LINE: Queueing Analysis Algorithms

User manual for MATLAB

Last revision: February 20, 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. The tool covered in this manual is implemented in the MATLAB language (https://sf.net/p/line-solver/code/ci/master/tree/matlab), with selected methods written in Java and C/C++.

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), BuTools (http://webspn.hit.bme.hu/~telek/tools/butools/), Q-MAM and SMCSolver (https://win.uantwerpen.be/~vanhoudt/), 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 (DES)

- Fluid/mean-field ordinary differential equations (FLD)

- Matrix analytic methods (MAM)

- Normalizing constant analysis (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). 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).

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.

1.2 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 MATLAB R2025a and later releases and requires the *Statistics and Machine Learning Toolbox*. Some advanced features also require the *Parallel Computing Toolbox*, the *Global Optimization Toolbox*, the *Optimization Toolbox*, and the *Symbolic Math Toolbox*.

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

1.3   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* [38]: G. Horváth and M. Telek. "BuTools 2: A Rich Toolbox for Markovian Performance Evaluation", in *Proc. of VALUETOOLS*, 2017.

- *SMCSolver* [5]: 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* [54]: A. Riska and E. Smirni. "ETAQA Solutions for Infinite Markov Processes with Repetitive Structure", in *INFORMS Journal on Computing*, 19(2):215–228, 2007.

- *KPC-Toolbox* [20]: 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* [4]: 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* [32]: 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* [5]: 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.

1.4   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/

1.5 Copyright and license

Copyright Imperial College London (2012-Present). LINE is freeware and open-source, released under the 3-clause BSD license.
License files of third-party libraries are listed at

https://sf.net/p/line-solver/code/ci/master/tree/matlab/lib/thirdparty

1.6 Acknowledgement

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. Before running LINE you will always need to add the installation folder and its subfolders to the MATLAB path. In order to do so, start MATLAB and change the active directory to the `matlab/` folder, then run

   ```
   lineStart
   ```

3. LINE is now ready to use. For example, you can run the demonstrators using

   ```
   allExamples
   ```

2.1.1 Software requirements

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

- MATLAB version 2025a or later, with the following toolboxes:

  - Global Optimization Toolbox
  - Optimization Toolbox

– Parallel Computing Toolbox

– Statistics and Machine Learning Toolbox

– Symbolic Math Toolbox

These libraries are shipped with LINE:

- Java Modelling Tools (http://jmt.sf.net): version 1.2.4 or later. The latest version is automatically downloaded at the first call of the JMT solver.

- KPC-Toolbox (https://github.com/kpctoolboxteam/kpc-toolbox): version 0.3.4 or later. This release is already included under the lib subfolder.

- MAMSolver (https://www.cs.wm.edu/MAMSolver/): tools for matrix-analytic methods. This release is already included under the lib subfolder.

- M3A (https://github.com/imperial-qore/M3A): version 1.0.0. This release is already included under the lib subfolder.

- BuTools (https://github.com/ghorvath78/butools): version 2.0 or later. This release is already included under the lib subfolder.

- Q-MAM/SMC (https://win.uantwerpen.be/~vanhoudt/tools/QBDfiles.zip). This release is already included under the lib subfolder.

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

- 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 (i.e., it is required that both commands work from within MATLAB using the dos or unix commands without need to specify the paths of the executables).

2.1.2 Documentation

This manual introduces the main concepts to define models in LINE and run its solvers. The document includes in particular several tables that summarize the features currently supported in the modeling language and by individual solvers. Additional resources are as follows:

- MATLAB manual: https://line-solver.sf.net/doc/LINE-matlab.pdf

- Java manual: https://line-solver.sf.net/doc/LINE-java.pdf

- Kotlin manual: https://line-solver.sf.net/doc/LINE-kotlin.pdf

- Python manual: https://line-solver.sf.net/doc/LINE-python.pdf

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

In this section, we present some examples that illustrate how to use LINE. The relevant scripts are included under the `examples/gettingstarted/` folder. The examples describe one of two main available classes of stochastic models within LINE:

- `Network` models are extended queueing networks. Typical instances are open, closed and mixed queueing networks, possibly including advanced features such as class-switching, finite capacity, priorities, non-exponential distributions, and others. Technical background on these models can be found in books such as [8, 43] or in tutorials such as [3, 42].

- `LayeredNetwork` models are layered queueing networks, i.e., models consisting of layers, each corresponding to a `Network` object, which interact through synchronous and asynchronous calls. Technical background on layered queueing networks can be found in [65].

2.2.1   Controlling verbosity

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

```
GlobalConstants.setVerbose(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

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

```
>> MVA(gallery_mm1_tandem).avgTable()
MVA analysis (method: default, lang: matlab) completed. Runtime: 0.027725 seconds. ...
Iterations: 181.
ans =
    Station      JobClass      QLen     Util     RespT     ResidT    ArvR     Tput

    _____     _____     _____    ____     _____    _____    ____     ____

    mySource     myClass           0       0          0         0       0        1
    Queue1       myClass      8.9992     0.9     8.9992    8.9992       1        1
    Queue2       myClass      8.9992     0.9     8.9992    8.9992       1        1
```

The examples in the gallery may also be used as templates to accelerate the definition of basic models. Example 9 later shows an example of gallery instantiation of a $M/E_2/1$ queue.

2.2.3 Tutorial 1: A M/M/1 queue

The M/M/1 queue is a classic model of a queueing system where jobs arrive into an infinite-capacity buffer, wait to be processed in first-come first-serve (FCFS) order, and then leave after service completion. Arrival and service times are assumed to be independent and exponentially distributed random variables.

In this example, we wish to compute average performance measures for the M/M/1 queue. We assume that arrivals come in at rate $\lambda = 1$ job/s, while service has rate $\mu = 2$ job/s. It is known from theory that the exact value of the server utilization in this case is $\rho = \lambda/\mu = 0.5$, i.e., 50%, while the mean response time for a visit is $R = 1/(\mu - \lambda) = 1$s. We wish to verify these values using JMT-based simulation, instantiated through LINE.

The general structure of a LINE script consists of four blocks:

1. Definition of nodes

2. Definition of job classes and associated statistical distributions

3. Instantiation of model topology

4. Solution

For example, the following script solves the M/M/1 model

```
model = Network('M/M/1');
%% Block 1: nodes
source = Source(model, 'Source');
queue = Queue(model, 'Queue', SchedStrategy.FCFS);
sink = Sink(model, 'Sink');
%% Block 2: classes
jobclass = OpenClass(model, 'Class1');
source.setArrival(jobclass, Exp(1));
queue.setService(jobclass, Exp(2));
%% Block 3: topology
model.link(Network.serialRouting(source,queue,sink));
```

```
%% Block 4: solution
AvgTable_native = JMT(model,'seed',23000,'samples',10000).avgTable();
AvgTable = IndexedAvgTable(AvgTable_native);
%% select a particular table row using direct indexing
ARow = AvgTable(queue, jobclass)
%% select a particular table row by node and class label
ARow = AvgTable('Queue', 'Class1')
```

In the example, `source` and `sink` are arrival and departure points of jobs; `queue` is a queueing station with FCFS scheduling; `jobclass` defines an open class of jobs that arrive, get served, and leave the system; `Exp(2.0)` defines an exponential distribution with rate parameter $\lambda = 2.0$; finally, the avg_table command solves for average performance measures with JMT's simulator, using for reproducibility a specific seed for the random number generator.

The result is a table with mean performance measures including: the number of jobs in the station either queueing or receiving service (`QLen`); the utilization of the servers (`Util`); the mean response time for a visit to the station (`RespT`); the mean residence time, i.e. the mean response time cumulatively spent at the station over all visits (`ResidT`); the mean throughput of departing jobs (`Tput`)

```
AvgTable =
  2x7 table
    Station     JobClass     QLen       Util       RespT      ResidT      Tput

    Source      Class1         0          0          0           0       0.99894
    Queue       Class1       0.9555     0.48736    0.95429     0.95429   0.99987
```

One can verify that this matches JMT results by first typing

```
model.jsimgView
```

which will open the model inside JSIMgraph, as shown in Figure 2.1. From this screen, the simulation can be started using the green "play" button in the JSIMgraph toolbar. A pre-defined gallery of classic models is also available, for example

```
model = gallery_mm1
```

returns a M/M/1 queue with 50% utilization.

If we want to select a particular row of the `AvgTable` data structure, we can use direct object-based indexing or filtering methods. In , we wrap the table with `IndexedAvgTable` to enable convenient syntax:

```
>> AvgTable = IndexedAvgTable(AvgTable_native);
>> ARow = AvgTable(queue, jobclass)
ans =
  1x7 table
    Station     JobClass     QLen       Util       RespT      ResidT      Tput

    Queue       Class1       0.9555     0.48736    0.95429     0.95429   0.99987
```
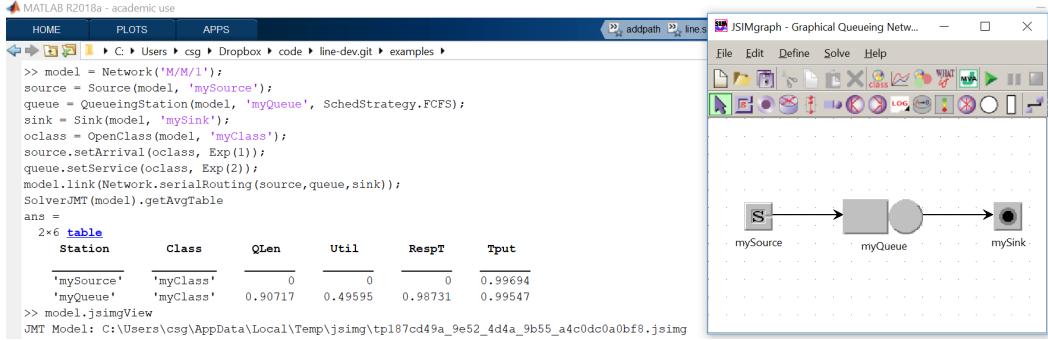
Figure 2.1: M/M/1 example in JSIMgraph launched from MATLAB

For filtering by only station or class, use the `filterBy` method. If we specify only `queue` or `jobclass`, `filterBy` will return all entries corresponding to that station or class. The same indexing works with Station and JobClass objects:

```
>> ARow = AvgTable(queue, jobclass)
```

if we specify only `queue` or only `jobclass`, tget will return all entries corresponding to that station or class.

2.2.4 Tutorial 2: A multiclass M/G/1 queue

We now consider a more challenging variant of the first example. We assume that there are two classes of incoming jobs with non-exponential service times. For the first class, service times are Erlang distributed with unit rate and variance $1/3$; they are instead read from a trace for the second class. Both classes have exponentially distributed inter-arrival times with mean $2s$.

To run this example, let us first change the working directory to the `examples/` folder. Then we specify the node block

```
model = Network('M/G/1');
source = Source(model,'Source');
queue = Queue(model, 'Queue', SchedStrategy.FCFS);
sink = Sink(model,'Sink');
```

The next step consists in defining the classes. We fit automatically from mean and squared coefficient of variation (i.e., SCV=variance/mean^2) an Erlang distribution and use the `Replayer` distribution to request that the specified trace is read cyclically to obtain the service times of class 2

```
jobclass1 = OpenClass(model, 'Class1');
jobclass2 = OpenClass(model, 'Class2');
```

```
source.setArrival(jobclass1, Exp(0.5));
source.setArrival(jobclass2, Exp(0.5));

queue.setService(jobclass1, Erlang.fitMeanAndSCV(1, 1/3));
queue.setService(jobclass2, Replayer('example_trace.txt'));
```

Note that the example_trace.txt file consists of a single column of doubles, each representing a service time value, e.g.,

```
1.2377474e-02
4.4486055e-02
1.0027642e-02
2.0983173e-02
...
```

We now specify a linear route through source, queue, and sink for both classes

```
P = model.initRoutingMatrix();
P{jobclass1} = Network.serialRouting(source,queue,sink);
P{jobclass2} = Network.serialRouting(source,queue,sink);
model.link(P);
```

and solve the model with JMT

```
>>jmtAvgTable = JMT(model,'seed',23000).avgTable()
jmtAvgTable =
  4x7 table
    Station     JobClass     QLen        Util      RespT      ResidT      Tput

    _____    _____    _____    _____    _____    _____    _____

    Source      Class1           0           0          0          0    0.50017
    Source      Class2           0           0          0          0    0.49114
    Queue       Class1     0.86153      0.4984     1.7389     1.7389    0.49953
    Queue       Class2     0.43751    0.049184    0.85879    0.85879    0.49064
```

We wish now to validate this value against an analytical solver. Since jobclass2 has trace-based service times, we first need to revise its service time distribution to make it analytically tractable, e.g., we may ask LINE to fit an acyclic phase-type distribution [7] based on the trace

```
queue.setService(jobclass2, Replayer('example_trace.txt').fitAPH());
```

We can now use a Continuous Time Markov Chain (CTMC) to solve the system, but since the state space is infinite in open models, we need to truncate it to be able to use this solver. For example, we may restrict to states with at most 2 jobs in each class, checking with the verbose option the size of the resulting state space

```
>> ctmcAvgTable2 = CTMC(model,'cutoff',2,'verbose',true).avgTable()
State space size: 46 states.
CTMC analysis completed in 0.096734 sec
```

