kouvatsos. Entropy maximisation and queueing network models. *Annals of Operations Research*, 48:63–126, 1994.

[90] A. E. Krzesinski. Multiclass queueing networks with state-dependent routing. *Performance Evaluation*, 7(2):125–143, 1987.

[91] J. Kuck, T. Dao, H. Rezatofighi, A. Sabharwal, and S. Ermon. Approximating the permanent by sampling from adaptive partitions. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[92] S. S. Lavenberg. A perspective on queueing models of computer performance. *Perform. Eval.*, 10(1):53–76, 1989.

[93] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative System Performance*. Prentice-Hall, 1984.

[94] E. Lelarasmee, A. E. Ruehli, and A. L. Sangiovanni-Vincentelli. The waveform relaxation method for time-domain analysis of large scale integrated circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 1(3):131–145, 1982.

[95] Z. Li and G. Casale. Matrix network analyzer: A new decomposition algorithm for phase-type queueing networks (work in progress paper). In *Companion of the 15th ACM/SPEC International Conference on Performance Engineering*, ICPE '24 Companion, page 34–39, New York, NY, USA, 2024. Association for Computing Machinery.

[96] D. V. Lindley. The theory of queues with a single server. *Proceedings of the Cambridge Philosophical Society*, 48(2):277–289, 1952.

[97] Q. Liu and D. A. Pierce. A note on Gauss-Hermite quadrature. *Biometrika*, 81(3):624–629, 1994.

[98] Zhen Liu. Performance analysis of stochastic timed Petri nets using linear programming approach. *IEEE Transactions on Software Engineering*, 24(11):1014–1030, 1998.

[99] Zhen Liu, Laura Wynter, Cathy H. Xia, and Li Zhang. Parameter inference of queueing models for IT systems using end-to-end measurements. *Performance Evaluation*, 63(1):36–60, 2006.

[100] A. Lolos, C. Alexopoulos, D. Goldsman, K. D. Dingeç, A. C. Mokashi, and J. R. Wilson. A fixed-sample-size method for estimating steady-state quantiles. In *Proceedings of the Winter Simulation Conference*, pages 457–468, 2023.

[101] A. Lolos, C. Alexopoulos, D. Goldsman, K. D. Dingeç, A. C. Mokashi, and J. R. Wilson. A fixed-sample-size procedure for estimating steady-state quantiles based on independent replications. In *Proceedings of the Winter Simulation Conference*, pages 223–234, 2025.

[102] J. Lüthi and G. Haring. Mean value analysis for queueing network models with intervals as input parameters. *Performance Evaluation*, 32(3):185–215, 1998.

[103] S. Majumdar and C. M. Woodside. Robust bounds and throughput guarantees for closed multiclass queueing networks. *Performance Evaluation*, 32(2):101–136, 1998.

[104] D. Manjunath and Biplab Sikdar. Integral expressions for the numerical evaluation of product form expressions over irregular multidimensional integer spaces. Manuscript, Indian Institute of Technology Bombay and Rensselaer Polytechnic Institute.

[105] Andrea Marin, Samuele Rota Bulò, and Simonetta Balsamo. A numerical algorithm for the decomposition of cooperating structured Markov processes. In *Proc. of the 20th IEEE MASCOTS*, pages 401–410. IEEE, 2012.

[106] H. Masuyama and T. Takine. Sojourn time distribution in a MAP/M/1 processor-sharing queue. *Oper. Res. Lett.*, 31(6):406–412, 2003.

[107] J. McKenna and D. Mitra. Integral representations and asymptotic expansions for closed markovian queueing networks: Normal usage. *Bell Syst. Tech. J.*, 61(5):661–683, May 1982.

[108] J. McKenna and D. Mitra. Asymptotic expansions and integral representations of moments of queue lengths in closed markovian networks. *J. ACM*, 31(2):346–360, April 1984.

[109] Carl D. Meyer. Stochastic complementation, uncoupling markov chains, and the theory of nearly reducible systems. *SIAM Review*, 31(2):240–272, 1989.

[110] A. S. Miner and G. Ciardo. Efficient reachability set generation and storage using decision diagrams. In *Proc. of ICATPN, LNCS 1639*, pages 6–25, 1999.

[111] D. Mitra and J. McKenna. Asymptotic expansions for closed markovian networks with state-dependent service rates. *J. ACM*, 33(3):568–592, July 1986.

[112] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[113] J. A. Morrison. Asymptotic analysis of a large closed queueing network with discriminatory processor sharing. *Queueing Systems*, 9(1-2):191–214, 1991.

[114] D. Neuse and K. M. Chandy. SCAT: A heuristic algorithm for queueing network models of computing systems. *ACM SIGMETRICS Perform. Eval. Rev.*, 10(3):59–79, 1981.

[115] M. F. Neuts. *Matrix-geometric solutions in stochastic models: an algorithmic approach*. Johns Hopkins University Press, 1981.

[116] M. Nuyens and A. Wierman. The foreground-background queue: A survey. *Performance Evaluation*, 65(3-4):286–307, 2008.

[117] T. J. Ott. The sojourn-time distribution in the m/g/1 queue with processor sharing. *Journal of Applied Probability*, 21(2):360–378, 1984.

[118] S. Palomo and J. Pender. Learning the tandem network Lindley recursion. In *Proceedings of the Winter Simulation Conference*, 2021.

[119] I. Perez and G. Casale. Variational inference for markovian queueing networks. *Advances in Applied Probability*, 53(3):687–715, 2021.

[120] J. F. Pérez and G. Casale. Assessing SLA compliance from Palladio component models. In *Proceedings of the 2nd MICAS*, 2013.

[121] J. F. Pérez and G. Casale. Line: Evaluating software applications in unreliable environments. *IEEE Trans. Reliability*, 66(3):837–853, Sept 2017.

[122] J. F. Pérez, J. Van Velthoven, and B. Van Houdt. Q-mam: A tool for solving infinite queues using matrix-analytic methods. In *Proc. of the 3rd International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS)*, 2008.

[123] Juan F. Pérez, Giuliano Casale, and Sergio Pacheco-Sanchez. Estimating computational requirements in multi-threaded applications. *IEEE Trans. Software Eng.*, 41(3):264–278, 2015.

[124] Tuan Phung-Duc, Hiroyuki Masuyama, Shoji Kasahara, and Yutaka Takahashi. A simple algorithm for the rate matrices of level-dependent qbd processes. In *Proc. of QTNA*, pages 46–52. ACM, 2010.

[125] M. Reiser. Mean-value analysis and convolution method for queue-dependent servers in closed queueing networks. *Perform. Eval.*, 1:7–18, 1981.

[126] M. Reiser and H. Kobayashi. Queueing networks with multiple closed chains: Theory and computational algorithms. *IBM J. Res. Dev.*, 19(3):283–294, 1975.

[127] M. Reiser and S. Lavenberg. Mean-value analysis of closed multichain queuing networks. *J. ACM*, 27:313–322, 1980.

[128] A. Riska and E. Smirni. MAMSolver: A matrix analytic methods tool. In *Proc. of the 12th Int. Conf. on Modelling Techniques and Tools (TOOLS)*, volume 2324 of *LNCS*, pages 205–211, 2002.

[129] A. Riska and E. Smirni. ETAQA solutions for infinite Markov processes with repetitive structure. *INFORMS Journal on Computing*, 19(2):215–228, 2007.

[130] T. G. Robertazzi. *Computer Networks and Systems*. Springer, 2000.

[131] J. W. Roberts. A service system with heterogeneous user requirements. In *Performance of Data Communications Systems and Their Applications*, pages 423–431. North-Holland, 1981.

[132] J. A. Rolia and K. C. Sevcik. The method of layers. *IEEE Trans. Software Engineering*, 21(8):689–700, August 1995.

[133] Keith W. Ross and Jie Wang. Monte carlo summation applied to product-form loss networks. *Probability in the Engineering and Informational Sciences*, 6(3):323–348, 1992.

[134] J. Ruuskanen, T. Berner, K.-E. Årzén, and A. Cervin. Improving the mean-field fluid model of processor sharing queueing networks for dynamic performance models in cloud computing. *Perform. Evaluation*, 151:102231, 2021.

[135] H. J. Ryser. *Combinatorial Mathematics*. Number 14 in Carus Mathematical Monographs. Mathematical Association of America, 1963.

[136] R. A. Sahner and K. S. Trivedi. Performance and reliability analysis using directed acyclic graphs. *IEEE Transactions on Software Engineering*, 13(10):1105–1114, 1987.

[137] C. H. Sauer. Computational algorithms for state-dependent queueing networks. *ACM Trans. Comput. Syst.*, 1(1):67–92, 1983.

[138] P. J. Schweitzer. Approximate analysis of multiclass closed networks of queues. In *Proc. of the Int'l Conf. on Stoch. Control and Optim.*, pages 25–29, Amsterdam, 1979.

[139] P. J. Schweitzer, G. Serazzi, and M. Broglia. A queue-shift approximation technique for product-form queueing networks. In *Computer Performance Evaluation (Tools'98)*, volume 1469 of *Lecture Notes in Computer Science*, pages 267–279. Springer, 1998.

[140] A. Seidmann, P. J. Schweitzer, and S. Shalev-Oren. Computerized closed queueing network models of flexible manufacturing systems: A comparative evaluation. *Large Scale Systems*, 12:91–107, 1987.

[141] K. Sevcik. Priority scheduling disciplines in queuing network models of computer systems. In *IFIP Congress*, 1977.

[142] Matthew Sheldon, Daphne Tuncer, and Giuliano Casale. TBI: Transient hierarchical modeling of large-scale vehicle sharing systems. *IEEE Transactions on Intelligent Transportation Systems*, 2025. In submission.

[143] R. Sinkhorn. A relationship between arbitrary positive matrices and doubly stochastic matrices. *The Annals of Mathematical Statistics*, 35(2):876–879, 1964.

[144] Simon Spinner, Giuliano Casale, Fabian Brosig, and Samuel Kounev. Evaluating approaches to resource demand estimation. *Performance Evaluation*, 92:51–71, 2015.

[145] A. Stefanek, R. A. Hayden, and J. T. Bradley. A new tool for the performance analysis of massively parallel computer systems. In *8th Workshop on Quantitative Aspects of Programming Languages (QAPL 2010)*, volume 28 of *EPTCS*, pages 159–181, 2010.

[146] W. J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, 1994.

[147] R. Suri, S. K. Sahu, and M. Vernon. Approximate mean value analysis for closed queuing networks with multiple-server stations. In *Proc. of the Industrial Engineering Research Conference*, pages 1–6, 2007.

[148] H. Tahilramani, D. Manjunath, and S.K. Bose. Approximate analysis of open network of ge/ge/m/n queues with transfer blocking. *Proc. of the 7th Int. Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 164–171, 1999.

[149] Y. Takahashi. A lumping method for numerical calculations of stationary distributions of Markov chains. Technical Report B-18, Department of Information Sciences, Tokyo Institute of Technology, Tokyo, Japan, June 1975.

[150] Y. C. Tay and R. Suri. Error bounds for performance prediction in queuing networks. *ACM Transactions on Computer Systems*, 3(4):227–254, 1985.

[151] D. Towsley. Queuing network models with state-dependent routing. *Journal of the ACM*, 27(2):323–337, 1980.

[152] M. Tribastone. A fluid model for layered queueing networks. *IEEE Trans. Software Engineering*, 39(6):744–756, June 2013.

[153] K. S. Trivedi and A. Bobbio. *Reliability and Availability Engineering: Modeling, Analysis, and Applications*. Cambridge University Press, 2017.

[154] N. G. van Kampen. *Stochastic Processes in Physics and Chemistry*. North-Holland, 3rd edition, 2007.

[155] P. O. Vontobel. The Bethe permanent of a nonnegative matrix. *IEEE Transactions on Information Theory*, 59(3):1866–1901, 2013.

[156] N. D. Vvedenskaya, R. L. Dobrushin, and F. I. Karpelevich. Queueing system with selection of the shortest of two queues: an asymptotic approach. *Problems of Information Transmission*, 32(1):20–34, 1996.

[157] M. J. Wainwright, T. S. Jaakkola, and A. S. Willsky. A new class of upper bounds on the log partition function. *IEEE Transactions on Information Theory*, 51(7):2313–2335, 2005.

[158] H. Wang and K. C. Sevcik. Experiments with improved approximate mean value analysis algorithms. *Perform. Eval.*, 39(1-4):189–206, 2000.

[159] W. Wang, G. Casale, and C. A. Sutton. A bayesian approach to parameter inference in queueing networks. *ACM Trans. Model. Comput. Simul.*, 27(1):2:1–2:26, 2016.

[160] Weikun Wang, Giuliano Casale, Ajay Kattepur, and Manoj K. Nambiar. Maximum likelihood estimation of closed queueing network demands from queue length data. In *Proc. of ACM/SPEC ICPE*, pages 3–14. ACM, 2016.

[161] W. Whitt. The queueing network analyzer. *Bell System Technical Journal*, 62(9):2779–2815, 1983.

[162] W. Whitt and W. You. A robust queueing network analyzer based on indices of dispersion. *Naval Research Logistics*, 69(1):36–50, 2022.

[163] P. Whittle. Partial balance and insensitivity. *Journal of Applied Probability*, 22(1):168–176, 1985.

[164] A. Wierman and M. Harchol-Balter. Classifying scheduling policies with respect to unfairness in an M/GI/1. In *Proc. ACM SIGMETRICS*, pages 238–249, 2003.

[165] S. F. Yashkov. A derivation of response time distribution for an m/g/1 processor-sharing queue. *Problems of Control and Information Theory*, 12:133–148, 1983.

[166] J. Zahorjan, D. L. Eager, and H. M. Sweillam. Accuracy, speed, and convergence of approximate mean value analysis. *Perform. Eval.*, 8(4):255–270, 1988.

[167] T. Zheng, J. Yang, M. Woodside, M. Litoiu, and G. Iszlai. Tracking time-varying parameters in software systems with extended Kalman filters. In *Proc. of the Conf. of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 334–345, 2005.

[168] S. Zhou and M. Woodside. A multiserver approximation for cloud scaling analysis. In *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering*, ICPE '22, page 129–136, New York, NY, USA, 2022. Association for Computing Machinery.

Appendix A

Examples

The table below lists the examples available under the `examples/` folder, grouped by the subfolder they belong to. Each entry is a Python script; the tutorial entries (`tut*`) and part of the gallery are additionally provided as Jupyter notebooks with the same name.