```
ctmcAvgTable2 =
  4x7 table
    Station     JobClass     QLen        Util        RespT       ResidT      Tput

    _____    _____     _____    _____    _____    _____    _____

    Source      Class1              0           0           0           0     0.44948
    Source      Class2              0           0           0           0     0.48424
    Queue       Class1        0.56734     0.44948      1.2863      1.2863     0.44107
    Queue       Class2        0.24456    0.048942     0.51396     0.51396     0.47583
```

However, we see from the comparison with JMT that the errors of CTMC are rather large. Since the truncated state space consists of just 46 states, we can further increase the cutoff to 4, trading a slower solution time for higher precision

```
>> ctmcAvgTable4 = CTMC(model,'cutoff',4,'verbose',true).avgTable()
State space size: 626 states.
CTMC analysis completed in 1.051784 sec
ctmcAvgTable4 =
  4x7 table
    Station     JobClass     QLen        Util        RespT       ResidT      Tput

    _____    _____     _____    _____    _____    _____    _____

    Source      Class1              0           0           0           0     0.49215
    Source      Class2              0           0           0           0     0.49626
    Queue       Class1         0.7958     0.49215      1.6187      1.6187     0.49162
    Queue       Class2        0.37558    0.050157     0.75763     0.75763     0.49573
```

To gain more accuracy, we could either keep increasing the cutoff value or, if we wish to compute an exact solution, we may call the matrix-analytic method (MAM) solver instead. MAM uses the repetitive structure of the CTMC to exactly analyze open systems with an infinite state space, calling

```
>> mamAvgTable = MAM(model).avgTable()
mamAvgTable =
  4x7 table
    Station     JobClass     QLen        Util        RespT       ResidT      Tput

    _____    _____     _____    _____    _____    _____    ____

    Source      Class1              0           0           0           0     0.5
    Source      Class2              0           0           0           0     0.5
    Queue       Class1        0.87646         0.5      1.7529      1.7529     0.5
    Queue       Class2          0.427    0.050536     0.85399     0.85399     0.5
```

The current MAM implementation is primarily constructed on top of the BuTools solver [38], the SMC solver [5], and MAMSolver (https://www.cs.wm.edu/MAMSolver/).

2.2.5 Tutorial 3: Machine interference problem

Closed models involve jobs that perpetually cycle within a network of queues. The machine interference problem is a classic example, in which a group of repairmen is tasked with fixing machines as they break and the goal is to choose the optimal size of the group. We here illustrate how to evaluate the performance

of a given group size. We consider a scenario with $S = 2$ repairmen, with machines that break down at a
rate of 0.5 failed machines/week, after which a machine is fixed in an exponential distributed time with rate
4.0 repaired machines/week. There are a total of $N = 3$ machines.

Suppose that we wish to obtain an exact numerical solution using Continuous Time Markov Chains
(CTMCs). The above model can be analyzed as follows:

```
model = Network('MRP');
%% Block 1: nodes
delay = Delay(model,'WorkingState');
queue = Queue(model, 'RepairQueue', SchedStrategy.FCFS);
queue.setNumberOfServers(S);
%% Block 2: classes
cclass = ClosedClass(model, 'Machines', N, delay);
delay.setService(cclass, Exp(0.5));
queue.setService(cclass, Exp(4.0));
%% Block 3: topology
model.link(Network.serialRouting(delay,queue));
%% Block 4: solution
solver = CTMC(model);
ctmcAvgTable = solver.avgTable()
```

Here, `delay` appears in the constructor of the closed class to specify that a job will be considered completed
once it returns to the delay (i.e., the machine returns in working state). We say that the delay is thus the
reference station of `cclass`. The above code prints the following result

```
ctmcAvgTable =
  2x7 table
      Station        JobClass      QLen        Util        RespT       ResidT       Tput

    WorkingState     Machines     2.6648      2.6648           2            2       1.3324
    RepairQueue      Machines     0.33516     0.16655      0.25154      0.25154     1.3324
```

As before, we can inspect and analyze the model in JSIMgraph using the command

```
model.jsimgView()
```

Figure 2.2 illustrates the result, demonstrating the automated definition of the closed class.

We can now also inspect the CTMC more in the details as follows

```
StateSpace = model.stateSpace()
InfGen = full(solver.generator())
```

which produces in output the state space of the model and the infinitesimal generator of the CTMC

```
StateSpace =
     0     1     2
     1     0     2
```
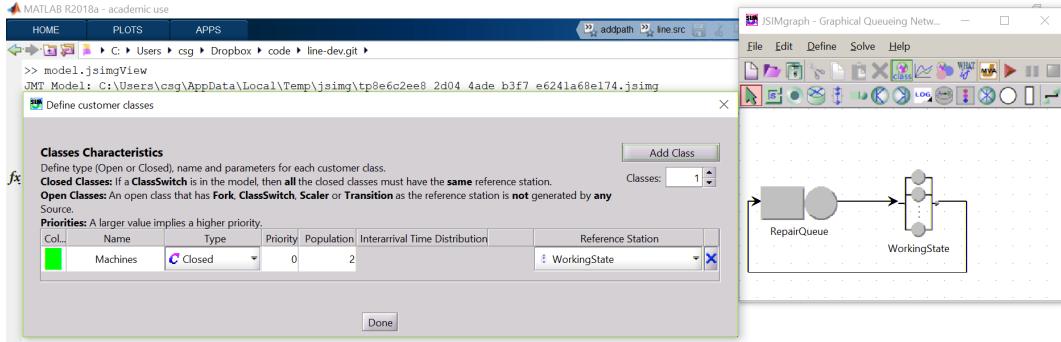
Figure 2.2: Machine interference model in JSIMgraph

```
        2       0       1
        3       0       0
InfGen =
    -8.0000     8.0000           0           0
     0.5000    -8.5000      8.0000           0
          0     1.0000     -5.0000      4.0000
          0          0      1.5000     -1.5000
```

For example, the first state (`0 1 2`) consists of two components: the initial `0` denotes the number of jobs in service in the `delay`, while the remaining part is the state of the FCFS `queue`. In the latter, the `1` means that a job of class 1 (the only class in this model) is in the waiting buffer, while the `2` means that there are two jobs in service at the `queue`.

As another example, the second state (`1 0 2`) is similar, but one job has completed at the queue and has then moved to the `delay`, concurrently triggering an admission in service for the job that was in the queue buffer. As a result of this, the buffer is now empty. The corresponding transition rate in the infinitesimal generator matrix is row 1 and column 2 of `InfGen`, which has value 8.0, that is the sum of the completion rates at the queue for each server in the first state, and where indexes 1 and 2 are the rows in `StateSpace` associated to the source and destination states.

On this and larger infinite generators, we may also list individual non-zero transitions as follows

```
>> model.printInfGen(InfGen,StateSpace)
[0 1 2]->[1 0 2]: 8.000000
[1 0 2]->[0 1 2]: 0.500000
[1 0 2]->[2 0 1]: 8.000000
[2 0 1]->[1 0 2]: 1.000000
[2 0 1]->[3 0 0]: 4.000000
[3 0 0]->[2 0 1]: 1.500000
```

The above printout helps in matching the state transitions to their rates.

To avoid having to inspect the `StateSpace` variable to determine to which station a particular column

refers to, we can alternatively use the more general invocation

```
>> [StateSpace,nodeStateSpace] = solver.stateSpace();
>> nodeStateSpace{delay}
ans =
     0
     1
     2
     3
>> nodeStateSpace{queue}
ans =
     1     2
     0     2
     0     1
     0     0
```

which automatically splits the state space into its constituent parts for each stateful node.

A further observation is that model.stateSpace() forces the regeneration of the state space at each invocation, whereas the equivalent function in the CTMC solver, solver.stateSpace(), returns the state space cached during the solution of the CTMC.

2.2.6   Tutorial 4: Round-robin load-balancing

In this example we consider a system of two parallel processor-sharing queues and we wish to study the effect of load-balancing on the average performance of an open class of jobs. We begin as usual with the node block, where we now include a special node, called the Router, to control the routing of jobs from the source into the queues:

```
model = Network('RRLB');
source = Source(model, 'Source');
lb = Router(model, 'LB');
queue1 = Queue(model, 'Queue1', SchedStrategy.PS);
queue2 = Queue(model, 'Queue2', SchedStrategy.PS);
sink  = Sink(model, 'Sink');
```

Let us then define the class block by setting exponentially-distributed inter-arrival times and service times, e.g.,

```
jobclass = OpenClass(model, 'Class1');
source.setArrival(jobclass, Exp(1));
queue1.setService(jobclass, Exp(2));
queue2.setService(jobclass, Exp(2));
```

We now wish to express the fact that the router applies a round-robin strategy to dispatch jobs to the queues. Since this is now a non-probabilistic routing strategy, we need to adopt a slightly different style to declare the routing topology as we cannot specific anymore routing probabilities. First, we indicate the connections between the nodes, using the addLinks function:
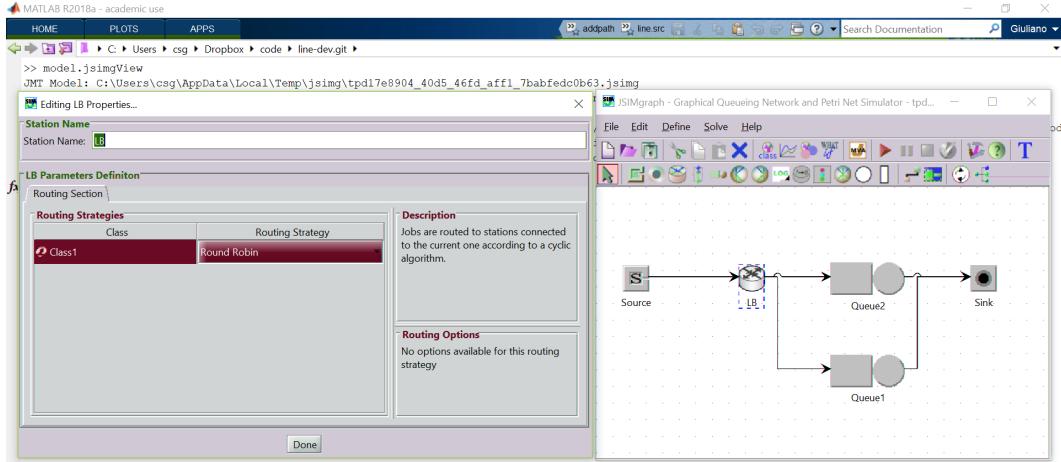
Figure 2.3: Load-balancing model

```
model.addLinks([source, lb;
                lb,     queue1;
                lb,     queue2;
                queue1, sink;
                queue2, sink]);
```

At this point, all nodes are automatically configured to route jobs with equal probabilities on the outgoing links (`RoutingStrategy.RAND` policy). If we solve the model at this point, we see that the response time at the queues is around $0.66s$.

```
>> jmtAvgTable = JMT(model,'seed',23000).avgTable()
jmtAvgTable =
  3x7 table
    Station    JobClass    QLen      Util       RespT      ResidT     Tput
    _____    _____    _____   _____    _____    _____    _____

    Source     Class1            0         0          0          0     1.0135
    Queue1     Class1      0.31612   0.24682    0.65411    0.65411      0.501
    Queue2     Class1      0.33403   0.25076    0.68406    0.34203    0.50413
```

After resetting the internal data structures, which is required before modifying a model we can require LINE to solve again the model using this time a round-robin policy at the router.

```
model.reset()
lb.setRouting(jobclass, RoutingStrategy.RROBIN);
```

A representation of the model at this point is shown in Figure 2.3.

Lastly, we run again JMT and find that round-robin produces a visible decrease in response times, which are now around $0.56s$.

```
>> jmtAvgTableRR = JMT(model,'seed',23000).avgTable()
jmtAvgTableRR =
  3x7 table
    Station     JobClass     QLen        Util       RespT      ResidT       Tput

    _____    _____    _____    _____    _____    _____    _____

    Source      Class1             0           0           0           0     1.0089
    Queue1      Class1       0.30429     0.26118     0.58482     0.58482     0.50526
    Queue2      Class1       0.29282     0.24397     0.57293     0.28647     0.50526
```

2.2.7 Tutorial 5: Modelling a re-entrant line

Let us now consider a simple example inspired to the classic problem of modeling *re-entrant lines*. This arises in manufacturing systems where parts (i.e., jobs) re-enter multiple times a machine (i.e., a queueing station), asking at each visit a different class of service. This implies, for example, that the service time at every visit could feature a different mean or a different distribution compared to the previous visits, thus modeling a different stage of processing.

To illustrate this, consider for example a degenerate model composed of a single FCFS queue and K classes. In this model, a job that completes processing in class k is routed back at the tail of the queue in class $k + 1$, unless $k = K$ in which case the job re-enters in class 1.

We take the following assumptions: $K = 3$ and class k has an Erlang-2 service time distribution at the queue with mean equal to k; the system starts with $N_1 = 1$ jobs in class 1 and zero jobs in all other classes.

```
model = Network('RL');
queue = Queue(model, 'Queue', SchedStrategy.FCFS);

K = 3; N = [1,0,0];
for k=1:K
    jobclass{k} = ClosedClass(model, ['Class',int2str(k)], N(k), queue);
    queue.setService(jobclass{k}, Erlang.fitMeanAndOrder(k,2));
end

P = model.initRoutingMatrix();
P{jobclass{1},jobclass{2}}(queue,queue) = 1.0;
P{jobclass{2},jobclass{3}}(queue,queue) = 1.0;
P{jobclass{3},jobclass{1}}(queue,queue) = 1.0;
model.link(P);
```

The corresponding JMT model is shown in Figure 2.4, where it can be seen that the class-switching rule is automatically enforced by introduction of a ClassSwitch node in the network.
We can now simulate the performance indexes for the different classes, for example using LINE's normalizing constant solver (NC)

```
>> ncAvgTable = NC(model).avgTable()
ncAvgTable =
```
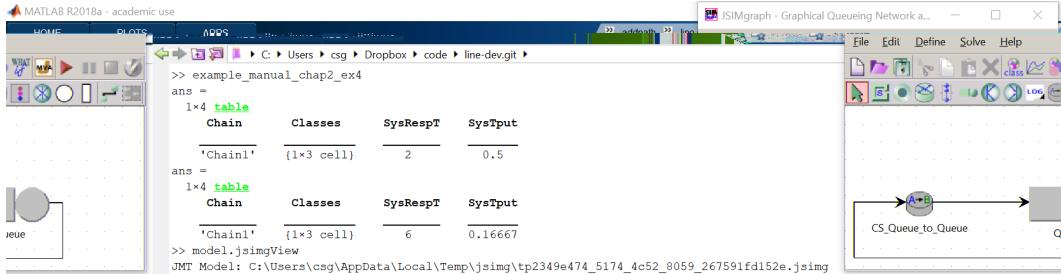
Figure 2.4: Re-entrant lines as an example of class-switching

Station	JobClass	QLen	Util	RespT	ResidT	Tput
Queue	Class1	0.16667	0.16667	1	0.33333	0.16667
Queue	Class2	0.33333	0.33333	2	0.66667	0.16667
Queue	Class3	0.5	0.5	3	1	0.16667

Suppose now that the job is considered completed, for the sake of computation of system performance metrics, only when it departs the queue in class K (here Class3). By default, LINE will return *system-wide* performance metrics using the avgSysTable method, i.e.,

```
>> ncAvgSysTable = NC(model).avgSysTable
ncAvgSysTable =
  1x4 table
    Chain          JobClasses          SysRespT      SysTput

    Chain1      {1x3 categorical}         2            0.5
```

This method identifies the model *chains*, i.e., groups of classes that can exchange jobs with each other, but not with classes in other chains. Since the job can switch into any of the three classes, in this model there is a single chain comprising the three classes.

To see the composition of Chain1, we look at the second column entry

```
>> ncAvgSysTable.JobClasses{1}
ans =
  1x3 categorical array
    Class1      Class2      Class3
```

We see that the throughput of the chain is 0.5, which means that LINE is counting every departure from the queue in any class as a completion for the whole chain. This is incorrect for our model since we want to count completions only when jobs depart in Class3. To require this behavior, we can tell to the solver that passages for classes 1 and 2 through the reference station should not be counted as completions

```
jobclass{1}.completes = false;
```

```
    jobclass{2}.completes = false;
```

This modification then gives the correct chain throughput, matching the one of `Class3` alone

```
>> ncAvgSysTable2 = NC(model).avgSysTable
ncAvgSysTable =
  1x4 table
    Chain          JobClasses          SysRespT     SysTput
    _____      _____      _____     _____

    Chain1     {1x3 categorical}          6         0.16667
```

2.2.8   Tutorial 6: A queueing network with caching

In this more advanced example, we show how to include in a queueing network a cache adopting a least-recently used (LRU) replacement policy. Under LRU, upon a cache miss the least-recently accessed item will be discarded to make room for the newly requested item.

   We consider a cache with a capacity of 50 items, out of a set of 1000 cacheable items. Items are accessed by jobs visiting the cache according to a Zipf-like law with exponent $\alpha = 1.4$ and defined over the finite set of items. A client cyclically issues requests for the items, waiting for a reply before issuing the next request. We assume that a cache hit takes on average $0.2ms$ to process, while a cache hit takes $1ms$. We ask for the average request throughput of the system, differentiated across hits and misses.

Node block   As usual, we begin by defining the nodes. Here a delay node will be used to describe the time spent by the requests in the system, while the cache node will determine hits and misses:

```
model = Network('QNC');
% Block 1: nodes
clientDelay = Delay(model, 'Client');
cacheNode = Cache(model, 'Cache', 1000, 50, ReplacementStrategy.LRU);
cacheDelay = Delay(model, 'CacheDelay');
```

Class block   We define a set of classes to represent the incoming requests (`clientClass`), cache hits (`hitClass`) and cache misses (`missClass`). These classes need to be closed to ensure that there is a single outstanding request from the client at all times:

```
% Block 2: classes
clientClass = ClosedClass(model, 'ClientClass', 1, clientDelay, 0);
hitClass = ClosedClass(model, 'HitClass', 0, clientDelay, 0);
missClass = ClosedClass(model, 'MissClass', 0, clientDelay, 0);
```

We then assign the processing times, using the `Immediate` distribution to ensure that the client issues immediately the request to the cache:

```
    clientDelay.setService(clientClass, Immediate());
    cacheDelay.setService(hitClass, Exp.fitMean(0.2));
    cacheDelay.setService(missClass, Exp.fitMean(1));
```

The next step involves specifying that the request uses a Zipf-like distribution (with parameter $\alpha = 1.4$) to select the item to read from the cache, out of a pool of 1000 items

```
    cacheNode.setRead(clientClass, Zipf(1.4,1000));
```

Finally, we ask that the job should become of class hitClass after a cache hit, and should become of class missClass after a cache miss:

```
    cacheNode.setHitClass(clientClass, hitClass);
    cacheNode.setMissClass(clientClass, missClass);
```

Topology block   Next, in the topology block we setup the routing so that the request, which starts in clientClass at the clientDelay, then moves from there to the cache, remaining in clientClass

```
    % Block 3: topology
    P = model.initRoutingMatrix();
    P{clientClass, clientClass}(clientDelay, cacheNode) = 1.0;
```

Internally to the cache, the job will switch its class into either hitClass or missClass. Upon departure in one of these classes, we ask it to join in the same class cacheDelay for further processing

```
    P{hitClass, hitClass}(cacheNode, cacheDelay) = 1.0;
    P{missClass, missClass}(cacheNode, cacheDelay) = 1.0;
```

Lastly, the job returns to clientDelay for completion and start of a new request, which is done by switching its class back to clientClass

```
    P{hitClass, clientClass}(cacheDelay, clientDelay) = 1.0;
    P{missClass, clientClass}(cacheDelay, clientDelay) = 1.0;
```

The above routing strategy is finally applied to the model

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

Solution block   To solve the model, since JMT does not support cache modeling, we use the native simulation engine provided within LINE, the SSA solver:

```
    % Block 4: solution
    ssaAvgTable = SSA(model,'samples',2e4,'seed',1,'verbose',true).avgTable()
```

The above script produces the following result

```
SSA samples:  20000
SSA analysis (method: default, lang: matlab) completed.
ssaAvgTable =
  3x 8 table
    Station        JobClass       QLen       Util       RespT      ResidT      ArvR ...
    Tput

    _____     _____    _____    _____    _____    _____    ____  ...

    _____

    Client         ClientClass           0          0    1e-08           0       0  ...
    2.8364
    CacheDelay     HitClass        0.43791    0.43791      0.2    0.15639       0  ...
    2.1895
    CacheDelay     MissClass       0.56209    0.56209        1    0.21803       0  ...
    0.56209
```

The departing flows from the `cacheDelay` are the miss and hit rates. Thus, the hit rate is 2.4554 jobs per unit time, while the miss rate is 0.50892 jobs per unit time.

Let us now suppose that we wish to verify the result with a longer simulation, for example with 10 times more samples. To this aim, we can use the automatic parallelization of `SSA` based on MATLAB's `spmd` construct

```
ssaAvgTablePara = SSA(model,'para','samples',2e4,'seed',1).avgTable()
```

This gives us a rather similar result, when run on a dual-core machine

```
Starting parallel pool (parpool) using the 'local' profile ...
connected to 2 workers.
ssaAvgTablePara =
 3x6 table
    Station        JobClass       QLen       Util       RespT      ResidT      Tput

    _____     _____    _____    _____    _____    _____    _____
    Client         ClientClass           0          0        0           0    3.0652
    CacheDelay     HitClass        0.49871    0.49871      0.2    0.16689    2.4935
    CacheDelay     MissClass       0.50129    0.50129        1    0.16553    0.50129
```

The execution time is longer than usual at the first invocation of the parallel solver due to the time needed by MATLAB to bootstrap the parallel pool, in this example around 22 seconds. Successive invocations of parallel SSA normally take much less, with this example around 7 seconds each.

2.2.9   Tutorial 7: Response time distribution and percentiles

In this example we illustrate the computation of response time percentiles in a queueing network model. We begin by instantiating a simple closed model consisting of a delay followed by a processor-sharing queueing station.

```
model = Network('Model');
% Block 1: nodes
node{1} = Delay(model, 'Delay');
node{2} = Queue(model, 'Queue1', SchedStrategy.PS);
```

There is a single class consisting of 5 jobs that circulate between the two stations, taking exponential service times at both.

```
jobclass{1} = ClosedClass(model, 'Class1', 5, node{1}, 0);
node{1}.setService(jobclass{1}, Exp(1.0));
node{2}.setService(jobclass{1}, Exp(0.5));

model.link(Network.serialRouting(node{1},node{2}));
```

We now wish to compare the response time distribution at the PS queue computed analytically with a fluid approximation against the simulated values returned by JMT. To do so, we call the `getCdfRespT` method

```
% Block 4: solution
RDfluid = FLD(model).cdfRespT();
RDsim = JMT(model,'seed',23000,'samples',1e4).cdfRespT();
```

The returned data structures, `RDfluid` and `RDsim`, are cell arrays consisting of a 2-column matrix. The element in position $\{i, r\}$ of the cell array describes the response times at station i for class r. The two columns within such matrices are defined similar to the output of MATLAB's `ecdf` function. The first column represents the cumulative distribution function (CDF) value $F(t) = Pr(T \leq t)$, where T is the random variable denoting the response time, while t is the percentile appearing in the corresponding entry of the second column.

For example, to plot the complementary CDF $1 - F(t)$ we can use the following code

```
% Plot results
semilogx(RDsim{2,1}(:,2),1-RDsim{2,1}(:,1),'r'); hold all;
semilogx(RDfluid{2,1}(:,2),1-RDfluid{2,1}(:,1),'--');
legend('jmt-transient','fluid-steady','Location','Best');
ylabel('Pr(T > t)'); xlabel('time t');
```

The graph shows that, although the simulation refers to a transient, while the fluid approximation refers to steady-state, there is a tight matching between the two response time distributions. which produces the graph shown in Figure 2.5. The graph shows that, although the simulation refers to a transient, while the fluid approximation refers to steady-state, there is a tight matching between the two response time distributions.

We can readily compute the percentiles from the `RDfluid` and `RDsim` data structures, e.g., for the 95th and 99th percentiles of the simulated distribution

```
>> prc95=max(RDsim{2,1}(RDsim{2,1}(:,1)<0.95,2))
prc95 =
```
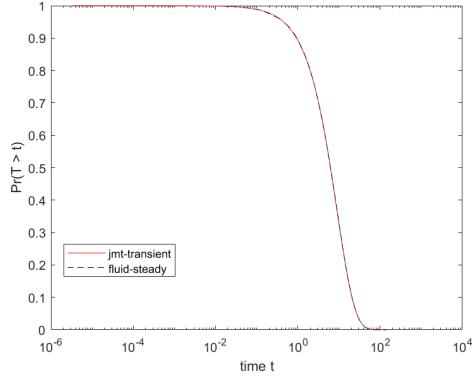
Figure 2.5: Comparison of simulated response time distribution and its fluid approximation

```
    27.0222
>> prc99=max(RDsim{2,1}(RDsim{2,1}(:,1)<0.99,2))
prc99 =
    41.8743
```

That is, 95% of the response times at the PS queue (node 2, class 1) are less than or equal to 27.0222 time units, while 99% are less than or equal to 41.8743 time units.

2.2.10   Tutorial 8: Optimizing a performance metric

In this example, we show how to optimize with the help of LINE a performance metric. We wish to find the optimal routing probabilities that minimize average response times for two parallel processor sharing queues. We assume that jobs are fed by a delay station, arranged with the two queues in a closed network topology.

We first define a MATLAB function (e.g., ex8.m) with header

```
function p_star = ex8()
```

Within the function definition, we instantiate the two queues and the delay station

```
model = Network('LoadBalCQN');
% Block 1: nodes
delay = Delay(model,'Think');
queue1 = Queue(model, 'Queue1', SchedStrategy.PS);
queue2 = Queue(model, 'Queue2', SchedStrategy.PS);
```

We assume that 16 jobs circulate among the nodes, and that the service rates are $\sigma = 1$ jobs per unit time at the delay, and $\mu_1 = 0.75$ and $\mu_2 = 0.50$ at the two queues:

```
% Block 2: classes
cclass = ClosedClass(model, 'Job1', 16, delay);
delay.setService(cclass, Exp(1));
queue1.setService(cclass, Exp(0.75));
queue2.setService(cclass, Exp(0.50));
```

We initially setup a topology with arbitrary values for the routing probabilities between delay and queues, ensuring that jobs completing at the queues return to the delay:

```
% Block 3: topology
P = model.initRoutingMatrix();
P{cclass}(queue1, delay) = 1.0;
P{cclass}(queue2, delay) = 1.0;
model.link(P);
```

We now return the system response time for the jobs as a function of the routing probability p to choose queue 1 instead of queue 2:

```
% Block 4: solution
    function R = objFun(p)
        P{cclass}(delay, queue1) = p;
        P{cclass}(delay, queue2) = 1-p;
        model.link(P);
        R = MVA(model,'exact','verbose',false).avgSysRespT;
    end
```

Lastly, we optimize the function we defined

```
    p_opt = fminbnd(@(p) objFun(p), 0,1)
```

In the above listing, `objFun` first updates the routing topology, prior to obtaining the corresponding system response time. To search for the optimal routing probability p, we have also used MATLAB's `fminbnd` which restricts the search range in the interval $[0, 1]$. Lastly, we optimize the function we defined We are now ready to run `ex8`. The execution returns the optimal value 0.6104878504366782.

2.2.11 Tutorial 9: Studying a departure process

This examples illustrates LINE's support for extracting simulation data about particular events in an extended queueing network, such as departures from a particular queue.

Our goal is to obtain the squared coefficient of variation of the inter-departure times from a $M/E_2/1$ queue, which has Poisson arrivals and 2-phase Erlang distributed service times.

Because this is a classic model, we can find it in LINE's model gallery. The additional return parameters (e.g., `source`,`queue`, ...) provide handles to the entities within the model.