Table A.1: Examples

Example	Problem
`ag_closed_network`	Closed network solved through the agent-based interface
`ag_gnetwork`	G-network with negative customers and signals
`ag_jackson_network`	Jackson network solved through the agent-based interface
`ag_multiclass_closed`	Multiclass closed network solved through the agent-based interface
`ag_tandem_open`	Open tandem queue solved through the agent-based interface
`cache_replc_rr`	A small cache model with an open arrival process and random replacement
`cache_replc_fifo`	A small cache model with a closed job population and FIFO replacement
`cache_replc_lru`	A cache model with least-recently-used replacement
`cache_replc_hlru`	h-LRU (LRU(m)) replacement over h lists of capacities $m_1, \ldots, m_h$
`cache_replc_qlru`	q-LRU replacement, where a missed item is admitted with probability q
`cache_replc_climb`	CLIMB (transposition) replacement, where a hit moves an item up one position
`cache_compare_replc`	Comparison of replacement policies across the `CTMC`, `MVA` and `NC` solvers
`cache_replc_routing`	A caching model with state-dependent output routing
`cache_mmap_rr_env`	MMAP-fed cache with two correlated classes in a random environment
`cache_rmf_transient`	Transient cache analysis by refined mean-field approximation
`retrieval_simple`	Cache with a single-queue retrieval system (delayed hits)
`retrieval_ps`	Retrieval system served by a single processor-sharing station
`retrieval_chain`	Cache with a chained retrieval system (delayed hits)
`retrieval_routing`	Cache with probabilistic per-item routing through a retrieval system
`retrieval_default`	Retrieval system whose per-item miss routing and service are taken by default
`cdf_respt_closed`	Station response time distribution in a single-class single-job closed network
`cdf_respt_closed_threeclasses`	Station response time distribution in a multi-chain closed network
`cdf_respt_open_twoclasses`	Station response time distribution in a multi-chain open network
`cdf_respt_distrib`	Simulation-based station response time distribution analysis
`cdf_respt_populations`	Station response time distribution under increasing job populations
`cl_basic`	Open cluster with four processor-sharing servers and random dispatching
`cl_closed`	Closed cluster with a think stage and three processor-sharing servers
`cl_multiclass`	Two-class open cluster, e.g. interactive against batch traffic
`cl_compare`	Comparison of random and round-robin dispatching on the same cluster

Continued on next page

Table A.1 – Examples. *Continued from previous page*

Example	Problem
cl_sweep	Arrival-rate sweep on a two-server cluster
dispatch_open	Open cluster built through the Cluster builder
dispatch_closed	Closed cluster built through the Cluster builder
cqn_repairmen	Solving a single-class exponential closed queueing network
cqn_repairmen_multi	Multi-server closed queueing network with repairmen
cqn_twoclass_hyperl	Solving a closed queueing network with a multi-class FCFS station
cqn_threeclass_hyperl	Solving exactly a multi-chain product-form closed queueing network
cqn_twoclass_erl	Closed network with round robin scheduling
cqn_multiserver	Local state space generation for a station in a closed network
cqn_multiserver_nc	Closed delay and multi-server FCFS queue solved by normalizing constants
cqn_twoqueues_multi	Closed queueing network with two multi-server queues
cqn_oneline	1-line exact MVA solution of a cyclic network of PS and INF stations
cqn_bcmp_theorem	Comparison of different scheduling policies that preserve the product-form solution
cqn_multichain_cs	Closed network with three chains, five classes and class switching
cqn_lcfs_multiclass	Multiclass closed network with LCFS and LCFS-PR scheduling
cqn_bas_blocking	Closed network with blocking-after-service (BAS) finite-buffer blocking
cqn_scheduling_dps	Parameterization of a discriminatory processor sharing (DPS) station
cqn_mmpp2_service	Automatic detection of solvers that cannot analyze the model
cs_implicit	Class switching with implicit routing
cs_single_diamond	Class switching controlled by a reducible Markov chain, single diamond pattern
cs_multi_diamond	Class switching controlled by a reducible Markov chain, multiple diamond patterns
cs_transient_class	Class switching with transient classes
ctmc_tandem_mm1	Tandem of two M/M/1 stations solved by the CTMC solver
ctmc_spn_mm1	M/M/1 queue modelled as a stochastic Petri net and solved by CTMC
ctmc_spn_vs_nc_closed_qn	Closed network as a stochastic Petri net: CTMC against NC comparison
est_ubr_open	Utilization-based regression (UBR) demand estimation on an open network
est_ubr_closed	UBR demand estimation on a closed network
est_ubr_changepoint	UBR sliding-window change-point detection on an open network
est_ubo_open	Utilization-based optimization (UBO) demand estimation on an open network
est_ubo_closed	UBO demand estimation on a closed network
est_ubo_changepoint	UBO sliding-window change-point detection on an open network
est_ubo_variants_closed	Comparison of UBO and maximum-likelihood estimators on a closed network
est_erps_open	Extended regression for processor sharing (ERPS) estimation, open network
est_erps_closed	ERPS demand estimation on a closed network
est_mle_open	Maximum-likelihood demand estimation on an open network
est_qmle_closed	Queue-length maximum-likelihood estimation on a closed network
est_mcmc_closed	Markov chain Monte Carlo estimation on a closed network
est_ekf_open	Extended Kalman filter estimation on an open network
est_ekf_closed	Extended Kalman filter estimation with automatic method selection
est_ekf_changepoint	Extended Kalman filter sliding-window change-point detection
est_trace_closed	Comparison of trace-based estimators (MLPS, FMLPS, Gibbs) on a PS station
fcr_lossn	Loss network analysis using finite capacity regions
fcr_constraints	Constraint types available for finite capacity regions
fcr_lincon	Linear admission constraints for finite capacity regions
fcr_mm1waitq	Finite capacity region with blocking compared against an M/M/1 queue
fcr_mm1kdrop	Finite capacity region with dropping compared against an M/M/1/K queue
fcr_oqnwaitq	Finite capacity region with blocking in a multiclass open network
fcr_oqndrop	Finite capacity region with dropping in a multiclass open network
fes_single_class	Single-class flow-equivalent server (FES) aggregation
fes_aggregation	Flow-equivalent server aggregation of a subnetwork
ld_fes_singleclass	Single-class FES aggregation with verification of Norton's theorem
ld_fes_multiclass	Multiclass FES aggregation with verification of Norton's theorem
ld_multiserver_fcfs	Solving a single-class load-dependent closed model
ld_multiserver_ps_twoclasses	Solving a two-node multiclass load-dependent closed model

Continued on next page

Table A.1 – Examples. *Continued from previous page*

Example	Problem
ld_multiserver_ps	Solving a three-node multiclass load-dependent closed model
ld_class_dependence	Load-dependent model with class dependence
fj_basic_open	A simple single class open fork-join network
fj_basic_closed	A simple single class closed fork-join network
fj_tiny_closed	Minimal closed fork-join model solved exactly by the native CTMC solver
fj_twoclasses_forked	A multiclass open fork-join network
fj_basic_nesting	A closed model with nested forks and joins
fj_deep_nesting	Fork-join network with deep nesting
fj_nojoin	An open model with a fork but without a join
fj_serialfjs_open	Two open fork-join subsystems in tandem
fj_serialfjs_closed	Closed serial fork-join network
fj_complex_serial	Complex serial fork-join network
fj_cs_prefork	Fork-join network with class switching before the fork
fj_cs_postfork	Two-class fork-join with a class that switches into the other after the fork
fj_cs_multi_visits	Two fork-join loops within the same chain
fj_route_overlap	A model with overlapping routes in a fork-join network
fj_asymm	Asymmetric fork-join network
fj_delays	Fork-join network with delays
fj_threebranches	Fork-join network with three branches
init_state_fcfs_exp	Specifying an initial state and prior in a single class model
init_state_fcfs_nonexp	Specifying an initial state and prior in a multiclass model
init_state_ps	Specifying an initial state and prior in a model with class-switching
ldes_warmstart	Warm start of the LDES simulator from an auxiliary solver
lcq_singlehost	Single-host layered cache queueing network
lcq_threehosts	Three-host layered network with a multi-level cache
lqn_basic	Basic layered network example
lqn_serial	Analyze a layered network specified in a LQNS XML file
lqn_twotasks	Layered network with two tasks
lqn_multi_solvers	Specifying and solving a basic layered network with several solvers
lqn_setup	Layered network with a setup/delay-off task
lqn_server_pools	Layered network whose processor servers are not interchangeable, declared with a class-compatibility graph
lqn_workflows	Workflow modeling in layered networks
lqn_bpmn	BPMN to layered network transformation
lqn_moment3	Third-moment (higher-order) analysis of a layered network
lqn_transient	Transient (time-dependent) analysis of a layered network
lqn_paramident	Identification of hidden LQN parameters from measured performance
lqn_sockshop	Layered model of the Sock Shop microservice benchmark
lqn_ofbiz	Layered model of the Apache OFBiz enterprise application
largescale_tbi	Large-scale transient fluid analysis with trajectory-based iteration (TBI)
mam_transient_mapmap1	Transient analysis of a MAP/MAP/1 queue by matrix-analytic methods
mqn_basic	Solving a queueing network model with both closed and open classes
mqn_singleserver_ps	Mixed model with single-server PS stations
mqn_singleserver_fcfs	Mixed model with single-server FCFS stations
mqn_multiserver_ps	A difficult mixed model with sparse routing among multi-server PS nodes
mqn_multiserver_fcfs	Mixed model with multi-server FCFS nodes
mqn_multichain_cs	Mixed network with multiple chains and class switching
oqn_basic	Solving a queueing network model with open classes, scalar cutoff options
oqn_oneline	1-line solution of a tandem network of PS and INF stations
oqn_cs_routing	Solving a queueing network model with open classes, matrix cutoff options
oqn_multichain_cs	Open network with three chains, five classes and class switching
oqn_trace_driven	Trace-driven simulation of an M/M/1 queue
oqn_nhpp	Open network with a non-homogeneous Poisson (cyclic) arrival process
oqn_vsinks	A model illustrating the emulation of multiple sinks

Continued on next page

Table A.1 – Examples. *Continued from previous page*

Example	Problem
oqn_fourqueues	A large multiclass example with PS and FCFS
opt_server_sizing	Minimum number of servers of an M/M/c queue under a utilization bound
opt_bisection_sizing	Exact server sizing by bisection, using $O(\log n)$ solver evaluations
opt_population_sizing	Largest closed-class population sustaining a service-level target
opt_service_rate	Cheapest continuous service rate meeting a response-time target
opt_load_balancing	Routing split from a source to a fast and a slow PS station
opt_decomposition	Two-variable problem (server count and service rate) solved by decomposition
opt_robust_sizing	Worst-case server sizing across average-load and peak-load scenarios
opt_pareto_frontier	Cost against utilization-bound Pareto frontier by parametric sweep
pas_mmk	Pass-and-swap (P&S) station as a multiserver order-independent M/M/K queue
pas_compatibility_5class	Pass-and-swap queue with a five-class compatibility (swapping) graph
pas_selfloop	Pass-and-swap queue whose swapping graph includes a self-loop
pas_saturation	Pass-and-swap saturation handling for large or unbounded buffers
pas_closed_tandem_fig6	Closed tandem of two pass-and-swap queues
pas_cyclic_ctmc_vs_bruteforce	Validation of the CTMC solver on a closed cyclic P&S network
pas_cyclic_normconst_bruteforce	Brute-force normalizing constant of a cyclic P&S network
pas_nc_sampling	Importance-sampling normalizing-constant analysis of a closed P&S network
polling_exhaustive_exp	Exhaustive polling with exponential switchover times
polling_exhaustive_det	Exhaustive polling with deterministic switchover times
polling_gated	Gated polling
polling_klimited	K-limited polling
switchover_basic	Basic switchover times model
prio_hol_open	A multiclass open example with PS, SIRO, FCFS and head-of-line priority
prio_hol_closed	A high-load multiclass example with PS, SIRO, FCFS and head-of-line priority
prio_psprio	A repairmen model with PS priority scheduling
prio_identical	Priority model with identical classes
renv_basic	Introductory random-environment model
renv_twostages_repairmen	Solving a model in a 2-stage random environment with exponential rates
renv_threestages_repairmen	Solving a model in a 3-stage random environment with Erlang rates
renv_fourstages_repairmen	Solving a model in a 4-stage random environment with Coxian rates
renv_node_breakdown	Node failure and repair through the random-environment breakdown API
renv_genqn	Random environment over an automatically generated queueing network
renv_lqn_twostages	Layered network operating in a two-stage random environment
renv_container_terminal	Daily-cycle container terminal in a random environment
renv_rotterdam_blending	Semi-open queueing network with environment blending
example_mapqn2renv	Transformation of an MMPP-service queue into a random-environment model
rewardModel_mm1k	Reward-based CTMC analysis using custom reward functions (setReward, getReward, getAvgReward)
rewardModel_multiclass	Multi-class reward analysis
rewardModel_aggregation	Aggregation operations on rewards
rewardModel_templates	Predefined reward templates
sdroute_closed	A model with round-robin routing
sdroute_twoclasses_closed	A model with round-robin routing after multi-class PH and MAP service
sdroute_open	A load-balancer modeled as a router
sdroute_jsq	Join-the-shortest-queue state-dependent routing
solver_custom	Template for a user-defined solver plugged into the solver interface
statepr_aggr	Computing marginal state probabilities for a node
statepr_aggr_large	Computing marginal state probabilities for a node under class-switching
statepr_sys_aggr	Computing joint state probabilities for a system with two nodes under class-switching
statepr_sys_aggr_large	Computing joint state probabilities under class-switching and with delay nodes
statepr_allprobs_ps	Computing probabilities under PS class-switching and with delay nodes
statepr_allprobs_fcfs	Computing probabilities under PS and FCFS class-switching and with delay nodes
spn_basic_open	Simulation of a simple stochastic Petri net model
spn_basic_closed	Basic closed stochastic Petri net

Continued on next page

Table A.1 – Examples. *Continued from previous page*

Example	Problem
spn_open_sevenplaces	Simulation of a complex stochastic Petri net model
spn_closed_twoplaces	Closed stochastic Petri net with two places
spn_closed_fourplaces	Closed stochastic Petri net with four places
spn_twomodes	Stochastic Petri net with two modes
spn_fourmodes	Stochastic Petri net with four modes
spn_inhibiting	Stochastic Petri net with inhibiting arcs
spn_pareto_service	Stochastic Petri net with Pareto-distributed firing times
spn_queueing_place	Closed queueing Petri net (QPN) with two queueing places
tut01_mm1_basics	M/M/1 queue basics
tut02_mg1_multiclass_solvers	Multiclass M/G/1 queue across solvers
tut03_repairmen	Machine interference (repairmen) problem
tut04_lb_routing	Round-robin load balancing
tut05_completes_flag	Use of the completes flag on a class
tut06_cache_lru_zipf	Queueing network with an LRU cache under Zipf-distributed references
tut08_respt_cdf	Response time distribution and percentiles
tut09_opt_load_balancing	Optimal load balancing
tut10_dep_process_analysis	Departure process analysis
tut11_lqn_basics	Layered queueing network basics
tut12_random_env	Random environments and the ENV solver
tut13_posterior_analysis	Posterior analysis with uncertain parameters
tut14_cluster	Open cluster model
tut15_bound_analysis	Bound analysis with the BA solver
wf_serial	Serial workflow model
wf_parallel	Parallel workflow model (AND-fork/join)
wf_branch	Probabilistic branching workflow model (OR-fork/join)
wf_loop	Workflow model with a loop
wf_erlang	Workflow with Erlang-distributed activity durations
wf_aph	Workflow with acyclic phase-type activity durations
wf_complex	Complex workflow combining sequence, branching and parallelism

Appendix B

Supported language features

B.1 Solver features

Once a model is specified, it is possible to use the `get_used_lang_features` function to obtain a list of the features of a model. For example, the following conditional statement checks if the model contains a FCFS node

```
if model.get_used_lang_features().list.SchedStrategy_FCFS:
    # ...
```

Every LINE solver implements the `support` to check if it supports all language features used in a certain model

```
print(JMT.supports(model))
```

B.1.1 Which solvers can solve this model

The question a model raises first is not which features it uses but which solvers and solver methods can consume them, and `find_solver` answers it directly from the model. `find_method` and `help` are aliases: the same question is asked both as "which solver do I use" and as "which method do I pass", and the answer serves both.