```
    [model,source,queue,sink,jobclass] = gallery_merl1;
```

We now extract 50,000 samples from simulation based on the underpinning continuous-time Markov chain

```
solver = CTMC(model,'cutoff',150,'seed',23000);
sa = solver.sampleSysAggr(1e5);
```

The returned data structure supplies information about the stateful nodes (here source and queue) at each of the 50,000 instants of sampling, together with the events that have been collected at these instants.

```
>> sa
sa =
  struct with fields:

        handle: {[1x1 Source]  [1x1 Queue]}
             t: [50000x1 double]
         state: {[Inf]  [50000x1 double]}
         event: {1x100000 cell}
   isaggregate: 1
```

As an example, the first two events occur both at timestamp 0 and indicate a departure event from node 1 (the type EventType.DEP maps to event: DEP) followed by an arrival event at node 2 (the type EventType.ARV maps to event: ARV) which accepts it always (prob: 1).

```
>> sa.event{1}
ans =
  Event with properties:

     node: 1
    event: DEP
    class: 1
     prob: NaN
    state: []
        t: 0
>> sa.event{2}
ans =
  Event with properties:

     node: 2
    event: ARV
    class: 1
     prob: 1
    state: []
        t: 0
```
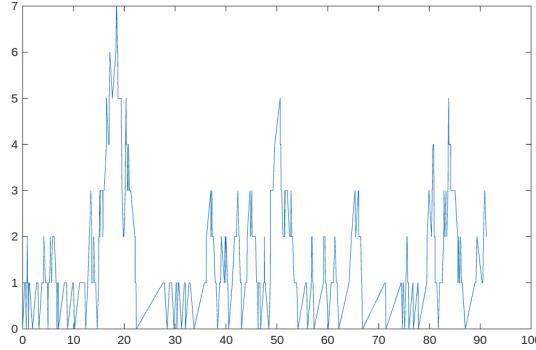
We may also plot the first 300 events as follows

```
plot(sa.t(1:300),sa.state{queue}(1:300));
```

that after adding axes labels gives the following figure

Figure 2.6: A sample path for a $M/E_2/1$ queueing station

We are now ready to filter the timestamps of events related to departures from the `queue` node

```
ind = model.getNodeIndex(queue);
filtEvent = cellfun(@(c) c.node == ind && c.event == EventType.DEP, sa.event);
```

Followed by a calculation of the time series of inter-departure times

```
interDepTimes = diff(cellfun(@(c) c.t, {sa.event{filtEvent}}));
```

We may now for example compute the squared coefficient of variation of this process

```
SCVdEst = var(interDepTimes)/mean(interDepTimes)^2
```

which evaluates to 0.8750. Using Marshall's exact formula for the $GI/G/1$ queue [45], we get a theoretical value of 0.8750. The calculation is included in the script associated to the example (`tut_09_dep_process_analysis.m`).

2.2.12 Tutorial 10: Basic layered queueing network

This example demonstrates how to model and analyze a basic layered queueing network (LQN) representing a simple client-server application with two tiers: a client layer and a database layer. The LQN consists of a client processor P1 with a reference task T1 (10 users), a database processor P2 with an infinite server task T2, synchronous calls from the client to the database, exponential service times at both layers.

We start by creating the layered network instance:

```
model = LayeredNetwork('ClientDBSystem');
```

Next, we define the processors and tasks:

```
% Create processors
P1 = Processor(model, 'ClientProcessor', 1, SchedStrategy.PS);
P2 = Processor(model, 'DBProcessor', 1, SchedStrategy.PS);
```

```
% Create tasks
T1 = Task(model, 'ClientTask', 10, SchedStrategy.REF).on(P1);
T1.setThinkTime(Exp.fitMean(5.0)); % 5-second think time
T2 = Task(model, 'DBTask', Inf, SchedStrategy.INF).on(P2);
```

Now we define the entries that represent service interfaces:

```
E1 = Entry(model, 'ClientEntry').on(T1);
E2 = Entry(model, 'DBEntry').on(T2);
```

Finally, we define the activities that specify the service demand and the synchronous calls

```
% Client activity: processes request and calls DB
A1 = Activity(model, 'ClientActivity', Exp.fitMean(1.0)).on(T1);
A1.bound_to(E1).synch_call(E2, 2.5); % 2.5 DB calls on average

% DB activity: processes database request
A2 = Activity(model, 'DBActivity', Exp.fitMean(0.8)).on(T2);
A2.bound_to(E2).replies_to(E2);
```

Now we solve the layered network using the LN solver with MVA applied to each layer:

```
solver = LN(model, @(m) MVA(m));
AvgTable = solver.avgTable();
AvgTable
```

The output shows the performance metrics for each node in the layered network:

Node	NodeType	QLen	Util	RespT	ResidT	ArvR	Tput
ClientProcessor	Processor	NaN	0.49913	NaN	NaN	NaN	NaN
DBProcessor	Processor	NaN	0.99823	NaN	NaN	NaN	NaN
ClientTask	RefTask	7.5	0.49913	NaN	1.7254	NaN	0.49913
DBTask	Task	6.6433	0.99823	NaN	5.324	NaN	1.2478
ClientEntry	Entry	7.5	NaN	15.035	NaN	NaN	0.49913
DBEntry	Entry	6.6433	NaN	5.324	NaN	NaN	1.2478
ClientActivity	Activity	7.5	0.49913	15.035	1.7254	NaN	0.49913
DBActivity	Activity	6.6433	0.99823	5.324	5.324	NaN	1.2478

The output shows the performance metrics for each node in the layered network:

Chapter 3

Network models

Throughout this chapter, we discuss the specification of `Network` models, which are extended queueing networks. 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.

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 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. In open networks with probabilistic routing, the arrival rate at a downstream station depends on the throughput of upstream stations weighted by the routing probabilities. State-dependent routing strategies such as join-the-shortest-queue or round-robin introduce additional complexity, as the effective arrival rate at each destination varies dynamically based on system state. The arrival rate metric is particularly useful for validating Little's Law, which relates queue length, arrival rate, and response time, and for analyzing bottleneck behavior where high arrival rates relative to service capacity lead to queue buildup and increased delays.

The above metrics refer to the performance characteristics of individual nodes. Response times and throughputs can also be system-wide, meaning that they can describe end-to-end performance during the visit to the network. In this case, these metrics are called *system* metrics. System-level metrics include `SysRespT` (system response time), `SysTput` (system throughput), and `SysQLen` (system queue length). These system metrics aggregate information across all stations to provide a holistic view of network performance from the perspective of jobs as they complete their journey through the system. The system response time measures the total time a job spends in the network from entering at the source (for open classes) or leaving the reference station (for closed classes) until it exits at the sink or returns to complete at the reference station. The system throughput captures the rate at which jobs complete their end-to-end execution, accounting for all routing, queueing, and service delays encountered along their path. The system queue length represents the total number of jobs present anywhere in the network, summed across all stations and classes. These metrics can be retrieved using the `avgSys` method, which returns system-level averages organized by chain rather than by individual station and class pairs. The corresponding `getAvgSys` function provides access to the same metrics in array format for programmatic manipulation. System-level metrics are particularly valuable when comparing alternative network configurations or evaluating the impact of parameter changes on overall system behavior, as they abstract away the details of individual station performance and focus on the end-to-end user experience.

3.1   Network object definition

3.1.1   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');
```

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
`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

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-serve 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 `setNumberOfServers` 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.

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 `setService` function or using the `setStrategyParam` function, see later the description of DPS scheduling for an example.

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

Table 3.2: Scheduling strategies available in LINE

Strategy	Code	Description
First-come first-serve	FCFS	Jobs are served in order of arrival
Infinite-server	INF	All jobs receive service simultaneously (delay station)
Processor-sharing	PS	Server capacity is shared equally among all present jobs
Service in random order	SIRO	Jobs are selected randomly for service
Discriminatory processor-sharing	DPS	Processor sharing with class-dependent weights
Generalized processor-sharing	GPS	Processor sharing with configurable weights per class
Shortest expected processing time	SEPT	Job with shortest expected processing time is served first
Shortest job first	SJF	Job with shortest service requirement is served first
Polling	POLLING	Server cycles through queues in round-robin fashion
Last-come first-serve	LCFS	Most recently arrived job is served first
Least attained service	LAS	Job with least accumulated service time gets priority
Earliest due date	EDD	Job with earliest deadline is served first
Limited processor sharing	LPS	Processor sharing with limit on concurrent jobs
Feedback scheduling	FB	Multi-level feedback scheduling
Longest expected processing time	LEPT	Job with longest expected processing time is served first

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 `addServerType` 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);
fastType = ServerType('FastServer', 2);
slowType = ServerType('SlowServer', 2);
queue.addServerType(fastType);
queue.addServerType(slowType);
queue.setService(jobclass, Exp(1.0), fastType);
queue.setService(jobclass, Exp(0.5), slowType);
```

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 3.3. The policy is set via `queue.setHeteroSchedPolicy(Het`

Table 3.3: 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 Longest Free Server: similar to ALIS but with a fairness-based sorting that accounts for server coverage across job classes
FAIRNESS	Distributes jobs fairly across all compatible server types, balancing load proportional to the number of servers of each type
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

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 to 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 the load-dependent service section under Advanced node parameters). 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).

Customer impatience and reneging. Queue nodes support customer impatience, where jobs may abandon the queue if their waiting time becomes excessive. This feature models real-world scenarios such as call centers where customers hang up after waiting too long, web users navigating away from slow-loading pages, or service systems where customers leave due to long queues. The `setPatience` method on Queue nodes specifies a patience distribution for a given job class. When a job arrives or begins waiting, a patience time is sampled from this distribution. If the job's waiting time exceeds this sampled patience time before service begins, the job abandons the queue and exits the system. The patience distribution can be any standard distribution supported by LINE, such as exponential for memoryless impatience or deterministic for fixed deadlines. The following example demonstrates a queue where jobs of a certain class have exponentially distributed patience with mean 10 time units.

```
queue = Queue(model, 'ServiceQueue', SchedStrategy.FCFS);
queue.setPatience(jobclass, Exp(0.1)); % mean patience = 10
```

This feature is primarily supported by the JMT solver through discrete event simulation, which can accurately track individual job patience times and trigger abandonment events. Customer impatience is particularly relevant for performance analysis of systems with quality-of-service constraints, where understanding abandonment rates and their impact on effective throughput is critical for capacity planning and service level agreement compliance.

Fork and Join nodes. The fork and join nodes are currently available only for the JMT solver. 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 `setTasksPerLink` 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).

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 `setProbRouting` 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, `SolverMAM` automatically detects valid FJ topologies (Source $\rightarrow$ Fork $\rightarrow$ K Queues $\rightarrow$ Join $\rightarrow$ Sink) and applies percentile computation as follows

```
solver = SolverMAM(model);
[percRT, percTable] = solver.getPerctRespT([90, 95, 99]);
```

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]);
```

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 [33] 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 3.1.5.

Beyond basic capacity constraints, cache nodes in LINE support weight-based occupancy policies where each cached item can consume a configurable amount of storage space rather than occupying a single slot. This is useful for modeling caches where objects have heterogeneous sizes, such as web caches storing documents of varying lengths. The FCRWeight and FCRMemOcc mechanisms enable specification of per-item weights counted against the total cache capacity. 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.

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 Section 2.2.6. Routing strategies need to be specified for each class using the `setRouting` method and valid choices are as follows

- Random routing (`RoutingStrategy.RAND`)

- Round robin (`RoutingStrategy.RROBIN`)

- Probabilistic routing (`RoutingStrategy.PROB`)

- Join-the-shortest-queue (`RoutingStrategy.JSQ`)

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.setRouting(oclass, RoutingStrategy.RAND);
```

It should be noted that `setRouting` 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.

Logger node.   A logger node is a node that closely resembles the logger node available in the JSIMgraph simulator within JMT. At present, models that include this element can only be solved using the JMT solver.

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 Logger node provides fine-grained control over what information is captured for each job passage through configuration properties. Event logging can be customized to include the simulation start time, the logger node name, the current timestamp when the job passes through, the unique job identifier, the job class, the time since the last job of the same class passed through, and the time since any job of any class last passed. These boolean configuration options allow selective recording based on analysis requirements, reducing log file size when only specific metrics are needed. The CSV output format writes one row per job passage with columns corresponding to the enabled logging options. Departure timestamp recording captures the exact simulation time when jobs leave the logger, enabling post-analysis of inter-arrival time distributions, service time variability, and queueing delays at different points in the network. The LogTunnel section that implements the logging behavior operates transparently without introducing queueing delay, ensuring that logger placement does not alter the performance characteristics being measured. Jobs pass through the logger instantaneously while their passage event is recorded, making loggers suitable for placement at any point in the network topology without affecting throughput or response time metrics. When multiple loggers are placed at different network locations, their combined output provides a detailed trace of job trajectories through the system, supporting visualization of flow patterns and identification of bottlenecks. The logger output can be post-processed to compute empirical distributions, perform statistical hypothesis tests comparing different model configurations, or validate simulation results against analytical solver predictions.

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

- `setStartTime`: record a timestamp for the wallclock time when the simulation started.

- `setJobID`: record a unique id for the passing job.

- `setJobClass`: record the class of the passing job.

- `setTimestamp`: record a timestamp for the simulated time when the job passed in the logger.

- `setTimeSameClass`: record the time elapsed since the last passage of a job of the same class.

- `setTimeAnyClass`: record the time elapsed since the last passage of a job of any class.

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.setJobClass(true);
```

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

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');
fastMode = trans.addMode('fast');  % returns Mode object
slowMode = trans.addMode('slow');
trans.setDistribution(fastMode, Exp(2.0));  % fast processing mode
trans.setDistribution(slowMode, Exp(0.5));  % slow processing mode
trans.setEnablingConditions(fastMode, jobclass, place, 1);  % requires 1 token
trans.setEnablingConditions(slowMode, jobclass, place, 1);
```

Examples are provided in `spn_basic_open`, `spn_twomodes`, and related files in the `examples/` folder.

3.1.2   Advanced node parameters

Scheduling parameters

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

```
queue.setService(class2, Cox2.fitMeanAndSCV(0.2,10), 5.0);
```

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

Finite buffers

The functions `setCapacity` and `setChainCapacity` 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. LINE does not allow one to specify buffer constraints at the level of individual classes unless chains contain a single class, in which case `setChainCapacity` is sufficient for the purpose.

For example,

```
cqn_threeclass_hyperl
delay.setChainCapacity([1,1])
model.refreshCapacity()
```

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 `setChainCapacity([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 `refreshCapacity` 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 `refreshStruct` 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 `refreshCapacity` call, since the internal representation of the `Network` object used by the solvers is still to be created.

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.setPollingType(PollingType.EXHAUSTIVE)
```

or for K-Limited with $K = 2$

```
queue.setPollingType(PollingType.KLIMITED, 2)
```

3.1.3   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 Source. 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');
% Block 1: nodes
delay = Delay(model, 'Delay');
queue = Queue(model, 'Queue1', SchedStrategy.FCFS);
```

```
% Block 2: classes
cclass = ClosedClass(model, 'Class1', 10, delay, 0);
slclass = SelfLoopingClass(model, 'SLC', 1, queue, 0);
delay.setService(cclass, Exp(1.0));
queue.setService(cclass, Exp(1.5));
queue.setService(slclass, Exp(1.5));
% Block 3: topology
P = model.initRoutingMatrix;
P{cclass} = [0.7,0.3;1.0,0];
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.

Table 3.4: Priority scheduling strategies available in LINE

Strategy	Code	Description
FCFS with priority	`FCFSPRIO`	FCFS within priority classes, non-preemptive
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

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 3.4). 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.

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 3.1.5.

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 within 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

Table 3.5: 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

such as the residence time. To cope with this issue, LINE offers a method for the class objects, called `setReferenceClass`, 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.

Networks with signals

Networks with signals are supported by the `SolverDES` (discrete event simulation) and `SolverMAM` (matrix-analytic methods) solvers. Table 3.5 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

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

```
negSignal = Signal(model, 'NegativeCustomer', SignalType.NEGATIVE);
negSignal.setRemovalDistribution(Geometric(0.5)); % batch removal
negSignal.setRemovalPolicy(RemovalPolicy.LCFS);   % remove newest first
source.setArrival(negSignal, Exp(0.1));
```

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

```
disaster = Signal(model, 'Disaster', SignalType.CATASTROPHE);
source.setArrival(disaster, Exp(0.01)); % rare catastrophic events
```

3.1.4 Routing strategies

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.initRoutingMatrix;
P{class1}(source,queue) = 1.0;
P{class1}(queue,[queue,delay]) = [0.3,0.7]; % self-loop with probability 0.3
P{class1}(delay,sink) = 1.0;
P{class2}(delay,queue) = 1.0; % note: closed class jobs start at delay
P{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.initRoutingMatrix;
P{class1} = [0,1,0,0;   % row: source
             0,.3,.7,0; % row: queue
             0,0,0,1;   % row: delay
             0,0,0,0];  % row: sink
P{class2} = [0,0,0,0;
             0,0,1,0;
             0,1,0,0;
             0,0,0,0];
model.link(P);
```

Where needed, the `getClassIndex` and `getNodeIndex` functions return the numerical index associated with a node name, for example `model.getNodeIndex('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 `getClassNames`, `getStationNames`, and `getNodeNames` 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 `printRoutingMatrix` function, e.g.,

```
>> model.printRoutingMatrix
Delay [Class1] => Queue1 [Class1] : Pr=1.000000
Delay [Class2] => Queue1 [Class2] : Pr=0.001000
Queue1 [Class1] => Queue1 [Class1] : Pr=0.300000
Queue1 [Class1] => Source [Class1] : Pr=0.700000
Queue1 [Class2] => Source [Class2] : Pr=1.000000
Source [Class1] => Sink [Class1] : Pr=1.000000
Source [Class2] => Queue1 [Class2] : Pr=1.000000
Sink [Class2] => Source [Class2] : Pr=1.000000
```

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 with configurable weights per destination.

- Power of k choices (`RoutingStrategy.KCHOICES`). Samples k candidate destinations uniformly at random (without replacement) from the set of available outgoing links, then routes the job to the one with the shortest queue length. The parameter k defaults to 2 if not specified. An optional Boolean parameter `memory` (default: `false`) enables memory-aware sampling. To set a specific value of k use `setRouting(class, RoutingStrategy.KCHOICES, k)`, where `k` is the integer number of candidates sampled per routing decision.

- 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.

- Reinforcement learning (`RoutingStrategy.RL`). This is a state-dependent routing strategy that uses a learned value function to make routing decisions based on current queue lengths. The value function can be either a tabular lookup table (for small state spaces) or a linear/quadratic function approximation (for larger state spaces). When the current state exceeds the learned state space, the strategy falls back to JSQ (Join-the-Shortest-Queue) routing. The RL routing policy is trained offline using temporal difference (TD) learning via the `rl_env` and `rl_td_agent` API classes, then applied to the network model. To use RL routing:

```
node.setRouting(jobClass, RoutingStrategy.RL, valueFunction, {actionNodes, ...
    stateSize});
```

where `valueFunction` is the learned value function (tabular array or regression coefficients), `actionNodes` is a vector of node indices where RL routing decisions are needed, and `stateSize` controls the state truncation (`0` for tabular, `>0` for function approximation). This strategy currently supports single-class models and is evaluated by solvers that compute state-dependent routing, such as CTMC.

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

```
model.addLink(queue, queue); %self-loop
model.addLink(queue, delay);
```

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

```
queue.setRouting(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.setRouting(class1,RoutingStrategy.PROB);
queue.setProbRouting(class1, queue, 0.7)
queue.setProbRouting(class1, delay, 0.3)
```

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

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, the getting started Example 1 on the M/M/1 queue illustrates a simplified way to specify a serial routing topology, i.e.,

```
model.link(Network.serialRouting(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

```
lambda = [10,20]; % lambda(r) – arrival rate of class r
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.tandemPsInf(lambda,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.tandemPs`, `Network.tandemFcfs`, and `Network.tandemFcfsInf` 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.cyclicPs`: cyclic network of PS queues

- `Network.cyclicPsInf`: cyclic network of PS queues and delay stations

- `Network.cyclicFcfs`: cyclic network of FCFS queues

- `Network.cyclicFcfsInf`: 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.,

```
N = [10,20]; % N(r) – closed population in class r
D = [11,12; 21,22]; % D(i,r) – class-r demand at station i (PS)
modelPsInf = Network.cyclicPs(N,D)
```

3.1.5   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.getChains()` 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:

```
class3 = ClosedClass(model, 'Class3', 0, queue, 0);

P = model.initRoutingMatrix;
P{class1,class1}(source, queue) = 1.0;
P{class1,class1}(queue, [queue,delay]) = [0.3,0.7];
P{class1,class1}(delay, sink) = 1.0;
P{class2,class3}(delay, queue) = 1.0;
P{class3,class2}(queue, delay) = 1.0;
model.link(P);
```

where `P{r,s}` is the routing matrix for jobs switching from class `r` to `s`. That is, `P{r,s}(i,j)` is the probability that a job in class `r` departs node `i` routing into node `j` as a job of class `s`.

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.

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{1} = OpenClass(model, 'Class1', 0);
jobclass{2} = OpenClass(model, 'Class2', 0);
...
%% Block 3: topology
C = csnode.initClassSwitchMatrix;
C(jobclass{1},jobclass{2}) = 1.0;
C(jobclass{2},jobclass{2}) = 1.0;
```

```
    csnode.setClassSwitchingMatrix(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.

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.getClassSwitchingMask` 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 `setHitClass` and `setMissClass`, so that it can be routed to a different destination based on whether it found the item in the cache or not. The `setRead` 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.setRead(initClass, refModel);
    cacheNode.setHitClass(initClass, hitClass);
    cacheNode.setMissClass(initClass, missClass);
```

Here `initClass`, `hitClass`, and `missClass` can be either open or closed instantiated as usual with the `OpenClass` or `ClosedClass` constructors.

3.1.6 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 Source 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 3.6.

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.setService(class2, Cox2.fitMeanAndSCV(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 `setArrival` instead of `setService` 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.setArrival(class2, Exp.fitMean(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 3.7.

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 3.8 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 `NaN`.

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 ($\varepsilon =$`GlobalConstants.FineTol`).

Fitting a distribution

The `fitMeanAndSCV` 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.fitMeanAndSCV(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 `fitMeanAndOrder`(μ, n), which instantiates a n-phase Erlang with given mean μ.

In distributions that are uniquely determined by more than two moments, `fitMeanAndSCV` 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.fitMeanAndSCV` 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.fitMeanAndSCVBalanced` instead assigns p in a two-phase hyper-exponential distribution so that $p/\mu_1 = (1 - p)/\mu_2$.

Inspecting and sampling a distribution

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

```
>> dist = Exp(1);
>> dist.getMean
ans =
     1
>> dist.getSCV
ans =
     1
```

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

```
>> dist.sample(3)
ans =
    0.2049
    0.0989
    2.0637
```

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

```
>> dist.evalCDFInterval(2,5)
ans =
    0.1286
>> dist.evalCDF(5)-dist.evalCDF(2)
ans =
    0.1286
```

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 `getRepresentation` function [8]. The result will be a cell array where element $k + 1$ corresponds to matrix D_k.

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* [16], 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 `setLoadDependence` 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.setLoadDependence(min(1:N,c)); % multi-server with c servers
```

where the i-th element of the vector argument of `setLoadDependence` is the scaling factor $\alpha(n_i)$. It is assumed by default that $\alpha(0) = 1$.

Class-dependent service

A generalization of load-dependent service is *class-dependent* service, where the service rate is now a function of the vector $n_i = [n_{i,1}, \ldots, n_{i,R}]$, where $n_{i,r}$ is the current number of class-r jobs at station i.

LINE supports class-dependence in the MVA solver, provided that this is specified as a function handle. The solver implicitly assumes that the function is smooth and defined also for fractional values of $n_{i,r}$. For example, in a two-class model we may write

```
    queue.setClassDependence(@(ni) min(ni(2),c));
```

applies a multiserver-type only to class-2 jobs, but not to the others.

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.setSwitchover(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 $i + 1$ mod 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.setSwitchover(jobclass1, Exp(1));
```

Impatience (customer abandonment)

LINE supports customer impatience through configurable patience distributions. When a job arrives at a queue, it draws a patience time from the specified distribution. If the job has not entered service before its patience expires, it reneges (abandons) the queue and is routed to the next station or sink. Impatience can be configured at queue-level or class-level:

```
queue.setPatience(jobclass, Exp(0.5))  % Mean patience = 2.0
jobclass.setPatience(Det(5.0))  % Fixed timeout = 5.0 everywhere
```

When both global and queue-specific patience are configured, the queue-specific setting takes precedence. Impatience supports all standard distributions except modulated processes (BMAP, MAP, MMPP2).

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.