The result carries one row per pair, with columns `Solver` (the method family), `Method` (the method name to pass as a solver method), `Runnable`, `Class`, `Metrics` and `Reason`. Because `Method` is a method name the solver constructor accepts, a row can be acted on without translation:

`Class` states what kind of answer the method returns: `exact`, `approx`, `bound` or `simulation`. Exactness is claimed of the *model* and not of the algorithm in the abstract, since the exactness of a normalizing constant or of mean value analysis is a property of the product-form model it is computed on: `mva.exact` reports `exact` on a product-form network and `approx` on one without a product-form solution, and the

matrix-analytic methods report `exact` only on the single-queue shape the QBD is stated for. `Metrics` lists the measure groups the family can report – `avg`, `tran`, `cdf`, `prob`, `tranprob`, `sample`, `cache`, `loss`, `orbit`, `moment` and `sens` – and is what the optional first argument filters on; a group may equally be named by any accessor that returns it. `Reason` is empty on a runnable row and otherwise names the offending features, so a refusal says what to change rather than that something is unsupported.

The command line offers the same report, `line-cli -f model.json --find-solver`, optionally narrowed by a measure and optionally including the refused pairs (`--find-solver-all`).

The gate `findSolver` applies is the one solver selection already applies before delegating, asked of every candidate rather than of the first feasible one, and `list_valid_methods` is the `Method` column of its runnable rows. The two therefore cannot disagree.

A supported feature set is a necessary but not always sufficient condition for a method to run. The `BA` solver is the clearest example: its feature set is deliberately narrow, admitting closed product-form-parameterized models only, because bounds are computed from the service demands and populations alone. Several of its methods then impose further structural restrictions that the feature set cannot express, such as the absence of infinite-server stations for `sb` and `sib`, or a single-class single-server model for `lr`. Those methods raise an error on an inadmissible model rather than returning a value that would not be a valid bound. Consequently `supports` may accept a model for which a particular `BA` method is still unavailable, and `list_valid_methods` reports the method names rather than their per-model admissibility.

B.2 State-dependent routing and service

A model is state-dependent when its routing decisions or its service rates are functions of the current state. On the routing side the strategies are those of Section B.6: `RROBIN` cycles through the destinations in a fixed order, `WRROBIN` does so through a list in which each destination appears as many times as its integer weight, and `JSQ` sends each job to the destination holding the fewest jobs. On the service side, `set_load_dependence` scales the station rate with its occupancy, which is how congestion effects, speed scaling and batch service are represented.

Support differs by solver. `CTMC` handles both families exactly, by carrying the dispatcher state and the occupancy-dependent rates in the state descriptor; the simulators `JMT`, `LDES` and `SSA` evaluate them on the sample path; `MVA` and `NC` handle load-dependent rates through their queue-dependent algorithms but relax state-dependent routing to its probabilistic counterpart. The example below dispatches jobs round-robin from a `Router` to three parallel queues.

```
router = Router(model, 'Dispatcher')
queue1 = Queue(model, 'Queue1', SchedStrategy.FCFS)
queue2 = Queue(model, 'Queue2', SchedStrategy.FCFS)
queue3 = Queue(model, 'Queue3', SchedStrategy.FCFS)
router.set_routing(jobclass, RoutingStrategy.RROBIN,
    [queue1, queue2, queue3])
```

Table B.1: Solver support for average performance metrics

		Network Solver							
Function	**Regime**	CTMC	LDES	FLD	JMT	MAM	MVA	NC	SSA
avg	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avg_table	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avg_chain	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avg_chain_table	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avg_node	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avg_node_table	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avg_node_chain	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avg_node_chain_table	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avg_sys	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avg_sys_table	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avg_arv_r	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avg_arv_r_chain	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avg_node_arv_r_chain	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avg_qlen	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avg_q_len_chain	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avg_node_q_len_chain	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avg_respt	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avg_respt_chain	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avg_node_respt_chain	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avg_sys_respt	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avg_tput	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avg_tput_chain	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avg_node_tput_chain	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avg_sys_tput	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avg_util	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avg_util_chain	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avg_node_util_chain	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
get_tran_avg	Transient	✓	✓	✓	✓	✓			

Examples demonstrating state-dependent routing can be found in the examples directory, including models that use RROBIN for load distribution and JSQ for dynamic load balancing.

B.3 Class functions

The table below lists the steady-state and transient analysis functions implemented by the solvers. Since the features of the LINE solver are the union of the features of the other solvers, in what follows it will be omitted from the description.

The functions listed above with the _table suffix (e.g., avg_table) provide results in tabular format corresponding to the corresponding core function (e.g., avg). The features of the core functions are as follows:

- avg: returns the mean queue-length, utilization, mean response time (for one visit), and throughput for

Table B.2: Solver support for advanced metrics

		Network Solver							
Function	**Regime**	CTMC	LDES	FLD	JMT	MAM	MVA	NC	SSA
get_cdf_resp_t	Steady-state	✓	✓	✓	✓	✓			
getAvgAoI	Steady-state			✓					
getCdfAoI	Steady-state			✓					
get_prob	Steady-state	✓	✓						
get_prob_aggr	Steady-state	✓	✓	✓	✓		✓	✓	
get_prob_marg	Steady-state					✓	✓	✓	
get_prob_sys	Steady-state	✓	✓						✓
get_prob_sys_aggr	Steady-state	✓	✓		✓			✓	
get_prob_norm_const_aggr	Steady-state				✓		✓	✓	
get_cdf_pass_t	Steady-state			✓		✓	✓		
get_avg_reward	Steady-state	✓	✓						
get_tran_cdf_pass_t	Transient		✓	✓					
get_tran_cdf_resp_t	Transient		✓		✓				
get_tran_prob	Transient	✓	✓						
get_tran_prob_aggr	Transient	✓	✓						
get_tran_prob_sys	Transient	✓	✓						
get_tran_prob_sys_aggr	Transient	✓	✓						
get_tran_reward	Transient	✓	✓						
sample	Transient		✓						✓
sample_aggr	Transient		✓		✓				✓
sample_sys	Transient		✓						✓
sample_sys_aggr	Transient		✓		✓				✓

each station and class.

- `avg_chain`: returns the mean queue-length, utilization, mean response time (for one visit), and throughput for every station and chain.
- `avg_sys`: returns the system response time and system throughput, as seen as the reference node, by chain.
- `get_cdf_resp_t`: returns the distribution of response times (for one visit) for the stations at steady-state.
- `get_cdf_pass_t`: returns the distribution of *passage* (first-visit) times at steady-state for the stations of interest, complementing the transient counterpart `get_tran_cdf_pass_t`.
- `get_prob_marg`: returns the marginal queue-length distribution $P(n_{i,r} = k)$ for a single class r at a station i, as opposed to the joint per-class distribution returned by `get_prob_aggr`.
- `getAvgAoI`: returns the mean and variance of Age of Information (AoI) and Peak AoI for single-queue open systems with capacity constraints.
- `getCdfAoI`: returns the cumulative distribution function of AoI and Peak AoI.
- `avg_node`: behaves similarly to `avg`, but returns performance metrics for each node and class. For example, throughputs at the sinks can be obtained with this method.

- `get_prob`: returns state probabilities at equilibrium at a given station.
- `get_prob_aggr`: returns marginal state probabilities for jobs of different classes at a given station.
- `get_prob_sys`: returns joint probabilities for a given system state.
- `get_prob_sys_aggr`: returns joint probabilities for jobs of different classes at all stations.
- `get_prob_norm_const_aggr`: returns the normalizing constant of the state probabilities for the model.
- `get_tran_avg`: returns transient mean queue length, utilization and throughput for every station and chain from a given initial state.
- `get_tran_cdf_pass_t`: returns the distribution of first passage times in transient regime.
- `get_tran_cdf_resp_t`: returns the distribution of response times in transient regime.
- `sample`: returns the transient marginal state for a station from a given initial state.
- `sample_aggr`: returns the transient marginal state for jobs of different classes at a given station from a given initial state.
- `sample_sys`: returns the transient marginal system state for a station from a given initial state.
- `sample_sys_aggr`: returns the transient marginal system state for jobs of different classes at a given station from a given initial state.
- `get_avg_reward`: returns the steady-state expected value $E[r] = \sum_s \pi(s)\, r(s)$ of each Markov reward function r defined via `set_reward`. The `CTMC` solver computes it exactly from the stationary distribution, while the `LDES` solver estimates it by time-integrating the reward over the exact joint state along the simulated sample path, so it is correct also for nonlinear rewards.
- `get_tran_reward`: returns the transient expected reward $E[r(X(t))]$ over time for each reward function. The `CTMC` solver uses the transient distribution from uniformization, while the `LDES` solver returns the reward trajectory sampled along the simulated path.

B.4 Node types

The table below shows the node types supported by the different solvers. It should be noted that the `FLD` solver is capable of handling and nodes, but due to low accuracy when run on open models this feature is disabled in the current release.

B.5 Scheduling strategies

The table below shows the supported scheduling strategies within LINE queueing stations. Each strategy belongs to a policy class:

- preemptive resume (`SchedStrategyType.PR`)
- preemptive non-resume, i.e. restart with an independent resample (`SchedStrategyType.PNR`)

Table B.3: Solver support for nodes

	Network Solver							
Strategy	CTMC	LDES	FLD	JMT	MAM	MVA	NC	SSA
Cache	✓	✓	✓	✓		✓	✓	✓
ClassSwitch	✓	✓	✓	✓	✓	✓	✓	✓
Delay	✓	✓	✓	✓	✓	✓	✓	✓
Fork	✓	✓		✓	✓	✓		✓
Join	✓	✓		✓	✓	✓		✓
Queue	✓	✓	✓	✓	✓	✓	✓	✓
Sink	✓	✓	✓	✓	✓	✓	✓	✓
Source	✓	✓	✓	✓	✓	✓	✓	✓

- non-preemptive (`SchedStrategyType.NP`)
- non-preemptive priority (`SchedStrategyType.NPPrio`)
- preemptive-resume priority (`SchedStrategyType.PRPrio`).

The `SchedStrategyType.PR` policy class includes scheduling strategies like `LCFSPR`, where a preempted job resumes service from the point it was interrupted. In contrast, the `SchedStrategyType.PNR` policy class includes strategies like `LCFSPI` (Last-Come-First-Served Preemptive-restart Independent), where a preempted job restarts from scratch with a new service time sample that is independent of the past.

The table primarily refeers to invocation of the `avg` methods. Specialized methods, such as transient or response time distribution analysis, may be available only for a subset of the scheduling strategies supported by a solver.

The table lists both classical and specialized disciplines; the definitions of the latter are recalled in Table 3.2. One entry is worth isolating because it is not a scheduling discipline at all: `SQ` is a *routing* strategy (Section B.6), dispatching each job to the shortest of d randomly sampled queues. The specialized disciplines are supported mainly by `CTMC`, `LDES`, `JMT` and `SSA`, as the feature table below records. Worked models are in the examples folder, notably the exhaustive and gated polling examples and the DPS examples with class-dependent weights.

B.6 Routing strategies

The table below summarises which routing strategies are honoured by each network solver. Probabilistic strategies (`RAND`, `PROB`) reduce to the entries of the routing matrix, so every solver supports them. State-dependent strategies (`RROBIN`, `WRROBIN`, `JSQ`, `SQ`) require either an explicit dispatcher in the simulator (`LDES`, `JMT`) or a per-event marginal-probability callback (`SSA`). Analytical solvers (`MVA`, `NC`, `MAM`, `FLD`) approximate non-product-form policies by their probabilistic relaxation and are not appropriate for dispatching-sensitive analyses; `CTMC` handles `RROBIN`, `WRROBIN`, `JSQ` and `SQ` exactly, by augmenting the state descriptor with the dispatcher state, at the cost of an enlarged state space.

Table B.4: Solver support for scheduling strategies

		Network Solver							
Strategy	**Class**	CTMC	LDES	FLD	JMT	MAM	MVA	NC	SSA
FCFS	NP	✓	✓	✓	✓	✓	✓	✓	✓
INF	NP	✓	✓	✓	✓	✓	✓	✓	✓
SIRO	NP	✓	✓		✓		✓	✓	✓
SEPT	NP	✓	✓		✓				✓
SRPT	NP		✓		✓				
FSP	NP		✓						
SJF	NP		✓		✓				
FCFSPRIO	NPPrio	✓	✓		✓		✓		✓
EDD	NPPrio		✓		✓				
EDF	PRPrio		✓		✓				
PS	PR	✓	✓	✓	✓	✓	✓	✓	✓
DPS	PR	✓	✓	✓	✓		✓		✓
GPS	PR	✓	✓		✓				✓
PSPRIO	PRPrio	✓	✓		✓				✓
DPSPRIO	PRPrio	✓	✓		✓				✓
GPSPRIO	PRPrio	✓	✓		✓				✓
HOL	NPPrio	✓	✓	✓	✓	✓	✓		✓
LCFS	NP	✓	✓	✓	✓		✓	✓	✓
LCFSPR	PR	✓	✓	✓	✓		✓	✓	✓
LPS	PR	✓	✓		✓				✓
POLLING	NP	✓	✓		✓		✓		✓
PAS	NP	✓	✓				✓	✓	✓
OI	NP	✓	✓				✓	✓	✓

B.7 Statistical distributions

The table below summarizes the current level of support for arrival and service distributions within each solver. `Replayer` represents an empirical trace read from a file, which will be either replayed as-is by the JMT solver, or fitted automatically to a `Cox` by the other solvers. Note that JMT requires that the last row of the trace must be a number, *not* an empty row.