3.2 Internals

In this section, we discuss the internal representation of the `Network` objects used within the LINE solvers. By applying changes directly to this internal representation it is possible to considerably speed up the sequential evaluation of models.

3.2.1 Representation of the model structure

For efficiency reasons, once a user requests to solve a `Network`, LINE calls internally generate a static representation of the network structure using the `refreshStruct` function. This function returns a representation object that is then passed on to the chosen solver to parameterize the analysis.

The representation used within LINE is the `NetworkStruct` class, which describes an extended multi-class queueing network with class-switching and acyclic phase-type (APH) service times. APH generalizes known distributions such as Coxian, Erlang, Hyper-Exponential, and Exponential. The representation can be obtained as follows

```
sn = model.getStruct()
```

The next tables present the properties of the NetworkStruct class. Table 3.9 gives the invariant properties of the class, which specify the initial parameterization of the nodes, which we refer to as static parameters. Table 3.2.1 further details parameters that require algorithmic calculations to derive from the static parameters.

For advanced nodes, such as Cache and Transition, additional parameters are specified under the `nodeparam` cell array for the corresponding node. Table 3.2.1 illustrates the properties specified within the `nodeparam{i}` cell array for Transition node i. Table 3.2.1 similarly illustrates the properties in `nodeparam{i}` array for a Cache node i.

As shown in the tables, internally to LINE there is an explicit differentiation between properties of nodes, stations, and stateful nodes. This distinction has impact in particular over routing and class-switching mechanisms, and also allows solvers to better differentiate between different kinds of nodes.

In some cases, one may want to access some properties of nodes that are contained in `NetworkStruct` fields that are however referenced by station or stateful node index. To help this and similar situations, the `NetworkStruct` class also provides static methods to quickly convert the indexing of nodes, stations, and stateful nodes, which is used in referencing its data structures:

- `nodeToStateful`

- `nodeToStation`

- `stationToNode`

- `stationToStateful`

- `statefulToNode`

As an example, we can determine the portion of the `nodevisits` field that refers to stateful nodes in chain $c = 1$ as follows

```
c = 1;
V = zeros(sn.nstateful,1);
sn = model.getStruct(); % NetworkStruct object
for ind=1:sn.nnodes
    if sn.isstateful(ind)
        isf = sn.nodeToStateful(ind);
        V(isf,1) = sn.nodevisits{c}(ind,2);
    end
end
```

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.jsimgView
```
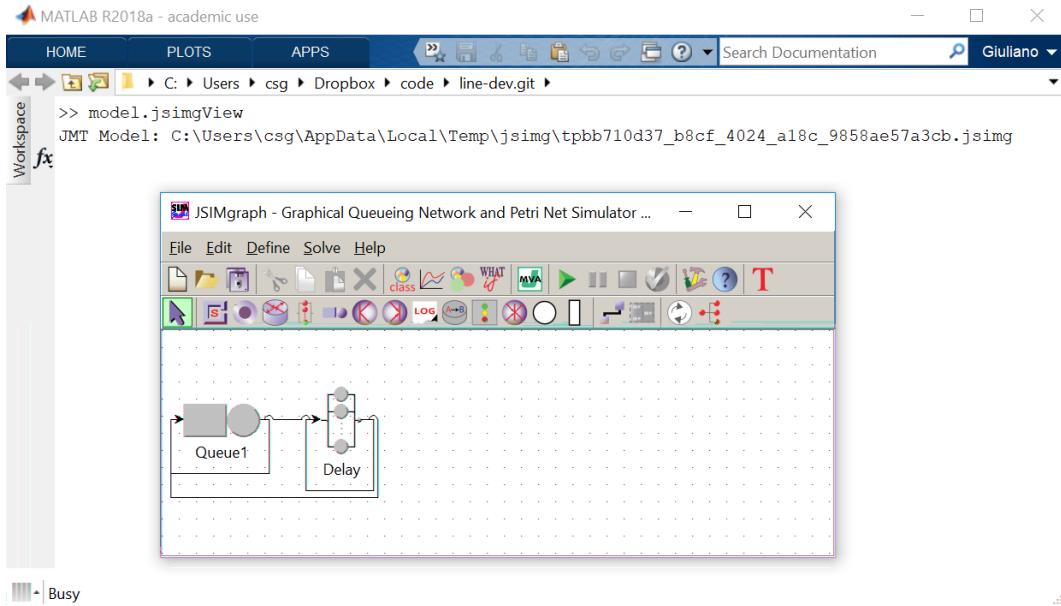
Figure 3.1: `jsimgView` function

An example is shown in Figure 3.1. Using a related function, `jsimwView`, 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, e.g.

```
G = model.getGraph();
plot(G,'EdgeLabel',G.Edges.Weight,'Layout','Layered')
```

plots a graph of the network topology in term of stations only. In a similar manner, the following variant of the same command shows the model in terms of nodes, which corresponds to the internal representation within LINE.

```
[~,H] = model.getGraph();
plot(H,'EdgeLabel',H.Edges.Weight,'Layout','Layered')
```

The latter example is invoked by default if we type

```
plot(model)
```

which also adds automatic node coloring to highlight the class-switch nodes.

Figure 3.2 shows the difference between the two commands for an open queueing network with two classes and class-switching. Weights on the edges correspond to routing probabilities. In the station topology on the left, note that since the Sink node is not a station, departures to the Sink are drawn as returns to
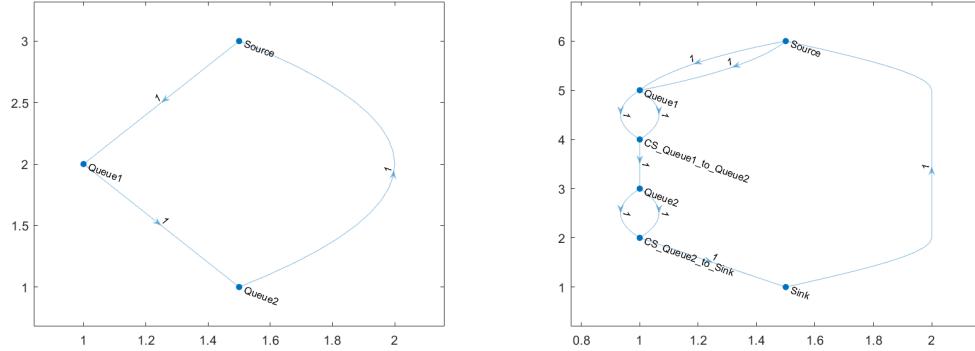
Figure 3.2: `getGraph` function: station topology (left) and node topology (right) for a 2-class tandem queueing network with class-switching.

the Source. The node topology on the right, illustrates all nodes, including `ClassSwitch` nodes that are automatically added by LINE to apply the class-switching routing strategy. Double arcs between nodes indicate that both classes are routed to the destination.

Furthermore, the graph properties concisely summarize the key features of the network

```
>> G.Nodes
ans =
  2x5 table
    Name            Type            Sched   Jobs    ClosedClass1
    _____          _____          _____  ____    _____

    'Delay'     'Delay'             'inf'    5           1
    'Queue1'    'Queue'             'ps'     0           2
>> G.Edges
ans =
  3x4 table
        EndNodes            Weight    Rate        Class
    _____        _____    ____    _____

    'Delay'     'Delay'      0.7       1      'ClosedClass1'
    'Delay'     'Queue1'     0.3       1      'ClosedClass1'
    'Queue1'    'Delay'       1        0.5    'ClosedClass1'
```

Here, `Edge.Weight` is the routing probability between the nodes, whereas `Edge.Rate` is the service rate of the node appearing in the first column under `EndNodes`.   Another way to debug a LINE model is to transform it into a graph object, i.e.,

3.4 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([pwd,'\\examples\\data\\JMT\\jmt_example.jsimg'],'Mod1')
```

where `sn` is an instance of the Network class.

Beyond JMT models, LINE supports importing workflow models specified using the Business Process Model and Notation (BPMN) 2.0 standard. BPMN is a widely adopted graphical notation for describing business processes and workflows, and its conversion to layered queueing networks enables performance analysis of process execution scenarios. The BPMN import capability transforms workflow elements into corresponding LayeredNetwork components that capture the essential performance characteristics of the process. In the conversion process, BPMN tasks are mapped to entries and activities in layered queueing networks, while BPMN resources become processors with associated scheduling policies. Parallel gateways in BPMN workflows are represented using AND-fork and AND-join precedence constructs within activity graphs, allowing the analysis of concurrent execution paths and their synchronization points. The multiplicity and replication settings of tasks and processors are derived from the resource allocation specified in the BPMN model. Service time distributions for activities are extracted from annotations or default exponential distributions are assigned when timing information is not explicitly provided. The reference to the MATLAB example file `lqn_bpmn.m` in the examples directory demonstrates a complete workflow that has been manually converted from BPMN specification to a LayeredNetwork representation, illustrating the correspondence between workflow constructs and layered queueing network elements. This example shows how processors, tasks, entries, and activities are structured to preserve the precedence relationships and resource consumption patterns of the original business process. While automated import from BPMN XML files is supported through parsing utilities, manual model construction following the patterns in `lqn_bpmn.m` provides greater control over the mapping of workflow semantics to performance model constructs. The ability to import and analyze BPMN models extends the applicability of LINE to enterprise application performance modeling, business process optimization, and service-oriented architecture analysis.

Network object can be saved in binary `.mat` files using MATLAB's standard `save` command. However, it is also possible to export a textual script that will dynamically recreate the same Network object. For example,

```
cqn_repairmen; LINE2MATLAB(model, 'script.m')
```

creates a new file `script.m` with code

```
model = Network('model');
```

```matlab
%% Block 1: nodes
node{1} = DelayStation(model, 'Delay');
node{2} = Queue(model, 'Queue1', SchedStrategy.FCFS);

%% Block 2: classes
jobclass{1} = ClosedClass(model, 'Class1', 10, node{1}, 0);

node{1}.setService(jobclass{1}, Exp.fitMean(1.000000)); % (Delay,Class1)
node{2}.setService(jobclass{1}, Exp.fitMean(1.500000)); % (Queue1,Class1)

%% Block 3: topology
P = model.initRoutingMatrix(); % initialize routing matrix
P{1,1}(1,1) = 7.000000e-01; % (Delay,Class1) -> (Delay,Class1)
P{1,1}(1,2) = 3.000000e-01; % (Delay,Class1) -> (Queue1,Class1)
P{1,1}(2,1) = 1; % (Queue1,Class1) -> (Delay,Class1)
model.link(P);
```

that is equivalent to the model specified in cqn_repairmen.m.

3.4.1   Creating a LINE model using JMT

Using the features presented in the previous section, one can create a model in JMT and automatically derive a corresponding LINE script from it. For instance, the following command performs the import and translation into a script, e.g.,

```matlab
LINE2MATLAB(JMT2LINE('myModel.jsimg'),'myModel.m')
```

transforms and save the given JSIMgraph model into a corresponding LINE model.

LINE also provides two static functions to inspect jsimg and jsimw files before conversion, called JMT.jsimgOpen and JMT.jsimwOpen require as an input parameter only the JMT file name, e.g., myModel.jsimg.

It is also possible to automate the editing and import of JMT models from MATLAB using the jsimgEdit command. This will open an empty JMT model and upon saving this will be automatically reimported into MATLAB.

3.4.2   Supported JMT features

Table 3.4.2 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.

Table 3.13: Supported JSIM features for automated model import and analysis

JMT Feature	Support	Notes
Distributions	Full	Phase-Type, Burst (MAP), Burst (MMPP2), Matrix Exponential (ME), Rational Arrival Process (RAP), Deterministic, Disabled, Exponential, Erlang, Gamma, Hyperexponential, Coxian, Logistic, Pareto, Uniform, Zero Service Time, Replayer, 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	Finite Capacity Region, ClassSwitch, Place, Delay, Logger, Queue, Router, Transition
Routing	Full	Random, Probabilities, Round Robin, 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
Nodes	Partial	Fork, Join, Source, Sink
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

3.5   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 `addRegion` 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 `setGlobalMaxJobs`, which specifies the total number of jobs allowed in the region. The drop behavior for each class is configured using `setDropRule`, which determines whether arriving jobs are dropped when the region is at capacity.

```
fcr = model.addRegion({queue1, queue2});
fcr.setGlobalMaxJobs(K);
fcr.setDropRule(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.WaitingQueue` 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.

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.

When using the JMT solver, two additional performance metrics can be requested for each FCR. The `FCRWeight` metric (`MetricType.FCRWeight`) reports the total weight of jobs currently residing in the region, which is useful when jobs are assigned heterogeneous weights reflecting their resource consumption (e.g., memory footprint or storage size). The `FCRMemOcc` metric (`MetricType.FCRMemOcc`) reports the total memory occupation of the region, aggregating the memory contributions of all jobs present across the stations in the region. These metrics are particularly relevant when modeling cache-like systems or storage pools where different job classes consume different amounts of shared resources.

3.6   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 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 `setReward` 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.setReward('QueueLength', @(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.setReward('Utilization', Reward.utilization(queue, oclass));
model.setReward('BlockingProb', Reward.blocking(queue));
```

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 `getAvgReward` method returns a vector of expected reward values in equilibrium, along with a cell array of reward names. For transient analysis, the `getReward` method returns time-indexed reward trajectories, along with the time points, reward names, and the state space representation.

```
solver = CTMC(model, options);
[R, names] = solver.getAvgReward();    % steady-state expected rewards
[t, V, names, stateSpace] = solver.getReward();   % transient trajectories
```

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_basic`, `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.