Table B.5: Solver support for routing strategies

	Network Solver							
Strategy	CTMC	LDES	FLD	JMT	MAM	MVA	NC	SSA
RAND	✓	✓	✓	✓	✓	✓	✓	✓
PROB	✓	✓	✓	✓	✓	✓	✓	✓
RROBIN	✓	✓		✓				✓
WRROBIN	✓	✓		✓				✓
JSQ	✓	✓		✓				✓
SQ	✓	✓		✓				✓
DISABLED	✓	✓	✓	✓	✓	✓	✓	✓

Table B.6: Solver support for statistical distributions

	Network Solver							
Distribution	CTMC	LDES	FLD	JMT	MAM	MVA	NC	SSA
APH	✓	✓		✓	✓	✓	✓	✓
BMAP					✓			
Coxian	✓	✓	✓	✓	✓	✓	✓	✓
Cox2		✓	✓	✓				
Exp	✓	✓	✓	✓	✓	✓	✓	✓
Erlang	✓	✓	✓	✓	✓	✓	✓	✓
HyperExp	✓	✓	✓	✓	✓	✓	✓	✓
MAP	✓	✓		✓	✓			✓
ME		✓		✓	✓			
MMPP2	✓	✓		✓	✓			✓
PH	✓	✓		✓				✓
RAP		✓		✓	✓			
Disabled	✓	✓	✓	✓	✓	✓	✓	✓
Det		✓		✓				
Gamma		✓		✓				
Lognormal		✓		✓				
Pareto		✓		✓				
Replayer		✓		✓				
Uniform		✓		✓				
Weibull		✓		✓				

Appendix C

Solver options

Table C.1 summarizes the main options available within the LINE solvers and their default values. Solver options are encoded in LINE in a structure array that is internally passed to the solution algorithms.

This can be specified as an argument to the constructor of the solver. For example, the following two constructor invocations are identical

```
s = JMT(model)
```

Modifiers to the default options can either be specified directly in the `options` data structure, or alternatively be specified as argument pairs to the constructor, i.e., the following two invocations are equivalent

```
s = JMT(model, samples=1000000)
```

Available solver options are as follows:

- `cache` (`logical`) controls reuse of a previous solution. If enabled, subsequent calls return the cached result until a `refresh` method is used. The option is enabled by default.
- `config` (`struct`) this is data structure to pass solver-specific configuration options to customize the execution of particular methods.
- `cutoff` (`integer` ≥ 1) requires to ignore states where stations have more than the specified number of jobs. A finite cutoff is required to analyze open classes using the CTMC solver; it defaults to 10 and is auto-adjusted for open and mixed models.
- `force` (`logical`) requires the solver to proceed with analyzing the model. This bypasses checks and therefore can result in the solver either failing or requiring an excessive amount of resources from the system.
- `iter_max` (`integer` ≥ 1) controls the maximum number of iterations that a solver can use, where applicable. If `iter_max=` n, this option forces the `FLD` solver to compute the ODEs over the timespan $t \in [0, 10n/\mu^{\min}]$, where $\mu^{\min}$ is the slowest service rate in the model. For the `MVA` solver this option instead regulates the number of successive substitutions allowed in the fixed-point iteration.

- `iter_tol` (`double`) controls the numerical tolerance used for convergence of iterative methods. In the `FLD` solver this option regulates both the absolute and relative tolerance of the ODE solver.
- `level` (`integer` ≥ 1, `BA` only) sets the depth of the bound hierarchies `pbh`, `cbh`, `pbk`, `bjbk` and `sib`, and defaults to 2. Raising it spends more computation and tightens the bracket monotonically toward the exact solution, which is reached at a finite depth: at `level` equal to the closed population for `pbh`, and at `level` equal to the number of stations for `cbh`. The option has no effect on the noniterative families or on the reduction bounds. See Section 8.2.2.
- `init_sol` (`solver dependent`) re-initializes iterative solvers with the given configuration of the solution variables. In the case of `MVA`, this is a matrix where element (i, j) is the mean queue-length at station i in class j. In the case of `FLD`, this is a model-dependent vector with the values of all the variables used within the ODE system that underpins the fluid approximation. In the case of `LDES`, this is a matrix of initial queue lengths used for transient analysis, where jobs are distributed according to this matrix at simulation start instead of placing all closed class jobs at reference stations.
- `confint` (`logical` or `real` $\in (0, 1)$) enables confidence interval reporting for simulation-based solvers (`JMT`, `SSA`, `LDES`). When set to `true`, a 95% confidence interval is used. When set to a numeric value between 0 and 1 (e.g., `0.99` for 99% confidence), that confidence level is used. When enabled, the `getAvgTable` output displays metrics in the format `mean ± half-width`. Default is `false` (disabled).
- `cimethod` (`string`, `LDES` only) specifies the confidence interval computation method. Valid values are `'obm'` (overlapping batch means, default), `'bm'` (non-overlapping batch means), `'spectral'` (Heidelberger–Welch spectral analysis), or `'none'` (disable CI computation). The `'spectral'` method estimates the spectral density at frequency zero via quadratic log-periodogram regression at low frequencies, accounting for autocorrelation between batches and producing wider but more statistically honest confidence intervals than batch means methods.
- `ciminbatch` (`integer` ≥ 2, `LDES` only) minimum batch size for confidence interval computation. Actual batch size is $\max(\texttt{ciminbatch}, \sqrt{n})$ where n is the sample count. Default is 10.
- `ciminobs` (`integer` ≥ 10, `LDES` only) minimum number of post-warmup observations required before computing confidence intervals. Default is 100.
- `config.variates` (`string`, `LDES` only) specifies the variance reduction method for simulation. Valid values are `'none'` (no variance reduction, default), `'antithetic'` (antithetic variates using synchronized 1-U method to generate paired samples with negative correlation), `'control'` (control variates using mean-based correction that applies post-hoc corrections based on deviation of sampled means from known theoretical means), or `'both'` (combined antithetic and control variates). Default is `'none'`.
- `events` (`integer` ≥ 1, `SSA` and `LDES`) event budget of the discrete-event simulation solvers: the number of reaction firings simulated by `SSA`, and the number of service completion events simulated by `LDES`. When set, it overrides `samples`, which remains accepted as a deprecated alias for the event budget; when unset (default), the solvers fall back to `samples` (10^4 events for `SSA`, $2 \cdot 10^5$ for `LDES`).
- `keep` (`logical`) determines if the model-to-model transformations store on file their intermediate

outputs. In particular, if `verbose`≥ 1 then the location of the `.jsimg` models sent to JMT will be printed on screen.

- `lang` (`string`) determines the codebase used to run the chosen method, either `'python'` (the default, the native no-JVM implementation) or `'java'`, which serializes the model to JSON and delegates the analysis to `jline.jar`. The default can also be set through the `LINE_SOLVER_LANG` environment variable. Under `'java'` the JAR is located through the `LINE_JLINE_JAR` environment variable or in a `common/` directory on the path from the package upwards; if it is found nowhere, it is downloaded automatically from the project distribution site and cached. When the download also fails, the resulting error names the four available remedies: setting `LINE_JLINE_JAR`, building the JAR from `jar/`, downloading it by hand, or reverting to native solving.
- `method` (`string`) configures the internal algorithm used to solve the model.
- `samples` (`integer` ≥ 1) statistical sample budget of the simulation- and sampling-based solvers; its unit is solver-dependent. For `JMT` it is the number of samples collected *for each* performance index, with a required minimum of $5 \cdot 10^3$. For `NC`, it is the number of Monte Carlo draws used by the sampling-based normalizing-constant methods. For `SSA` and `LDES`, it is a deprecated alias of `events` (see above) and is used only when `events` is not set.
- `seed` (`integer` ≥ 1) controls the seed used by the pseudo-random number generators. For example, simulation-based solvers will give identical results across invocations only if called with the same `seed`.
- `stiff` (`logical`) requires the solver to use a stiff ODE solver.
- `timeout` (`double`, seconds, default `Inf`) sets a cooperative wall-clock budget measured from solver launch. The budget is polled at checkpoints inside the long-running phases of the analysis, in particular during state-space generation and inside iterative analyzers. When the budget is exhausted the solver raises an error rather than returning a partially converged or partially enumerated result, so that a timed-out run cannot be mistaken for a completed one. The default `Inf` disables the check.
- `timespan` (`real interval`) requires the transient solver to produce a solution in the specified temporal range. If the value is set to `[float('inf'), float('inf')]` the solver will only return a steady-state solution. For the `FLD` solver and in simulation, this setting has the same computational cost of `[0, float('inf')]`, therefore the latter is used as default for this solver.
- `timestep` (`double`) controls the fixed time interval for transient analysis in the `CTMC` solver. When specified, the solver generates equally-spaced time points instead of using adaptive time stepping. If not specified or set to empty, the solver uses adaptive ODE time stepping. This option only affects transient analysis and is ignored for steady-state computations.
- `tol` default numerical tolerance for all uses other than the ones where `iter_tol` is used.
- `tranfilter` (`string`, `LDES` only) specifies the transient (warmup) detection method. Valid values are `'mser5'` (MSER-5 automatic truncation, default), `'fixed'` (fixed fraction warmup removal), or `'none'` (no transient filtering). The `SSA` solver does not expose adaptive transient detection but supports a configurable warmup discard via `warmupfrac` (see below).

- `mserbatch` (`integer` ≥ 1, `LDES` only) batch size for the MSER transient detection algorithm. Only used when `tranfilter='mser5'`. Default is 5 (standard MSER-5).
- `warmupfrac` (`real` $\in [0, 1)$, `SSA` and `LDES`) fraction of collected samples discarded as warmup before computing steady-state averages. For `LDES` this is used only when `tranfilter='fixed'` (default 0.2); for `SSA` it controls the initial-sample discard directly (default 0.0, i.e. disabled). Values of 0.1–0.3 are typical on open networks.
- `obmoverlap` (`real` $\in [0, 1]$, `LDES` only) overlap fraction for overlapping batch means confidence intervals. A value of 0.5 means 50% overlap (standard OBM). Only used when `cimethod='obm'`. Default is 0.5.
- `spectralLowFreqFrac` (`real` $\in (0, 1]$, `LDES` only) fraction of the lowest periodogram frequencies used for the log-periodogram regression in the Heidelberger–Welch spectral method. A smaller value concentrates the regression near frequency zero, giving a more local estimate of the spectral density; a larger value uses more frequencies for a more stable but potentially biased estimate. Only used when `cimethod='spectral'`. Default is 0.25.
- `cnvgon` (`logical`, `LDES` only) enables convergence-based stopping. When enabled, simulation stops when the confidence interval half-width relative to the mean falls below the convergence tolerance for all metrics. Default is `false`.
- `cnvgtol` (`real` $\in (0, 1)$, `LDES` only) convergence tolerance threshold. Simulation stops when $(\text{CI half-width}/\text{mean}) <$ `cnvgtol` for all metrics. A value of 0.05 means 5% relative precision. Default is 0.05.
- `cnvgbatch` (`integer` ≥ 1, `LDES` only) minimum number of batches required before checking for convergence. More batches provide more reliable confidence interval estimates. Default is 20.
- `cnvgchk` (`integer` ≥ 0, `LDES` only) number of events between convergence checks. A value of 0 means auto-calculate as `samples`$/50$. Default is 0 (auto).
- `verbose` controls the verbosity level of the solver. Supported levels are 0 for silent, 1 for standard verbosity, 2 for debugging.

Table C.1: Default values of the LINE solver options and their default assignments (Python)

	Solver default							
Option	MVA	CTMC	LDES	FLD	JMT	MAM	NC	SSA
cache	True	True	True	True	True	True	True	True
confint			False		False			False
config								
cutoff		10						
force	False	False	False	False	False	False	False	False
keep			True		True			
init_sol	None			None				
hide_immediate				False				
iter_max	10^3			200				
iter_tol	10^{-6}			10^{-4}		10^{-4}		
method	'default'	'default'	'default'	'default'	'default'	'default'	'default'	'default'
pstar				20.0				
samples			$2 \cdot 10^5$		10^4			10^4
events			unset					unset
seed	23000	23000	23000	23000	23000	23000	23000	23000
softmin_alpha				20.0				
stiff				True				
timeout	inf	inf	inf	inf	inf	inf	inf	inf
timespan		[inf,inf]	[inf,inf]	[0.0,inf]	[inf,inf]		[inf,inf]	[0.0,inf]
tol		10^{-4}		10^{-4}				
verbose	1	1	1	1	1	1	1	1

Appendix D

Interoperability

This chapter collects the facilities that LINE offers for moving models and results in and out of the tool. We first describe the scripts that import external models and export LINE models to reusable scripts, then the portable JSON serialization format used for model interchange across the MATLAB, Java, and Python codebases, and finally the command-line interface (`line-cli`) that drives the solvers from the terminal.

D.1 Model import and export

LINE offers a number of scripts to import external models into `Network` object instances that can be analyzed through its solvers. The available scripts are as follows:

- `JMT2LINE` imports a JMT simulation model (`.jsimg` or `.jsimw` file) instance.

This script requires in input the filename and desired model name, and returns a single output, e.g.,

```
sn = JMT2LINE.load('examples/data/JMT/jmt_example.jsimg', 'Mod1')
```

where `sn` is an instance of the class.

JSON model serialization. LINE models can be saved to and loaded from a portable JSON format conforming to the `line-model.schema.json` specification. This enables model exchange across codebases and integration with external tools. Supported model types include `Network`, `LayeredNetwork`, `Workflow`, and `Environment`.

```
from line_solver.io import save_model, load_model
save_model(model, 'mymodel.json')           # save
model = load_model('mymodel.json')          # load
```

D.1.1 Supported JMT features

Table D.1.1 lists the JSIMgraph/JSIMwiz model features supported by the `JMT2LINE` transformation. We indicate as "Fully" supported a feature that is supported in the import and such that the resulting model can be solved in LINE using at least `JMT`. A feature with "Partial" support implies that some core aspects of this feature available in JSIM are not available in LINE.

A few notes are needed to clarify the entries with partial support:

- `Fork` and `Join` are supported with their default policies. Advanced policies, such as partial joins or setting a distribution for the forked tasks on each output link, are not supported yet.
- a single `Sink` and a single `Source` can be instantiated in a LINE model, whereas there is no such constraint in JMT.
- `Finite Capacity Region` is supported on export (LINE to JMT), but the `JMT2LINE` import does not yet parse JSIM blocking regions.

Table D.1: Supported JSIM features for automated model import and analysis

JMT Feature	Support	Notes
Distributions	Full	Phase-Type, Burst (MAP), Burst (MMPP2), Deterministic, Disabled, Exponential, Erlang, Gamma, Hyperexponential, Coxian, Lognormal, Pareto, Uniform, Zero Service Time, Replayer, Trace, Weibull
Classes	Full	Open class, Closed class, Class priorities
Metrics	Full	Number of customers, Residence Time, Throughput, Response Time, Throughput per sink, Utilization, Arrival Rate
Nodes	Full	ClassSwitch, Place, Delay, Queue, Router, Transition
Routing	Full	Random, Probabilities, Round Robin, Weighted Round Robin, Power of k, Join the Shortest Queue
Mechanisms	Full	Polling, Setup times, Delay-off times, Switchover times, Impatience, Load Dependence, Retrial, Soft deadlines, Heterogeneous servers, Server Compatibilities
Scheduling	Full	FCFS, FCFSPRIO, LCFS, LCFS-PR, SIRO (Random), SJF, SEPT, LJF, LEPT, PS, DPS, GPS, PS Priority, DPS Priority, GPS Priority, LPS, EDD, EDF, SRPT, FSP
Nodes	Partial	Fork, Join, Source, Sink, Finite Capacity Region
Distributions	No	Burst (General), Normal
Nodes	No	Scaler, Semaphore
Routing	No	Shortest Response Time, Least Utilization, Fastest Service, Load Dependent, Class Switch Routing
Metrics	No	Drop rate, Response time per sink, Power
Scheduling	No	TBS, QBPS
Mechanisms	No	Parallelism

D.2 JSON model format

LINE defines a portable JSON format for saving and loading queueing network models. The format is supported across the MATLAB, Java and Python-native codebases and enables model interchange without

loss of information. A formal JSON Schema specification is available at `doc/line-model.schema.json` in the LINE distribution.

The reference for this format is the implementation, that is the four bridges

- `matlab/src/io/linemodel_save.m`
- `matlab/src/io/linemodel_load.m`
- `python/line_solver/io/linemodel_io.py`
- `jar/src/main/java/jline/io/LineModelIO.java`

Where a reader accepts more than one spelling of a key, the spelling emitted by the writers is described below as *canonical* and the remaining ones as *legacy aliases*.

D.2.1 API

Models are saved and loaded using each codebase's JSON I/O functions:

```
from line_solver import save_model, load_model
save_model(model, 'mymodel.json')
model = load_model('mymodel.json')
```

D.2.2 Top-level structure

Every LINE JSON file has the following envelope:

```
{
  "format": "line-model",
  "version": "1.0",
  "model": { ... }
}
```

The `model` object must contain a `type` field set to one of `Network`, `LayeredNetwork`, `Workflow`, or `Environment`. Any other value is rejected by the loaders.

Models held in a foreign format (`.lqnx`, `.jmva`, `.jsimg`) are not referenced from within a LINE JSON file; they are passed directly to the tool that reads them, which dispatches on the file extension. See Section D.3 for the `line-cli` behaviour.

D.2.3 Network models

A `Network` model describes an extended queueing network. Its fields are:

Table D.2: Network model fields

Field	Type	Description
`type`	`"Network"`	Discriminator of the model kind. Required.
`name`	`string`	Model name. Node and class names must be unique within it. Required.

Field	Type	Description
nodes	array of Node	Node definitions, in the order that fixes the node indices used by the routing matrix. Required.
classes	array of JobClass	Job class definitions, in the order that fixes the class indices. Required.
routing	RoutingMatrix	Routing probability matrix, given per class pair over the nodes. Required.
routingStrategies	{key: {key: RoutingStrategy}}	Per-node, per-class routing strategy: {nodeName: {className: strategy}}. Written at model level; the readers also accept it nested inside the 'routing' object and prefer the nested copy. Only strategies other than PROB and RAND are emitted, since the routing matrix already captures those.
routingWeights	{key: {key: {key: number}}}	Per-node, per-class, per-destination routing weights for WRROBIN: {nodeName: {className: {destNode: weight}}}.
routingParams	{key: {key: SQParams}}	Per-node, per-class parameters of the parameterized routing strategies: {nodeName: {className: params}}. Absent params make a SQ node fall back to d=2, which warns.
finiteCapacityRegions	array of FiniteCapacityRegion	Finite capacity regions constraining groups of stations jointly.
globalDependence	GlobalDependence	Global (Whittle) rate scaling phi(n) over the whole network state, declared through setGlobalDependence. Model level rather than per node, since its argument is not station-local.
rewards	array of Reward	Reward definitions evaluated on the state space by the CTMC solver.

The routingStrategies, routingWeights and routingParams objects are written at model level. The readers also accept them nested inside the routing object, and prefer the nested copy when both are present.

Nodes

Each node is an object with at least name and type fields. Table D.3 lists the node types emitted by the writers. Additional fields depend on the node type.

Table D.3: Node types

Type	Description
Source	Generates arrivals for open classes
Sink	Absorbs departing jobs
Queue	Queueing station with one or more servers
Delay	Infinite-server station
Fork	AND-fork for parallel branching
Join	AND-join for synchronization; references a paired Fork via forkNode
Router	Probabilistic routing node (no queueing)
Cache	Cache node with hit/miss class transformation

Type	Description
ClassSwitch	Deterministic class-switching node
Place	Place node for stochastic Petri nets
Transition	Transition node for stochastic Petri nets
Logger	Logging/monitoring node

A node of any other class is serialized under its own class name, obtained by reflection; such a node is not restored by the readers.

Table D.4 lists the fields available on station nodes (Queue, Delay, Source, Place).

Table D.4: Station node fields

Field	Type	Description
name	string	Node name, unique within the model. Routing and per-class maps key on it. Required.
type	Source, Sink, Queue, Delay, Fork, Join, ...	Node type. A node of any other class is serialized under its own class name, obtained by reflection, and is not restored by the readers. Required.
scheduling	SchedStrategy	Scheduling strategy of the station. Omitted for nodes that do not serve jobs.
servers	integer	Number of servers (default 1). There is no infinite sentinel: use a Delay node for an infinite-server station.
buffer	integer	Total buffer capacity; omit for infinite.
classCap	{key: integer}	Per-class buffer capacity: {className: capacity}.
dropRule	{key: DropRule}	Per-class drop rule: {className: rule}.
service	{key: Distribution}	Per-class service/arrival distribution: {className: Distribution}.
markedClasses	array of string	Source only: marked (MMAP) arrival binding; class names ordered by mark index, so mark k emits jobs of the k-th listed class. All listed classes share one MMAP arrival entry in 'service'.
schedParams	{key: number}	Per-class scheduling parameters (DPS weights, GPS shares, priorities): {className: value}.
serverTypes	array of ServerType	Heterogeneous server types for this node.
heteroSchedPolicy	ORDER, ALIS, ALFS, FAIRNESS, FSF, RAIS	Heterogeneous server scheduling policy.
switchoverTimes	array of object	Class-to-class switchover times: [{from, to, distribution}]. A per-class (rather than class-pair) switchover omits 'to'.
pollingType	GATED, EXHAUSTIVE, KLIMITED	Polling strategy for POLLING scheduling. Crosses as the uppercase enum name.
pollingPar	integer	K limit for KLIMITED polling.
loadDependence	LoadDependence	Load-dependent scaling factor f(n) at this station.
classDependence	ClassDependence	Class-dependent scaling beta_{i,r}(n) at this station. A sibling of loadDependence, not one of its modes.
jointDependence	JointDependence	Joint-dependent (non-product-form) scaling eta_i(n) at this station. A sibling of classDependence; the handle may read the joint per-class vector arbitrarily, so the demands are not product form.

Field	Type	Description
oiServiceRate	{key: {key: number}}	Order-independent per-state service rate table: {className: {state: rate}}.
oiCutoffs	array of integer	Per-class cutoffs of the oiServiceRate table.
swapGraph	array of array of number	Pass-and-swap graph, read only for a station scheduled PAS.
departureDiscipline	{key: N, FIFO}	Place only: per-class departure discipline {className: rule}.
balking	{key: object}	Per-class balking configuration. Keys are class names.
retrial	{key: object}	Per-class retrial configuration. Keys are class names.
patience	{key: object}	Per-class patience (reneging) configuration. Keys are class names. A node-scoped patience overrides the class-scoped one declared on JobClass.
orbitImpatience	{key: Distribution}	Per-class orbit impatience distribution: {className: Distribution}.
immediateFeedback	{key: boolean}	Per-class immediate feedback flag: {className: bool}.
setupTime	{key: Distribution}	Per-class server setup time: {className: Distribution}.
delayOffTime	{key: Distribution}	Per-class server delay-off time: {className: Distribution}.
initialState	number or array of number	Initial state vector of a stateful node: a Place's token counts, a Cache's contents, a closed pass-and-swap (PAS) queue's ordered job placement, and the state row of any other stateful node. Written for every stateful node that has one, because a reader calls the model initialized only when all of them do.
stateSpace	array of array of number	Explicit state space of a stateful node. Emitted together with statePrior.
statePrior	array of number	Prior probability over the rows of stateSpace.

Table D.5 lists the fields of the remaining node types.

Table D.5: Fields of non-station nodes

Field	Type	Description
classSwitchMatrix	{key: {key: number}}	For ClassSwitch nodes: explicit switching probability matrix. {fromClass: {toClass: probability}}.
csMatrix	array of array of number	For ClassSwitch nodes: legacy 2D array form of classSwitchMatrix, indexed by class order.
tasksPerLink	integer	For Fork nodes: number of tasks generated per outgoing link.
forkNode	string	For Join nodes: name of the paired Fork node.
joinStrategy	STD, PARTIAL, Quorum, QUORUM, Guard	For Join nodes: join strategy. The writers emit PARTIAL for a quorum join; the readers accept QUORUM as an alias of Quorum. An unrecognized value falls back to STD.
joinQuorum	integer	Quorum count for a quorum join.
itemSizes	integer or array of integer	Cache nodes: per-item storage cost (size), positive integers, one per item. Also readable as cache.itemSizes.

Field	Type	Description
costCaps	integer or array of integer	Cache nodes: cap on the total storage cost of the items resident in each list; a single value declares one cache-wide cap. Requires itemSizes. Also readable as cache.costCaps.
modes	array of TransitionMode	For Transition (SPN) nodes: firing modes.

Cache node fields are given in Section D.2.3.

Table D.6: Scheduling strategies

Strategy	Description
INF	Infinite servers (Delay)
FCFS	First Come First Served
FCFSPR	FCFS with preemptive resume
FCFSPI	FCFS with preemptive restart
LCFS	Last Come First Served
LCFSPR	LCFS with preemptive resume
LCFSPI	LCFS with preemptive restart
SIRO	Service In Random Order
SJF	Shortest Job First
LJF	Longest Job First
PS	Processor Sharing
DPS	Discriminatory Processor Sharing (uses schedParams for weights)
GPS	Generalized Processor Sharing
SEPT	Shortest Expected Processing Time
LEPT	Longest Expected Processing Time
SRPT	Shortest Remaining Processing Time
SRPTPRIO	SRPT with priorities
HOL	Head-Of-Line priority
FCFSPRIO	FCFS with priorities
FORK	Fork node pseudo-strategy
EXT	External world pseudo-strategy
REF	Reference station pseudo-strategy
POLLING	Polling discipline (uses pollingType, pollingPar)
PSPRIO	PS with priorities
DPSPRIO	DPS with priorities
GPSPRIO	GPS with priorities
LCFSPRIO	LCFS with priorities
LCFSPRPRIO	LCFS preemptive resume with priorities
LCFSPIPRIO	LCFS preemptive restart with priorities
FCFSPRPRIO	FCFS preemptive resume with priorities
FCFSPIPRIO	FCFS preemptive restart with priorities
EDD	Earliest Due Date
EDF	Earliest Deadline First
LPS	Limited Processor Sharing
PSJF	Preemptive Shortest Job First
FB	Foreground-Background
LRPT	Longest Remaining Processing Time
SETF	Shortest Elapsed Time First
PAS	Pass and swap; the swap relation travels in swapGraph

Strategy names cross as uppercase enum names.

Cache nodes

Cache configuration is written as flat node-level keys. The readers accept both these keys and a nested `cache` object using the alternative names given in the last column of Table D.7, and prefer the nested object when both are present.

Table D.7: Cache node fields

Field	Type	Description
`numItems`	`integer`	Cache nodes: total number of cacheable items. Also readable as cache.items.
`itemLevelCap`	`integer or array of integer`	Cache nodes: capacity per list level. Also readable as cache.capacity.
`replacementStrategy`	`ReplacementStrategy`	Cache nodes: replacement policy. Also readable as cache.replacement.
`admissionProb`	`number`	Cache nodes: probability of admitting a missed item.
`hitClass`	`{key: string}`	Cache nodes: hit class mapping {inputClassName: outputClassName}.
`missClass`	`{key: string}`	Cache nodes: miss class mapping {inputClassName: outputClassName}.
`popularity`	`{key: Distribution}`	Cache nodes: per-class item popularity distributions {className: Distribution}.
`accessGraph`	`array of array of number`	Cache nodes: item access graph.
`accessProb`	`array`	Cache nodes: explicit per-class item access probabilities.
`retrievalSystem`	`object`	Cache nodes: delayed-hit retrieval configuration.
`cache`	`CacheConfig`	Nested alternative to the flat cache keys above. The writers emit the flat keys; the readers accept both and prefer this nested object when present.

Table D.8: Cache replacement policies

Policy	Description
`RR`	Random replacement
`FIFO`	First-in first-out
`SFIFO`	Strict first-in first-out
`LRU`	Least recently used
`HLRU`	h-LRU
`CLIMB`	CLIMB
`QLRU`	q-LRU

Load and class dependence

The `loadDependence` object specifies scaling of the service rate in the total station population n:

```
"loadDependence": { "type": "loadDependent", "scaling": [1.0, 1.8, 2.4] }
```

where `scaling` holds $f(n)$ for $n = 1, 2, \ldots$

The `classDependence` object is a sibling key of `loadDependence`, not one of its modes. It specifies the class-dependent scaling $\beta_{i,r}(\boldsymbol{n})$ in the per-class population vector $\boldsymbol{n}$ at the station. The underlying object is a function handle, which cannot be serialized, so the writer materializes it over the per-class box lattice $0 \leq n_r \leq c_r$, where c_r is the r-th entry of `cutoffs`:

Table D.9: Class dependence fields

Field	Type	Description
type	classDependent	Shape of the class-dependent rate scaling.
cutoffs	array of integer	Per-class cutoffs, ordered by class index. A closed class is bounded by its population; an open class uses a saturation cutoff beyond which beta is constant.
scaling	{key: array of number}	Lattice table keyed by the comma-joined 0-based per-class counts ("n1,n2,...,nR"). Each value is the length-R vector [beta_1(n), ..., beta_R(n)]; a scalar-valued handle is broadcast to R equal entries by the writer.
peak	array of number	Declared peak (maximum) rate scaling per class [p_1,...,p_R], used to normalize utilization as U_r = T_r*S_r/p_r (the same T*S/c convention as multiserver stations). A scalar peak is broadcast to R entries by the writer. When absent (legacy models) the reader derives it as the lattice peak of scaling.

A reader rebuilds a handle that clamps the population to the cutoffs, so β saturates beyond the tabulated range. When `peak` is absent (legacy models) the reader derives it as the lattice peak of the scaling.

The `jointDependence` object is the non-product-form twin of `classDependence`, storing the joint-dependent scaling $\eta_i(\boldsymbol{n})$ (see §5.2.4). It is materialized over the same per-class box lattice and rebuilt with clamping; unlike `classDependence`, the handle may read foreign class marginals, so the demands are not product form.

Table D.10: Joint dependence fields

Field	Type	Description
type	jointDependent	Shape of the joint-dependent rate scaling.
cutoffs	array of integer	Per-class cutoffs, ordered by class index. A closed class is bounded by its population; an open class uses a saturation cutoff beyond which eta is constant.
scaling	{key: array of number}	Lattice table keyed by the comma-joined 0-based per-class counts ("n1,n2,...,nR"). Each value is the length-R vector [eta_1(n), ..., eta_R(n)]; a scalar-valued handle is broadcast to R equal entries by the writer.

Field	Type	Description
peak	array of number	Declared peak (maximum) rate scaling per class [p_1,...,p_R], used to normalize utilization as U_r = T_r*S_r/p_r (the same T*S/c convention as multiserver stations). A scalar peak is broadcast to R entries by the writer. When absent the reader derives it as the lattice peak of scaling.

The globalDependence object is the model-level twin of the two above. It stores the global (Whittle) scaling $\phi(\boldsymbol{n})$ declared through setGlobalDependence (field gdscaling), whose argument is the whole $(M \times R)$ network state rather than one station's slice, so it is materialized over a lattice of that state rather than over a per-station box. Only the $(\text{station}, \text{class})$ *slots* a class can occupy carry a coordinate: a source holds no jobs, and a class with zero per-class capacity at a station never appears there, so those entries are pinned to zero. The restriction is lossless, since no service event ever fires at such a slot, and it is what keeps the lattice finite. The slots array names each coordinate by station and class, so a reordering on the writing side cannot shift one silently; cutoffs bounds each coordinate, a closed class at its own population and an open one at cutoff, the third argument of the setter. Each key is the comma-joined slot counts and each value the full $(M \times R)$ scaling flattened row-major, with peak flattened the same way. A reader rebuilds a handle that clamps the population to cutoffs, exactly as the two objects above do. A lattice above 2×10^5 points is refused by name rather than truncated: a silently shortened table would leave a delegated backend solving the same model at unscaled rates outside the tabulated range, disagreeing with the native solver without any diagnostic.

Table D.11: Global dependence fields

Field	Type	Description
type	globalDependent	Shape of the global rate scaling.
stations	array of string	Station names in station-index order, fixing the row order of the flattened (M x R) values below.
classes	array of string	Class names in class-index order, fixing the column order of the flattened (M x R) values below.
slots	array of object	The varying coordinates of the lattice, each named by station and class so a reordering on the writing side cannot shift one silently. Every (station,class) pair outside this list is pinned to 0.
cutoffs	array of integer	Per-slot cutoffs, in slot order. A closed class is bounded by its population, an open class by 'cutoff' below; the reader clamps to these, so phi saturates beyond the tabulated range.
cutoff	integer	The open-class truncation the writer used, i.e. the third argument of setGlobalDependence. It plays no part in solving and is carried so a reader restores the same declaration.

Field	Type	Description
scaling	{key: array of number}	Lattice table keyed by the comma-joined 0-based slot counts ("c1,c2,...,cP"), or "0" when no slot varies. Each value is the full (M x R) scaling flattened row-major, so the reader restores the matrix without re-deriving which return form the handle used.
peak	array of number	Declared peak (maximum) rate scaling, the (M x R) matrix flattened row-major, used to normalize utilization as U = T*S/peak (the same T*S/c convention as multiserver stations).

Impatience, balking and retrials

Three node-level objects, each keyed by class name, describe customer impatience:

```
"balking":  { "Class1": { "strategy": "QUEUE_LENGTH",
                          "thresholds": [ { "minJobs": 5, "maxJobs": -1,
                                            "probability": 0.5 } ] } },
"retrial":  { "Class1": { "delay": { "type": "Exp", "params": { "lambda": 2.0 } },
                          "maxAttempts": -1 } },
"patience": { "Class1": { "distribution": { "type": "Exp",
                                            "params": { "lambda": 0.1 } },
                          "impatienceType": "reneging" } }
```

The balking `strategy` is one of `QUEUE_LENGTH`, `EXPECTED_WAIT`, or `COMBINED`; a `maxJobs` of -1 denotes infinity. The `maxAttempts` field is -1 when the number of retrials is unlimited. The `impatienceType` is one of `reneging`, `balking`, or `retrial`.

A patience distribution may also be attached to a job class rather than to a station; see Section D.2.3. A node-scoped patience overrides the class-scoped one.

Petri net nodes

A `Place` node is a station: it carries `scheduling`, `servers`, `service`, `departureDiscipline` and `initialState`. A `Transition` node carries a `modes` array:

Table D.12: Transition mode fields

Field	Type	Description
name	string	Mode name, unique within its transition. Required.
distribution	Distribution	Firing time distribution.
timingStrategy	TIMED, IMMEDIATE	Whether the mode fires after a delay or immediately.
numServers	integer or "Infinity"	Number of firing servers; the string "Infinity" denotes an infinite-server mode.
firingPriority	integer	Integer priority breaking ties among simultaneously enabled modes; higher fires first.
firingWeight	number	Weight randomizing the choice among enabled modes of equal highest priority.

Field	Type	Description
enablingConditions	array of Arc	Token multiplicity required on each input place for the mode to be enabled.
inhibitingConditions	array of Arc	Token multiplicity on each input place above which the mode is inhibited.
firingOutcomes	array of Arc	Tokens deposited on each output place when the mode fires.

Every arc object has the same three fields: `node` (name of the `Place`), `class` (name of the job class), and `count` (number of tokens).

```
{ "name": "T1", "type": "Transition",
  "modes": [
    { "name": "mode1", "timingStrategy": "TIMED", "numServers": 1,
      "distribution": { "type": "Exp", "params": { "lambda": 1.0 } },
      "enablingConditions": [ { "node": "P1", "class": "Class1", "count": 1 } ],
      "firingOutcomes":     [ { "node": "P2", "class": "Class1", "count": 1 } ] }
  ] }
```

Job classes

Each class is an object with at least `name` and `type` fields.

Table D.13: Job class fields

Field	Type	Description
name	string	Class name, unique within the model. Per-class maps key on it. Required.
type	Open, Closed, SelfLooping, Signal	Class type. SelfLooping specializes Closed, so the writers test for it first. Required.
population	integer	For Closed and SelfLooping classes: number of jobs.
refNode	string	Reference node name (Source for Open, home station for Closed/SelfLooping/ClosedSignal).
priority	integer	Class priority (0=highest).
deadline	number	Class deadline (Inf = no deadline).
isReferenceClass	boolean	True if the class is the reference class of its chain.
patience	Distribution	Class-scoped patience distribution. A node-scoped patience overrides it.
impatienceType	ImpatienceType	Qualifies the class-scoped patience.
replySignalClass	string	Name of the class carrying the reply of a reply signal. Without it, REPLY signals are inert after a round trip.
signalType	negative, reply, catastrophe	For Signal classes: signal type.
openOrClosed	Open, Closed	For Signal classes: whether the signal is open or closed.
targetClass	string	For Signal classes: name of the associated job class.
removalDistribution	Distribution	For Signal classes: distribution for batch removal count.
removalPolicy	random, fcfs, lcfs	For Signal classes: policy for selecting jobs to remove.

Since `SelfLooping` classes specialize `Closed` ones, the writers test for them first; a reader that ignores the `SelfLooping` value would silently obtain an ordinary closed class.

Routing

Routing is specified by a probability matrix. The `routing` object has its `type` field set to `matrix` and its `matrix` object uses composite keys of the form `"fromClass,toClass"`, each mapping to a nested dictionary `{fromNode: {toNode: probability}}`.

```
"routing": {
  "type": "matrix",
  "matrix": {
    "Class1,Class1": {
      "Source": { "Queue1": 0.5, "Queue2": 0.5 },
      "Queue1": { "Sink": 1.0 },
      "Queue2": { "Sink": 1.0 }
    },
    "Class1,Class2": {
      "Queue1": { "Queue2": 0.3 }
    }
  }
}
```

Only non-zero probabilities need to be specified. Same-class routing (e.g., `"Class1,Class1"`) describes routing without class switching; cross-class entries (e.g., `"Class1,Class2"`) describe class-switching transitions.

Routing strategies other than `PROB` and `RAND`, which the matrix already captures, are carried by the model-level `routingStrategies` object:

```
"routingStrategies": { "Router1": { "Class1": "RROBIN" } }
```

Table D.14: Routing strategies

Strategy	**Description**
`RAND`	Random selection with equal probability
`PROB`	Probabilistic routing, given by the routing matrix
`RROBIN`	Round-robin over the destinations
`WRROBIN`	Weighted round-robin; uses `routingWeights`
`JSQ`	Join the shortest queue
`FIRING`	Firing routing, used by Petri net transitions
`SQ`	Shortest queue of d, SQ(d); uses `routingParams`
`DISABLED`	Class may not depart the node

`DISABLED` is emitted explicitly because the routing matrix records only positive probabilities and therefore cannot express it.

Weights for `WRROBIN` are given per node, class and destination:

```
"routingWeights": { "Router1": { "Class1": { "Queue1": 2.0, "Queue2": 1.0 } } }
```

The `routingParams` object mirrors that shape and carries the parameters of the parameterized strategy `SQ`, whose entry is `{"d": int}`.
Omitting `routingParams` for a `SQ` node makes the reader warn and fall back to $d = 2$.

Finite capacity regions

A finite capacity region groups stations under a shared capacity constraint.

Table D.15: Finite capacity region fields

Field	Type	Description
name	string	Region name, unique within the model. Required.
stations	array of object	Names of the stations the region spans; the constraint applies to their joint population. Required.
globalMaxJobs	integer	Maximum total jobs in the region.
globalMaxMemory	integer	Maximum total memory in the region.
classMaxJobs	{key: integer}	Per-class maximum jobs: {className: maxJobs}.
classMaxMemory	{key: integer}	Per-class maximum memory: {className: maxMemory}.
constraintA	array of array of number	Linear constraint matrix A of A*x <= b.
constraintB	array of number	Linear constraint capacity vector b of A*x <= b.
dropRule	{key: DropRule}	Per-class drop rule: {className: rule}.

Rewards

The `rewards` array carries reward definitions for CTMC analysis in a declarative form:

```
"rewards": [ { "name": "q1", "type": "QLen", "node": "Queue1", "class": "Class1" } ]
```

The `type` field is one of `QLen`, `Util`, or `Blocking`, matching the `Reward.queueLength`, `Reward.utilization` and `Reward.blocking` templates. The `class` field is optional; when omitted the reward aggregates over all classes.

Only a reward created through a `Reward.*` template carries the structural metadata needed to reproduce it. A reward defined from a bare function handle, or through `Reward.custom`, is not reproducible from JSON: the writer warns and omits it rather than emit a reward that would be wrong on reload.

D.2.4 Distributions

Distributions are used throughout the format for service times, arrival processes, think times, and host demands. A distribution object always contains a `type` field and one of several parameterization modes.

Direct parameterization

The `params` object provides distribution-specific parameters.

Table D.16: Distribution types and parameters

Type	**Parameters**	**Description**
`Exp`	`lambda`	Exponential of rate λ. `rate` is accepted as a read-side alias
`Det`	`value`	Deterministic
`Erlang`	`lambda, k`	Erlang (per-phase rate λ, k phases)
`HyperExp`	`p: [ ], lambda: [ ]`	Hyperexponential (weights and rates, n phases)
`Coxian`	`mu: [ ], phi: [ ]`	Coxian (phase rates μ and completion probabilities ϕ)
`Gamma`	`alpha, beta`	Gamma (`alpha` shape, `beta` scale)
`Lognormal`	`mu, sigma`	Lognormal
`Normal`	`mu, sigma`	Normal
`Uniform`	`a, b`	Uniform over $[a, b]$
`Pareto`	`alpha, scale`	Pareto (`alpha` shape, `scale` scale)
`Weibull`	`alpha, beta`	Weibull. Note the roles: `alpha` is the *scale* and `beta` the *shape*
`Zipf`	`s, n`	Zipf (exponent, items)
`Geometric`	`p`	Geometric
`Binomial`	`p, n`	Binomial
`Poisson`	`lambda`	Poisson
`Bernoulli`	`p`	Bernoulli
`Discrete Uniform`	`min, max`	Discrete uniform
`Discrete Sampler`	`p: [ ], x: [ ]`	Empirical discrete (probs and values)
`EmpiricalCDF`	`x: [ ], F: [ ]`	Empirical CDF (support and cdf values)
`NHPP`	`breakpoints: [ ], rates: [ ], cyclic`	Non-homogeneous Poisson process, piecewise-constant intensity (cyclic when `cyclic` is true)
`MMPP2`	`lambda0, lambda1, sigma0, sigma1`	Two-state MMPP
`MMDP2`	`r0, r1, sigma0, sigma1`	Two-state MMDP: deterministic rates in the two states, and the two switching rates
`ME`	`alpha: [ ], A: [[ ]]`	Matrix-exponential
`RAP`	`H0: [[ ]], H1: [[ ]]`	Rational arrival process
`DMAP`	`D0: [[ ]], D1: [[ ]]`	Discrete MAP
`BMAP`	`D: [[[ ]]]`	Batch MAP; `D[0]`$= \boldsymbol{D}_0$, `D[1]`$= \boldsymbol{D}_1, \ldots$
`MarkedMMPP`	`D: [[[ ]]], K`	Marked MMPP (M3PP) with K marking types
`Replayer`	`fileName, mean`	Trace replay; see Section D.2.4
`Immediate`	(none)	Zero service time
`Disabled`	(none)	No service (class disabled at this node)

Distributions carrying no `params` block, namely `PH`, `APH`, `MAP`, `MMAP` and `Expolynomial`, are described in Sections D.2.4 and D.2.4.

Parameterization of `MMDP2`. The two-state Markov-modulated deterministic process is parameterized by deterministic *rates* in every codebase: `r0` and `r1` are the rates in the two states, and `sigma0`, `sigma1` the switching rates. The stationary mean rate is $(r_0\sigma_1 + r_1\sigma_0)/(\sigma_0 + \sigma_1)$ and the reported mean is its reciprocal. The Python-native class formerly took inter-arrival *times* `d0`, `d1`; that spelling is no longer written or read.

Fit specification

Instead of direct parameters, a distribution can be constructed by fitting from summary statistics via the `fit` object. This is the preferred mode when the user specifies a distribution by its moments rather than its native parameters.

Table D.17: Fit methods

Field	**Type**	**Description**
`method`	`fitMean, fitMeanAndSCV, fitMeanAndOrder, fitCentral, fitRawMoments`	Fitting method: which moments the parameters are derived from. Required.
`mean`	`number`	Target mean, matched exactly by every method.
`scv`	`number`	Squared coefficient of variation.
`order`	`integer`	Number of phases/stages.
`moments`	`array of number`	The three moments for fitCentral (central) or fitRawMoments (raw). Must have length 3. Alternatively give the named fields: mean/var/skew for fitCentral, m1/m2/m3 for fitRawMoments. Both methods resolve to Cox2 when type is Coxian or Cox2, and to APH otherwise.
`var`	`number`	Variance, for fitCentral given by named fields.
`skew`	`number`	Skewness, for fitCentral given by named fields.
`m1`	`number`	First raw moment, for fitRawMoments given by named fields.
`m2`	`number`	Second raw moment, for fitRawMoments given by named fields.
`m3`	`number`	Third raw moment, for fitRawMoments given by named fields.

The `moments` array must have length 3 for `fitCentral` and `fitRawMoments`; a shorter array is rejected. Both methods resolve to `Cox2` when the `type` is `Coxian` or `Cox2`, and to `APH` otherwise.
Example:

```
{ "type": "Erlang", "fit": { "method": "fitMeanAndOrder", "mean": 2.0, "order": 3 } }
```

Phase-type and MAP representations

Phase-type distributions (`PH`, `APH`, `Coxian`, `Cox2`) can be specified via a `ph` object containing:

- `alpha`: initial probability row vector $\boldsymbol{\alpha}$

- `T`: sub-generator matrix $\boldsymbol{T}$ (negative diagonal, non-negative off-diagonal)

Example:

```
{ "type": "PH", "ph": { "alpha": [1.0, 0.0], "T": [[-2.0, 2.0], [0.0, -1.0]] } }
```

Markovian Arrival Processes are specified via a `map` object containing `D0`, the background transition matrix $\boldsymbol{D}_0$, and `D1`, the arrival transition matrix $\boldsymbol{D}_1$.

Marked MAPs (`MMAP`) instead use an `mmap` object containing `D0` and `D1k`, an array holding one arrival matrix per mark. The aggregate $\boldsymbol{D}_1 = \sum_k \boldsymbol{D}_{1k}$ is rebuilt on load. A 1×1 matrix may be encoded as a bare scalar.

Special distributions

- `Replayer`: replays inter-arrival times from a trace file, given by `params.fileName`. The writer also emits `params.mean` and, when it can fit one, a `ph` block; the reader uses the file when it resolves and otherwise falls back to the fitted phase-type. A trace built from in-memory samples lives in a generated temporary file whose path does not survive a round trip, and the writer warns in that case.
- `Expolynomial`: specified via an `expolynomial` object with `density` (a density expression in expolynomial (GEN) format, such as `"0.5 * x^2 * Exp[-0.3 x]"`), `eft` (earliest firing time) and `lft` (latest firing time, with the string `"Inf"` allowed for an unbounded support).
- `Prior`: a weighted mixture of alternative distributions for posterior or uncertainty analysis, specified by `distributions` (an array of distribution objects) and `probabilities` (an array of weights).
- `Workflow`: an activity graph that is compiled to a phase-type distribution, specified by a `workflow` object holding `activities` (each with a `name` and a nested `distribution`) and `precedences` between them. The compiled phase-type law is what the solvers see; the graph is retained so that a round trip does not lose the structure.

A distribution for which no branch matches is written as its own type name carrying `params.mean` and `params.scv`, with a warning. On reading, an unrecognized `type` carrying a mean and an SCV is reconstructed as an `APH` matching those two moments, again with a warning.

D.2.5 LayeredNetwork models

A `LayeredNetwork` model describes a layered queueing network (LQN). It is organized hierarchically into hosts (processors), tasks, entries, and activities.

Table D.18: LayeredNetwork model fields

Field	Type	Description
`type`	`"LayeredNetwork"`	Discriminator of the model kind. Required.
`name`	`string`	Model name. Required.

Field	Type	Description
hosts	array of LHost	Host (processor) definitions. Canonical key; the MATLAB and Python readers accept 'processors' as a legacy alias, the Java reader does not. Required.
processors	array of LHost	Legacy alias of 'hosts'. Read by MATLAB and Python only.
tasks	array of LTask	Task definitions; each names the host it runs on. Required.
entries	array of LEntry	Entry definitions; each names the task that offers it. Required.
activities	array of LActivity	Activity definitions; each names the task it belongs to. Required.
precedences	array of LPrecedence	Precedence relations over the activities of a task.

The canonical key for the host array is `hosts`; the MATLAB and Python readers accept `processors` as a legacy alias, the Java reader does not.

Hosts

Table D.19: Host fields

Field	Type	Description
name	string	Host (processor) name, unique within the model. Required.
multiplicity	integer	Number of processors (default 1).
scheduling	FCFS, PS, INF, HOL, RAND	Host scheduling (default INF).
quantum	number	Scheduling quantum (time slice) for PS.
speedFactor	number	Processor speed factor (default 1.0).
replication	integer	Number of replicas.

Tasks

Table D.20: Task fields

Field	Type	Description
name	string	Task name, unique within the model. Required.
host	string	Name of the host. Canonical key; the MATLAB and Python readers accept 'processor' as a legacy alias.
processor	string	Legacy alias of 'host'. Read by MATLAB and Python only.
multiplicity	integer	Number of task threads/copies (default 1).
scheduling	REF, FCFS, INF, HOL, POLL	Task scheduling strategy.
priority	integer	Task priority at its host, 0 being the highest.
thinkTime	Distribution	Think time distribution (for REF tasks). Alternative to the thinkTimeMean/thinkTimeSCV pair.

Field	Type	Description
thinkTimeMean	number	Mean think time (for REF tasks).
thinkTimeSCV	number	Think time squared coefficient of variation.
replication	integer	Number of replicas.
fanIn	object	Fan-in from the named task.
fanOut	object	Fan-out to the named task.
taskType	SetupTask, FunctionTask, CacheTask	Specialized task kind. 'FunctionTask' is the legacy name of 'SetupTask' and is still read; setup and delay-off times are accepted on any task.
setupTime	Distribution	Server activation (setup) time distribution. Alternative to the setupTimeMean/setupTimeSCV pair.
setupTimeMean	number	Mean setup time paid when the task resumes from an idle period.
setupTimeSCV	number	Squared coefficient of variation of the setup time.
delayOffTime	Distribution	Delay-off time distribution, i.e. the idle period before a server powers off. Alternative to the delayOffTimeMean/delayOffTimeSCV pair.
delayOffTimeMean	number	Mean idle time the task waits before powering off.
delayOffTimeSCV	number	Squared coefficient of variation of the delay-off time.
totalItems	integer	Number of cacheable items (CacheTask).
cacheCapacity	integer or array of integer	Cache capacity per list level (CacheTask).
replacementStrategy	ReplacementStrategy	Replacement policy (CacheTask).

A think time may also be given as a `thinkTime` distribution object instead of the `thinkTimeMean`/`thinkTimeSCV` pair; likewise `setupTime` and `delayOffTime` accept distribution objects.

Entries

Table D.21: Entry fields

Field	Type	Description
name	string	Entry name, unique within the model. Required.
task	string	Name of parent task. Required.
arrival	Distribution	Open arrival distribution for this entry.
forwarding	array of object	Forwarding to other entries.
entryType	ItemEntry	Specialized entry kind.
totalItems	integer	Number of items (ItemEntry).
accessProb	Distribution	Item popularity distribution (ItemEntry).

Activities

Table D.22: Activity fields

Field	Type	Description
name	string	Activity name, unique within its task. Required.

Field	Type	Description
task	string	Name of parent task. Required.
hostDemand	Distribution	Host demand distribution.
boundToEntry	string	Entry name this activity is bound to. Canonical key; the MATLAB and Python readers accept 'boundTo' as a legacy alias.
boundTo	string	Legacy alias of 'boundToEntry'. Read by MATLAB and Python only.
repliesTo	string	Entry name this activity replies to.
callOrder	string	Call order for this activity's calls.
synchCalls	array of LCall	Synchronous (blocking) calls to entries.
asynchCalls	array of LCall	Asynchronous (non-blocking) calls to entries.

Each call object has an `entry` field (target entry name) and an optional `mean` field (mean number of calls, default 1).

Activity precedences

Precedence relations define control flow within a task. Two schemas are accepted. The first is the *typed* form, written by the MATLAB and Python bridges, in which the pre/post split is implied by the type: a fork names its predecessor first, a join names its successor last.

Table D.23: Typed precedence fields

Field	Type	Description
task	string	Parent task name.
type	Serial, AndFork, AndJoin, OrFork, OrJoin, Loop, ...	Kind of precedence relation between the pre and post activity sets. Required.
activities	array of string	Ordered list of activity names.
probabilities	array of number	For OrFork: branch probabilities. Defaults to uniform.
loopCount	number	For Loop: mean number of iterations.
preActivity	string	For Loop: the loop trigger activity; the loop body is then carried whole in 'activities'.

The second is the *explicit* form, written by the Java bridge, which carries the precedence types verbatim and so needs no shape inference. A precedence object carrying `preActs` or `postActs` is read in this form.

Table D.24: Explicit precedence fields

Field	Type	Description
task	string	Parent task name. Required.
preActs	array of string	Predecessor activity names.
postActs	array of string	Successor activity names.
preType	pre, pre-AND, pre-OR	Pre-condition type (default 'pre').

Field	Type	Description
postType	post, post-AND, post-OR, post-LOOP, post-CACHE	Post-condition type (default 'post').
preParams	array of number	Parameters of the pre-condition, such as an AND-join quorum. Defaults to all predecessors required.
postParams	array of number	Parameters of the post-condition, such as OR-fork probabilities or a loop count.

D.2.6 Workflow models

A `Workflow` model describes an activity graph that compiles to a phase-type distribution. Its `type` field is set to `"Workflow"` and it carries a `name`, an `activities` array of {`name`, `distribution`} objects, and a `precedences` array using the typed precedence schema of Table D.23.

A workflow may equally appear as a distribution, through a `workflow` object with the same `activities` and `precedences` fields.

D.2.7 Environment models

An `Environment` model describes a queueing network operating in a random environment. Each environment stage is associated with a `Network` sub-model.

Table D.25: Environment model fields

Field	Type	Description
type	`"Environment"`	Discriminator of the model kind. Required.
name	string	Model name. Required.
numStages	integer	Number of environment stages.
stages	array of EnvStage	Environment stages, in the order that fixes the 0-based stage indices used by the transitions. Required.
transitions	array of object	Transitions between stages. 'from' and 'to' are 0-based stage indexes, not names. There is no scalar rate field: the stage holding time is a full Distribution object.

A transition object holds `from` and `to` as 0-based *stage indexes*, not stage names, and a `distribution` object (Section D.2.4) giving the stage holding time. There is no scalar rate field: an exponential holding time of rate λ is written `{"type": "Exp", "params": {"lambda": λ}}`.

Environment state not carried by the format

The reset policy of an environment transition (Section 6.6.1) and its state-dependent rate rule are function-valued and are not serialized. The consequences are described in Section D.2.9.

D.2.8 Examples

M/M/1 queue

```
{
  "format": "line-model",
  "version": "1.0",
  "model": {
    "type": "Network",
    "name": "M/M/1",
    "nodes": [
      { "name": "Source", "type": "Source",
        "service": {
          "Class1": { "type": "Exp",
                      "fit": { "method": "fitMean", "mean": 1.0 } } } },
      { "name": "Queue", "type": "Queue", "scheduling": "FCFS",
        "service": {
          "Class1": { "type": "Exp",
                      "fit": { "method": "fitMean", "mean": 0.5 } } } },
      { "name": "Sink", "type": "Sink" }
    ],
    "classes": [
      { "name": "Class1", "type": "Open" }
    ],
    "routing": {
      "type": "matrix",
      "matrix": {
        "Class1,Class1": {
          "Source": { "Queue": 1.0 },
          "Queue": { "Sink": 1.0 }
        }
      }
    }
  }
}
```

Two-task layered queueing network

```
{
  "format": "line-model",
  "version": "1.0",
  "model": {
    "type": "LayeredNetwork",
    "name": "lqn_twotasks",
    "hosts": [
      { "name": "P1", "scheduling": "PS" },
      { "name": "P2", "scheduling": "PS" }
    ],
    "tasks": [
      { "name": "T1", "host": "P1", "multiplicity": 100,
        "scheduling": "REF",
        "thinkTime": { "type": "Erlang",
```

```
            "fit": { "method": "fitMeanAndOrder",
                     "mean": 10, "order": 1 } } },
        { "name": "T2", "host": "P2", "multiplicity": 1,
          "scheduling": "INF" }
      ],
      "entries": [
        { "name": "E1", "task": "T1" },
        { "name": "E2", "task": "T2" }
      ],
      "activities": [
        { "name": "A1", "task": "T1",
          "hostDemand": { "type": "Exp", "params": { "lambda": 1.0 } },
          "boundToEntry": "E1",
          "synchCalls": [ { "entry": "E2" } ] },
        { "name": "A2", "task": "T2",
          "hostDemand": { "type": "Exp", "params": { "lambda": 1.0 } },
          "boundToEntry": "E2", "repliesTo": "E2" }
      ]
    }
  }
```

Random environment

The model below alternates between an `Up` stage and a `Down` stage in which the queue serves at half its nominal rate. Each stage carries a complete `Network` sub-model, and the two stages are connected by exponential transitions.

```
{
  "format": "line-model",
  "version": "1.0",
  "model": {
    "type": "Environment",
    "name": "server_breakdown",
    "numStages": 2,
    "stages": [
      { "name": "Up",
        "model": {
          "type": "Network", "name": "up",
          "nodes": [
            { "name": "Delay", "type": "Delay",
              "service": { "Class1": { "type": "Exp",
                "params": { "lambda": 0.2 } } } },
            { "name": "Queue1", "type": "Queue", "scheduling": "FCFS",
              "service": { "Class1": { "type": "Exp",
                "params": { "lambda": 1.0 } } } }
          ],
          "classes": [
            { "name": "Class1", "type": "Closed",
              "population": 3, "refNode": "Delay" }
          ],
          "routing": {
            "type": "matrix",
            "matrix": {
```

```
                "Class1,Class1": {
                  "Delay": { "Queue1": 1.0 },
                  "Queue1": { "Delay": 1.0 }
                }
              }
            }
          }
        },
        { "name": "Down",
          "model": {
            "type": "Network", "name": "down",
            "nodes": [
              { "name": "Delay", "type": "Delay",
                "service": { "Class1": { "type": "Exp",
                  "params": { "lambda": 0.2 } } } },
              { "name": "Queue1", "type": "Queue", "scheduling": "FCFS",
                "service": { "Class1": { "type": "Exp",
                  "params": { "lambda": 0.5 } } } }
            ],
            "classes": [
              { "name": "Class1", "type": "Closed",
                "population": 3, "refNode": "Delay" }
            ],
            "routing": {
              "type": "matrix",
              "matrix": {
                "Class1,Class1": {
                  "Delay": { "Queue1": 1.0 },
                  "Queue1": { "Delay": 1.0 }
                }
              }
            }
          }
        }
      ],
      "transitions": [
        { "from": 0, "to": 1,
          "distribution": { "type": "Exp", "params": { "lambda": 0.1 } } },
        { "from": 1, "to": 0,
          "distribution": { "type": "Exp", "params": { "lambda": 1.0 } } }
      ]
    }
  }
```

D.2.9 State not carried by the format

Some model state is function-valued and therefore has no JSON representation. A round trip through the format silently drops it, and the reloaded model reverts to the corresponding default. Users must reapply such state programmatically after loading.

Table D.26: Model state not carried by the JSON format

State	Consequence of a round trip
Environment reset policy (Section 6.6.1)	Reverts to the identity rule, so jobs keep their positions across a stage transition instead of being redistributed
Environment state-dependent rate rule (Section 6.6.1)	Reverts to the identity rule. ENV then no longer selects options.method='statedep', so the reloaded model is solved as a plain random environment
State-dependent routing functions	Reverts to the routing matrix
Custom rewards (Section D.2.3)	Omitted with a warning at save time

The reset policy is a per-transition function, supplied as the fourth argument of add_transition, and may differ on each transition. It is not a model-level enumeration and there is no field in the JSON format that selects between clearing and keeping the network state.

D.2.10 Serialization conventions

The following conventions apply to the JSON serialization:

- Matrices are stored as nested JSON arrays, e.g., [[1,2],[3,4]] for a 2×2 matrix. A 1×1 matrix may be encoded as a bare scalar.
- Enumerations cross as names, never as ordinal numbers, since the numeric values of the enumerations differ across codebases.
- NaN values are represented as JSON null.
- Infinity values are clamped to $\pm 10^{308}$, except where a key documents the string "Infinity".
- All identifiers (node names, class names) are strings and must be unique within their scope.
- ClassSwitch nodes that are auto-generated by link() are excluded from serialization.
- An absent key means the corresponding feature is not configured.
- Optional fields may be omitted; the loader applies default values as documented in the schema.

D.3 Command-line interface (line-cli)

The LINE command-line interface (line-cli) provides a standalone tool for solving queueing network models directly from the terminal. This interface is useful for batch processing, integration with external tools, scripting, and environments where interactive use of MATLAB, Python, or Java is not desirable.

D.3.1 Installation

The CLI tool requires:

- Java 8+: The Java Runtime Environment (JRE) version 8 or later.
- Python 3.11+: Python 3.11 or later.
- jline.jar: The LINE solver JAR file, typically located in the `common/` directory.

The CLI is included in the standard LINE distribution. After extracting the release archive, verify the installation by running:

```
python line-cli.py info
```

If the JAR file is not automatically detected, set the `LINE_JAR_PATH` environment variable:

```
export LINE_JAR_PATH=/path/to/jline.jar
```

D.3.2 Basic Usage

The CLI supports several commands for solving models and querying information.

Solving a Model. To solve a queueing network model, use the `solve` command with the optional `-s` (or `--solver`) option to specify the solver algorithm (when omitted, the solver defaults to `auto`):

```
python line-cli.py solve model.jsimg -s mva
```

If you are unsure of which solver to use, the `auto` solver provides an automatic choice. To list all available solvers:

```
python line-cli.py list solvers
```

The following table summarizes the solvers presently interfaced with the cli tool:

Solver	Name	Formats	Notes
auto	Automatic Selection	All	Selects best solver for model
ag	Agent-based (RCAT/INAP)	json/jsim	Reversed-rate product form
ba	Bound Analysis	json/jsim	Closed-form bounds
ctmc	CTMC	json/jsim	Exact, small models only
env	Random Environment	json	Environment models only
fld	Fluid ODE	json/jsim	Fast, approximate
jmt	Java Modelling Tools	json/jsim	External simulation
ldes	LDES	json/jsim/pnml	Simulation-based (alias des)
ln	Layered Network	json/lqnx/xml	For LQN models (MVA layers)
ln.mva, ln.nc, ln.comom	Layered Network	json/lqnx/xml	Layer engine named
lqns	LQN Solver	json/lqnx/xml	External LQNS tool
mam	Matrix Analytic	json/jsim	Supports Fork-Join percentiles
mva	Mean Value Analysis	json/jsim	Fast, general purpose
nc	Normalizing Constant	json/jsim	Exact analysis
qns	QNS	json/jsim	External QNSolver integration
ssa	Stochastic Simulation	json/jsim	State-space sampling
uq	Uncertainty Quantification	json/jsim	Needs --uq-solver

The input format is automatically detected from the file extension. Supported formats include:

- .json: LINE's own portable model format, carrying a , a or an
- .jsimg, .jsim, .jsimw: JMT simulation model formats
- .lqnx, .xml: Layered Queueing Network Solver XML format
- .pnml: place/transition net (ISO/IEC 15909-2)

Models can also be piped via stdin when the input format is specified:

```
cat model.jsimg | python line-cli.py solve -i jsimg -s mva
```

The CLI also supports multiple output formats via the -o or --output-format option:

```
python line-cli.py solve model.jsimg -s mva -o table   # Human-readable table
python line-cli.py solve model.jsimg -s mva -o json    # JSON format
python line-cli.py solve model.jsimg -s mva -o csv     # Comma-separated values
python line-cli.py solve model.jsimg -s mva -o raw     # Raw solver output
```

If unspecified, the table output format is used by default.

To save results to a file, further add the -O option:

```
python line-cli.py solve model.jsimg -s mva -o json -O results.json
```

Analysis Types. The -a or --analysis option specifies which metrics to compute:

```
python line-cli.py solve model.jsimg -s mva -a all   # Average and system ...
metrics (default)
python line-cli.py solve model.jsimg -s mva -a avg   # Average metrics only
python line-cli.py solve model.jsimg -s mva -a sys   # System metrics only
```

Advanced analysis types include:

- `stage`, `chain`, `node`, `nodechain`: stage-, chain-, node-, and node-chain-level averages
- `cdf-respt`, `cdf-passt`: Response/passage time CDFs
- `perct-respt`: Response time percentiles (MAM solver)
- `tran-avg`, `tran-cdf-respt`, `tran-cdf-passt`: transient averages and transient CDFs
- `prob`, `prob-aggr`, `prob-marg`: State probabilities (CTMC/SSA)
- `prob-sys`, `prob-sys-aggr`, `prob-sys-marg`: System state probabilities
- `sample`, `sample-aggr`, `sample-sys`, `sample-sys-aggr`: State trajectory sampling (SSA)
- `reward`, `reward-steady`, `reward-value`: Reward metrics (CTMC)
- `cache`, `item`: cache hit/miss metrics, and the per-item table
- `orbit`, `loss`, `region-loss`, `deadline`: retrial orbit, class-level and finite-capacity-region loss, and deadline-miss metrics
- `normconst`: the log normalizing constant $\log G(N)$ (`nc`, `mva`)
- `busyperiod`: mean busy period of a subnetwork, with `--busyperiod-subnet` naming it and `--busyperiod` the orders (`nc`, `ldes`)
- `sens`: sensitivity of the mean metrics to the service demands
- `generator`, `statevec`, `moments`, `interval`: solver-internal reports (CTMC generator, fluid state vector, moment closure, UQ envelope)

  To list all analysis types:

```
python line-cli.py list analysis
```

D.3.3 Server Modes

The CLI can run as a server to accept solve requests over the network.

WebSocket Server. Start a WebSocket server for real-time communication:

```
python line-cli.py server -p 5863
```

This starts a WebSocket server on port 5863 (default). Clients can connect to `ws://localhost:5863` to submit models and receive results.

REST API Server. The `rest` subcommand is at present an alias of `server`: it execs the JAR with `-p`, so it starts the same WebSocket server on the requested port (8080 by default) and does not answer HTTP requests.

```
python line-cli.py rest -p 8080
```

The HTTP interface is a separate Maven module, `io/rest-api/`, which is not part of the root build and has to be packaged and launched on its own:

```
cd io/rest-api
mvn clean package
java -cp target/line-rest.jar:common/jline.jar jline.rest.LineRestServer --port ...
8080
```

It serves JSON under the base path `/api/v1`, for instance:

- `POST /api/v1/models/solve`: Submit a model for solving
- `POST /api/v1/models/solve/async`: Submit a model as a job, tracked under `/api/v1/jobs`
- `GET /api/v1/health`: Health check endpoint
- `GET /api/v1/solvers`: Solver catalogue and per-solver capabilities

The complete contract, including the analysis and metrics routes, is given by `io/rest-api/openapi.yaml`.

D.3.4 Advanced Options

Stochastic Solver Options. For stochastic solvers (`ldes`, `ssa`, `jmt`), a random seed can be specified for reproducibility:

```
python line-cli.py solve model.jsimg -s ssa -d 12345
```

Probability and Sampling Options. For probability and sampling analysis:

```
# State probability at node 0
python line-cli.py solve model.jsimg -s ctmc -a prob -n 0

# Sample 5000 events from node 1
python line-cli.py solve model.jsimg -s ssa -a sample -n 1 --events 5000

# Compute specific percentiles
python line-cli.py solve model.jsimg -s mam -a perct-respt --percentiles ...
50,90,95,99
```

Output Metrics. The CLI outputs performance metrics organized in two tables.

Average Metrics (per station and job class):

`QLen`	Average queue length (number of jobs)
`Util`	Server utilization (0–1)
`RespT`	Response time (seconds)
`ResidT`	Residence time (seconds)
`Tput`	Throughput (jobs/second)
`ArvR`	Arrival rate (jobs/second)

System Metrics (per chain):

`SysRespT`	End-to-end response time
`SysTput`	System throughput
`SysQLen`	Total jobs in system

D.3.5 Command Reference

Command/Option	Description
`solve <file>`	Solve a queueing model
`info`	Display system information
`list solvers`	List available solvers
`list formats`	List supported formats
`list analysis`	List analysis types
`server`	Start WebSocket server
`rest`	Start WebSocket server on port 8080 (alias of `server`)
`-s, -solver`	Solver algorithm
`-i, -input-format`	Input format (auto-detected)
`-o, -output-format`	Output format (table/json/csv/raw)
`-a, -analysis`	Analysis type(s)
`-d, -seed`	Random seed
`--events`	Maximum simulation events (`ldes`, `ssa`)
`--samples`	Simulation samples / Monte Carlo draws
`--cutoff`	State-space cutoff per open class (`ctmc`, `ssa`)
`--timespan`	Horizon of the transient analyses, `T1`, `T0,T1` or `T0:T1`
`--method`	Algorithm within the chosen solver
`--uq-solver`	Engine `uq` runs at each design point
`--busyperiod-subnet`	Stations forming the `-a busyperiod` subnetwork
`-n, -node`	Node index (for prob/sample)
`-c, -class-idx`	Class index (for prob-marg)
`-v, -verbose`	Verbose output
`-q, -quiet`	Suppress status messages
`-O, -output-file`	Write results to file
`-p, -port`	Server port
`-H, -host`	Server host address

D.3.6 Examples

Sample Output. Running the CLI on a sample open queueing network model:

```
python line-cli.py solve example.json -s mva
```

produces the following output:

```
Solving example.json...

Average Metrics
===============
Station   Class    Metric  Value    Unit
----------------------------------------
Source 1  Class A  QLen    0
Source 1  Class A  Util    0
Source 1  Class A  RespT   0
Source 1  Class A  ResidT  0
```

```
Source 1  Class A  ArvR    0
Source 1  Class A  Tput    2
Source 1  Class B  QLen    0
Source 1  Class B  Util    0
Source 1  Class B  RespT   0
Source 1  Class B  ResidT  0
Source 1  Class B  ArvR    0
Source 1  Class B  Tput    1
Queue 1   Class A  QLen    1.3333
Queue 1   Class A  Util    0.4
Queue 1   Class A  RespT   0.66667
Queue 1   Class A  ResidT  0.44444
Queue 1   Class A  ArvR    2
Queue 1   Class A  Tput    2
Queue 1   Class B  QLen    1
Queue 1   Class B  Util    0.3
Queue 1   Class B  RespT   1
Queue 1   Class B  ResidT  0.33333
Queue 1   Class B  ArvR    1
Queue 1   Class B  Tput    1
Queue 2   Class C  QLen    0.81818
Queue 2   Class C  Util    0.45
Queue 2   Class C  RespT   0.27273
Queue 2   Class C  ResidT  0.27273
Queue 2   Class C  ArvR    3
Queue 2   Class C  Tput    3

System Metrics
==============
Station  Class                    Metric    Value    Unit
------------------------------------------------------------
Chain1   Class A Class B Class C  SysRespT  0.77778
Chain1   Class A Class B Class C  SysTput   3

Execution time: 0.252s
```

Index

Entries are two-level. A member of an enumerated family — a solver, a node type, a scheduling or routing strategy, a distribution, a metric, a model class, an option, a solution method — is listed both under its own name and under its family, so that `FCFS` can be reached either alphabetically or through *scheduling strategies*. Page numbers point at the definition or the explanation of a term, not at every mention of it.