3.7 Stochastic Petri nets

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.

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.addMode('Mode1');
T1.setNumberOfServers(mode, Inf);
T1.setDistribution(mode, Exp(4));
T1.setEnablingConditions(mode, jobclass1, P1, 1);   % requires 1 token in P1
T1.setFiringOutcome(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 `setInhibitingConditions` method specifies that a transition cannot fire when a designated place holds more than a specified threshold number of tokens.

```
T1.setInhibitingConditions(mode, jobclass1, P2, 2);   % inhibit if P2 has >2 tokens
```

SPN models in LINE are analyzed by the JMT solver, which translates the Petri net specification into its native simulation engine and performs discrete-event simulation to estimate performance metrics. 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.

3.8   Signals and G-networks

LINE supports the modeling of G-networks, also known as Gelenbe networks, through the `Signal` class. G-networks 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.

```
negClass = Signal(model, 'Negative', SignalType.NEGATIVE);
source.setArrival(negClass, 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.

G-network models can be analyzed using the `MAM` solver with the INAP or INAPPLUS methods, which exploit the product-form structure of certain G-network configurations through the RCAT (Reversed Compound Agent Theorem) framework. Under specific conditions on the arrival and service processes, G-networks admit product-form solutions that allow efficient exact analysis without state-space explosion. This capability makes it possible to analyze large-scale G-networks with multiple positive and negative customer classes while maintaining tractability.

```
options = SolverOptions();
options.method = 'inap';
solver = SolverMAM(model, options);
[QN, UN, RN, TN] = solver.getAvg();
```

The example `ag_gnetwork` demonstrates the construction and analysis of a G-network model with both positive customers and negative arrivals, illustrating how signal classes interact with ordinary job classes to produce non-trivial equilibrium behavior. G-networks 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.

Table 3.6: Phase-type distributions available in LINE

Distribution	Description
Exponential	$\text{Exp}(\lambda)$, where λ is the rate of the exponential
n-phase Erlang	$\text{Erlang}(\alpha, n)$, where α is the rate of each of the n exponential phases
2-phase hyper-exponential	$\text{HyperExp}(p, \lambda_1, \lambda_2)$, that returns an exponential with rate λ_1 with probability p, and an exponential with rate λ_2 otherwise
n-phase hyper-exponential	$\text{HyperExp}(p, \lambda)$, 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	$\text{Coxian}(\mu_1, \mu_2, \phi_1)$, 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	$\text{Coxian}(\mu, \phi)$, 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	$\text{APH}(\alpha, T)$, 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	$\text{ME}(\alpha, A)$, 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 [6]
Rational arrival process	$\text{RAP}(H_0, H_1)$, 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) [2]
Markovian arrival process	$\text{MAP}(D_0, D_1)$, which defines a point process with phase-dependent arrival rates via matrices D_0 (hidden transitions) and D_1 (arrivals)
Markov-modulated Poisson process	$\text{MMPP2}(\lambda_1, \lambda_2, \sigma_1, \sigma_2)$, a two-phase MMPP with arrival rates λ_i and switching rates σ_i
Batch Markovian arrival process	$\text{BMAP}(D)$, which generalizes MAP to allow batch arrivals via a set of arrival matrices $D = \{D_0, D_1, \ldots, D_k\}$
General phase-type	$\text{PH}(\alpha, T)$, which defines a phase-type distribution with initial probability vector α and transient generator T, allowing cyclic phase transitions
Markov-modulated deterministic process	$\text{MMDP}(Q, R)$, 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 (SolverFLD with method mfq)
2-state Markov-modulated deterministic process	$\text{MMDP2}(r_0, r_1, \sigma_0, \sigma_1)$, 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	$\text{MarkedMAP}(D, K)$, 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	$\text{MarkedMMPP}(D, K)$, 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

Table 3.7: 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

Table 3.8: 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}

Table 3.9: NetworkStruct static properties

Field	Type	Description
cdscaling(n_{ir})	cell	class-dependent scaling when station i contains n_{ir} in class r
classdeadline(r)	double	deadline for class r (Inf = no deadline)
classnames{r}	char	Name of class r
classprio(r)	integer	Priority of class r (0 = highest priority)
connmatrix(i,j)	logical	true if node i can route jobs to node j
droprule(i,r)	integer	drop rule for class r at station i
fj(f,j)	logical	true if forked tasks from fork node f join at node j
gsync{s}	struct	Data structure specifying a global synchronization s among nodes
heterorates{i}{t,r}	double	Service rate for server type t and class r at station i (heterogeneous servers)
heteroproc{i}{t,r}	cell	Service process matrices for server type t and class r at station i
heteroprocid(i,t,r)	integer	Service process type id for server type t and class r at station i
heteroschedpolicy(i)	integer	Heterogeneous scheduling policy id at station i
immfeed(i,r)	logical	true if class r uses immediate feedback at station i
inchain(c)	integer	indexes of classes in chain c
isCatastrophe(r)	logical	true if class r is a catastrophe signal class
isslc(r)	logical	true if class r is a self-looping class
issignal(r)	logical	true if class r is a signal class
isstatedep(i,s)	logical	true if node i has state-dependent section ($s = 1$: input, $s = 2$: service, $s = 3$: routing)
isstation(i)	logical	true if node i is a station
isstateful(i)	logical	true if node i is a stateful node
ljdcutoffs(i,r)	double	Per-class cutoffs for joint dependence at station i and class r
ljdscaling{i}	cell	limited joint-dependent scaling functions (linearized per station)
lldscaling(i,n_i)	double	load-dependent scaling when station i contains n_i jobs, including the ones in service
lst{i,r}	function handle	Laplace-Stieltjes transform of the service or arrival distribution for class r at station i
mu{i,r}(k)	double	Service or arrival rate in phase k for class r at station i, with mu{i,r} =NaN if Disabled and mu{i,r} = 10^7 if Immediate.
nclasses	integer	Number of classes in the network
nclosedjobs	integer	Total number of jobs in closed classes
njobs(r)	integer	Number of jobs in class r (Inf for open classes)
nnodes	integer	Number of nodes in the network
nregions	integer	Number of finite capacity regions
nservers(i)	integer	Number of servers at station i
nservertypes(i)	integer	Number of server types at station i (0 = homogeneous)
nstations	integer	Number of stations in the network
nstateful	integer	Number of stateful nodes in the network
nodenames{i}	string	Name of node i
nodeparam{i}	double	Parameters for local variable instantiation at stateful node i
nodetype{i}	integer	Type of node i (e.g., NodeType.Sink)
nvars(i,r)	integer	Number of local state variables for stateful node i. Position r describes state variables for class-r service; position r =nclasses+s for class-s routing; positions after 2·nclasses are used for class-independent variables.
phases(i,r)	integer	Number of phases for service process of class r at station i
phasessz(i,r)	integer	Number of state vector elements used to describe phase
phaseshift(i,r)	integer	Position shift to read phase element in state
proc{i,r}	cell	Matrix representation of [1] the class r service process at station i
procid(i,r)	integer	Service or arrival process type id at station i for class r (e.g., ProcessType.HyperExp)
refclass(c)	integer	Index of reference class for class c

Table 3.10: NetworkStruct computed properties

Field	Type	Description
`cap`(i)	`integer`	Total capacity at station i
`chains`(c,r)	`logical`	`true` if class r is in chain c, or `false` otherwise
`classcap`(i,r)	`integer`	Maximum buffer capacity available to class r at station i
`csmask`(r,s)	`logical`	true if class r can switch into class s at some node
`nchains`	`integer`	Number of chains in the network
`nodevisits`{c}(i,r)	`double`	Number of visits that a job in chain c pays to node i in class r
`phi`{i,r}(k)	`double`	Completion probability in phase k for class r at station i
`pie`{i,r}(k)	`double`	Entry probability in phase k for class r at station i
`rates`(i,r)	`double`	Service rate of class r at station i (or arrival rate if i is a Source)
`rt`(idx_{ir}, idx_{js})	`double`	Probability of routing from stateful node i to j, switching class from r to s where, e.g., $idx_{ir} = (i-1) * \text{nclasses} + r$.
`rtnodes`(idx_{ir}, idx_{js})	`double`	Same as `rt`, but i and j are nodes, not necessarily stateful ones.
`rtfun`(**st1**, **st2**)	`function handle`	State-dependent routing table given initial (`st1`) and final (`st2`) state cell arrays. Table entries are defined as in `rt`.
`scv`(i,r)	`double`	Squared coefficient of variation of class r service times at station i (or inter-arrival times if station i is a Source)
`space`{t}	`integer`	The t-th state in the state space (or a portion thereof). This field may be initially empty and updated by the solver during execution.
`state`{i}	`integer`	Current state of stateful node i. This field may be initially empty and updated by the solver during execution.
`visits`{c}(i,r)	`double`	Number of visits that a job in chain c pays to stateful node i in class r

Table 3.11: `nodeparam`{i} fields for a Transition node i

Field	Type	Description
`enabling`{m}(k,r)	`integer`	Enabling condition for mode m at transition node i with respect to class r jobs at linked node k
`firing`{m}(k,r)	`integer`	Firing outputs of class r jobs for mode m at transition node i towards linked node k
`firingprio`(m)	`integer`	Firing priority for mode m at transition node i
`firingphases`(m)	`integer`	Number of phases for firing process of mode m at node i
`firingproc`{m}	`cell`	Matrix representation of the mode m firing process at transition node i
`firingprocid`(m)	`integer`	Firing process type id for mode m (e.g., `ProcessType.HyperExp`)
`fireweight`(m)	`integer`	Firing weight for mode m at transition node i
`inhibiting`{m}(k,r)	`integer`	Inhibiting condition for mode m at transition node i with respect to class r jobs at linked node k
`modenames`{m}	`string`	Name of mode m at transition node i
`nmodes`	`integer`	Number of modes for transition node i
`nmodeservers`{m}	`integer`	Number of servers for mode m
`timing`(m)	`integer`	Firing type at node i for mode m (e.g., `TimingStrategy.IMMEDIATE`)

Table 3.12: `nodeparam`$\{i\}$ fields for a Cache node i

Field	Type	Description
`accost{`r,i`}(`l,h`)`	`double`	Access cost for class r to move item i from list l to list h
`hitclass(`r`)`	`integer`	Class switching for class r under a cache hit
`itemcap(`l`)`	`integer`	Item capacity in list l
`missclass(`r`)`	`integer`	Class switching for class r under a cache miss
`nitems`	`integer`	Number of items in the cache
`pread{`r`}(`i`)`	`double`	Probability for class r of reading item i
`replacestrat`	`integer`	Replacement policy id (e.g., `ReplacementStrategy.LRU`)
`actualhitprob(`r`)`	`double`	Actual hit probabilities (computed)
`actualmissprob(`r`)`	`double`	Actual miss probabilities (computed)

Chapter 4

Analysis methods

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.

Queue length in Finite Capacity Regions.   When a station is part of a Finite Capacity Region (FCR) with `WaitingQueue` 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.

Fork-join specific metrics.   For models containing fork-join structures, LINE provides specialized metrics that capture the end-to-end behavior of parallel execution sections. The `FJQLen` metric represents the queue length measured at the fork-join scope, counting jobs from the moment they are dispatched at the fork until their synchronization is completed at the join. This differs from standard per-station queue length metrics, which measure only the jobs residing at individual stations. Similarly, the `FJRespT` metric captures the response time from fork dispatch to join synchronization, measuring the total time required for all parallel tasks spawned by a fork to complete and synchronize. These metrics are essential for analyzing parallel processing systems, workflow applications, and distributed computing scenarios where understanding the aggregate behavior of forked execution paths is more relevant than individual station performance. Standard metrics measure individual station behavior in isolation, treating each queue or delay node independently. In contrast, fork-join metrics aggregate across all stations between a fork and its corresponding join, providing insight into the overall latency and resource consumption of parallel sections. These specialized metrics are

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: $\mathtt{SysQLen}(c) = \mathtt{SysTput}(c) \cdot \mathtt{SysRespT}(c)$.
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.
Tardiness	Tard	Mean excess of per-visit response time over the soft deadline d_r for class-r jobs at node i, defined as $\mathrm{E}[\max(0, \mathtt{RespT}_{i,r} - d_r)]$. Requires a soft deadline be assigned to the job class.
System Tardiness	SysTard	Mean excess of system response time over the soft deadline for chain c, defined as $\mathrm{E}[\max(0, \mathtt{SysRespT}(c) - d_r)]$ averaged across completing classes in the chain.

reported through the standard `getAvgTable` output when fork-join structures are present in the model. The fork-join metrics are computed by solvers that support fork-join modeling, including the MVA, JMT, and DES solvers.

Tardiness metrics. LINE supports the computation of tardiness metrics for models where job classes are assigned soft deadlines. A soft deadline d_r specifies the maximum acceptable response time for class-r jobs; jobs that complete within the deadline incur no tardiness, while those that exceed the deadline contribute a tardiness equal to the excess. Formally, the per-visit tardiness at node i for class r is $\mathrm{E}[\max(0, \mathtt{RespT}_{i,r} - d_r)]$. Soft deadlines are assigned at class construction time using the optional deadline parameter:

```
deadline = 0.5;  % soft deadline in time units
oclass = OpenClass(model, 'MyClass', 0, deadline);
% equivalently via the class property:
% oclass.deadline = deadline;
```

Once deadlines are set, the `Tard` metric gives the per-station tardiness for each class, and the `SysTard` metric gives the chain-level system tardiness. Both metrics are computed by the JMT solver via simulation and appear in the output of `avgTable`. Jobs without a deadline (default) have tardiness zero.

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.

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.avgQLen()
% UN(i,r): utilization of class r at station i
UN = solver.avgUtil()
% RN(i,r): mean response time of class r at station i (per visit)
RN = solver.avgRespT()
```

```
% TN(i,r): mean throughput of class r at station i
TN = solver.avgTput()
% AN(i,r): mean arrival rate of class r at station i
AN = solver.avgArvR()
% WN(i,r): mean residence time of class r at station i (summed on visits)
WN = solver.avgResidT()
```

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

```
[QN,UN,RN,TN,AN,WN] = solver.avg()
```

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

```
AvgTable = solver.avgTable()
```

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(1,2)
QN(queue,cclass)
QN(model.getStationIndex('MyQueue'),model.getClassIndex('MyClass'))
```

Similar methods are defined to obtain aggregate performance metrics at chain level at each station, namely `avgQLenChain` for queue-lengths, `avgUtilChain` for utilizations, `avgRespTChain` for response times, `avgTputChain` for throughputs, and the `avgChain` method to obtain all the previous metrics.

4.2.2   Station response time distribution

FLD supports the computation of response time distributions for individual classes through the `getCdfRespT` 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.cdfRespT();
plot(FC{2,1}(:,2),FC{2,1}(:,1)); xlabel('t'); ylabel('Pr(RespT<t)');
```

4.2.3 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. The solver automatically detects valid AoI topologies (Source → Queue → Sink) with capacity 1 (bufferless) or 2 (single-buffer) and computes exact AoI distributions using Markovian Fluid Queue analysis. The getAvgAoI method returns the mean and variance of both AoI and Peak AoI, while getCdfAoI returns their cumulative distribution functions:

```
model = Network('AoI_Example');
source = Source(model, 'Source');
queue = Queue(model, 'Queue', SchedStrategy.FCFS);
queue.setNumberOfServers(1);
queue.setCapacity(1);  % bufferless
sink = Sink(model, 'Sink');
jobclass = OpenClass(model, 'Updates');
source.setArrival(jobclass, Exp(1));
queue.setService(jobclass, Erlang(2, 2));
model.link(Network.serialRouting(source, queue, sink));

solver = FLD(model, 'method', 'mfq');
[AoI, PAoI, aoiTable] = solver.getAvgAoI();
[AoI_cdf, PAoI_cdf] = solver.getCdfAoI();
```

The solver supports FCFS, LCFS, and preemptive LCFS scheduling strategies, mapping them to appropriate preemption/replacement policies for the underlying MFQ analysis.

4.2.4 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*, instead than by class. In this way, since a job switching from a class to another remains by definition in the same chain, there is no ambiguity in attributing the system metrics to the chain. The solver functions avgSys and avgSysTable 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 Little's 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 = cell(2); % 2-classes model
P{1,1} = [0,1; 0,0]; % routing within class 1 (no switching)
P{1,2} = [0,0; 1,0]; % routing from class 1 into class 2
P{2,1} = [0,0; 1,0]; % routing within class 2 (no switching)
P{2,2} = [0,1; 0,0]; % routing from class 2 into class 2

model.link(P);
```

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

4.3   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.

4.3.1   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

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 $r = 1, ..., R$.

Table 4.2: State descriptors for Markovian scheduling policies

Sched. strategy	Station state vector	State condition
EXT	$[\text{Inf}, s_{11}, ..., s_{1K_1}, ..., s_{R1}, ..., s_{RK_R}]$	$\sum_k s_{rk} = 1, \forall r$
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

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

- `State.fromMarginal`: enumerates all states that have the same marginal state $[n_1, n_2, ..., n_R]$.

- `State.fromMarginalAndRunning`: restricts the output of `State.fromMarginal` to states with given number of running jobs, irrespectively of the service phase in which they currently run.

- `State.fromMarginalAndStarted`: restricts the output of `State.fromMarginal` to states with given number of running jobs, all assumed to be in service phase $k = 1$.

- `State.fromMarginalBounds`: similar to `State.fromMarginal`, but produces valid states between given minimum and maximum value of the number of resident jobs.

- `State.toMarginal`: 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 look at the state of station 3, which is a multi-server FCFS station. There are 4 classes all having exponential service times except class 2 that has Erlang-2 service times. We are interested to states with 2 running jobs in class 1 and 1 in class 2, and with 2 jobs, respectively of classes 3 and 4, waiting in the buffer. We can automatically generate this state space, which we store in the `space` variable, as:

```
>> cqn_multiserver;
>> space = State.fromMarginalAndRunning(model,node{3},[2,1,1,1],[2,1,0,0])
space =
     4       3       2       1       0       0       0
     4       3       2       0       1       0       0
     3       4       2       1       0       0       0
     3       4       2       0       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.fromMarginalAndStarted(model,node{3},[2,1,1,1],[2,1,0,0])
space =
     4       3       2       1       0       0       0
     3       4       2       1       0       0       0
```

We see that the above state space restricted the one obtained with `State.fromMarginalAndRunning` 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.fromMarginal` 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.fromMarginal(model,node{3},[2,1,1,1],[2,1,0,0])
space =
     4       3       2       1       0       0       0
     4       3       2       0       1       0       0
     4       2       2       0       0       1       0
     4       1       1       1       0       1       0
     4       1       1       0       1       1       0
     3       4       2       1       0       0       0
     3       4       2       0       1       0       0
     3       2       2       0       0       0       1
     3       1       1       1       0       0       1
     3       1       1       0       1       0       1
     2       4       2       0       0       1       0
```

```
2    3    2    0    0    0    1
2    1    1    0    0    1    1
1    4    1    1    0    1    0
1    4    1    0    1    1    0
1    3    1    1    0    0    1
1    3    1    0    1    0    1
1    2    1    0    0    1    1
1    1    0    1    0    1    1
1    1    0    0    1    1    1
```

Assigning a prior to an initial state

It is possible to assign the initial state to a station using the `setState` 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 `setStatePrior` and `getStatePrior` can be used to check and change the prior probabilities for the initial states specified for a station or stateful node.

4.3.2 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 `initFromMarginal`, `initFromMarginalAndRunning`, and `initFromMarginalAndStarted`. 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.isValid`.

It is also possible to request LINE to automatically identify a valid initial state, which is done using the `initDefault` 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 `initFromAvgQLen` method is a wrapper for `initFromMarginal` 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.

4.3.3   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.

4.3.4   State space generation

As discussed in Example 3, the state space of a model can be obtained by either invoking `model.stateSpace()` or `solver.stateSpace()` 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 5.2.2.

4.3.5   State probability analysis

Beyond average performance metrics, LINE provides capabilities for analyzing the probability distribution over network states, which is essential for understanding the fine-grained behavior of queueing systems. State probability analysis enables computation of joint state probabilities that capture the simultaneous occupancy of multiple stations, as well as marginal distributions that describe the probability of finding a specific number of jobs at an individual station regardless of the state of other stations. These distributions are particularly valuable for capacity planning, as they reveal the likelihood of congestion events and help identify the probability of exceeding service level thresholds. Joint state probabilities can be obtained using the `getProbSys` method, which returns the probability of each global system state in the state space. For large state spaces where enumerating all joint states becomes computationally prohibitive, the `getProbSysAggr` method provides aggregated probabilities by class across all stations, reducing the dimensionality while preserving key distributional information. Marginal state probabilities for individual stations are accessible

through `getProb`, which returns the full probability mass function for the number of jobs at a specified station, and `getProbAggr`, which aggregates these probabilities by class. State aggregation operations combine probabilities over subsets of states that share common properties, such as all states with a fixed total number of jobs or all states where a particular station is below capacity. The extraction and interpretation of state probabilities requires understanding the state encoding scheme used by the solver, where states are typically represented as vectors describing the number of jobs of each class at each station along with phase information for non-exponential service distributions. For FCFS scheduling, the state also encodes the order of jobs in the buffer, leading to larger state spaces than for processor-sharing or infinite-server stations. Practical examples demonstrating state probability extraction can be found in the provided example files, including `statepr_aggr.m` which shows how to aggregate state probabilities over classes and stations, and `statepr_allprobs_fcfs.m` which illustrates the enumeration and processing of all states in a FCFS queueing network. These examples demonstrate patterns such as computing the probability that a station is empty, the probability that the total number of jobs in the system exceeds a threshold, and the conditional probability distribution of one station's state given knowledge about another station's state.

4.4 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 $E[n(t)|s_0]$.

4.4.1 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_cqn(2) % closed single class queueing network with 2 stations
[Qt,Ut,Tt] = model.getTranHandles();
```

After solving the model, we will be able to retrieve *both* steady-state and transient averages as follows

```
[QNt,UNt,TNt] = CTMC(model,'timespan',[0,1]).tranAvg(Qt,Ut,Tt);
plot(QNt{1,1}.t, QNt{1,1}.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

```
>> CTMC(model).avgTable
State space size: 21 states.
CTMC analysis (method: default) completed in 0.187991 seconds.
ans =
  3x7 table
    Station     JobClass     QLen        Util      RespT      ResidT      Tput

    _____     _____     _____     _____   _____   _____   _____

    Delay 1     Class1       0.62015     0.62015        2          2       0.31008
    Queue 1     Class1       1.3018      0.62015    4.1984     4.1984      0.31008
    Queue 2     Class1       3.078       0.93023    9.9267     9.9267      0.31008
>> CTMC(model,'timespan',[0,1]).avgTable
State space size: 21 states.
CTMC analysis completed in 0.118398 sec
ans =
  3x7 table
    Station     JobClass     QLen        Util      RespT      ResidT      Tput

    _____     _____     _____     _____   _____      _____      _____

    Delay 1     Class1       3.0615      3.0615        0          0       1.5308
    Queue 1     Class1       1.6775      0.84395       0          0       0.42197
    Queue 2     Class1       0.26098     0.23425       0          0       0.078083
```

4.4.2   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 getTranCdfRespT, 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 getCdfRespT.

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 getTranCdfFirstPassT provides this distribution, assuming as initial state the one specified for the model, e.g., using setState or initDefault. This function is available only in FLD and has a similar syntax as getCdfRespT.

4.5   Sample path analysis

With LINE is also possible to obtain a particular sample path from the stochastic process underlying the queueing network. The following functions are available for this purpose:

- `sample`: returns a data structure including the time-varying state of a given stateful node, labelled with information about the events that changed the node state.

- `sampleAggr`: returns a data structure similar to the one provided by `sample`, but where the state is aggregate to count the number of jobs in each class at the node.

- `sampleSys`: similar to the `sample` function, but returns the state of every stateful node in the model.

- `sampleSysAggr`: similar to the `sampleAggr` function, but returns the aggregated state of every stateful node in the model.

It is worth noting that the JMT solver only supports `sampleAggr` since the simulator does not offer a simple way to extra detailed data such as phase change information in the service process. This information is instead available with the SSA solver.

For example, the following command extract a sample path consisting of 10 samples for a $APH(2)/M/1$ queue:

```
>> model=gallery_aphm1; samplePath = JMT(model).sampleAggr(model.nodes{2},1e1); ...
[samplePath.t, samplePath.state]
JMT Model: /tmp/jsimg/tpa52644b6_17ce_46ee_9c09_8da7c2d1d3c6.jsimg
ans =
         0         0
    0.3255    1.0000
    0.3562         0
    1.1478    1.0000
    1.2220         0
    1.8727    1.0000
    2.6076    2.0000
    2.6862    3.0000
    3.4737    2.0000
    3.7663    3.0000
    4.2110    2.0000
```

In the example, `samplePath.t` refers to the time since initialization at which the node 2 (here the $APH(2)/M/1$ queueing station) enters the state shown in the second column.

If we repeat the same experiment with the SSA solver and using the `sampleSys` function, we now have the full state space of the model, including both the source and the queueing station:

```
>> model=gallery_aphm1; samplePath = SSA(model).sampleSys(10); [samplePath.t, ...
samplePath.state{1}, samplePath.state{2}]

SSA analysis (method: default) completed in 0.011636 seconds.
ans =
    0.4776       Inf    1.0000         0         0         0
    0.5843       Inf         0    1.0000         0         0
    1.2043       Inf    1.0000         0         0    1.0000
```

```
    1.2952        Inf    1.0000           0           0           0
    1.3196        Inf         0      1.0000           0           0
    1.5724        Inf    1.0000           0           0      1.0000
    1.5975        Inf         0      1.0000           0      1.0000
    1.6129        Inf         0      1.0000           0           0
    1.6226        Inf    1.0000           0           0      1.0000
    2.5289        Inf    1.0000           0           0           0
```

4.6   Sensitivity analysis and numerical optimization

Frequently, performance and reliability analysis requires to change one or more model parameters to see the sensitivity of the results or to optimize some goal function. In order to do this efficiently, we have discussed before the internal representation of the `Network` objects used within the LINE solvers. By applying changes directly to this internal representation it is possible to considerably speed-up the sequential evaluation of models as discussed next.

4.6.1   Fast parameter update

Successive invocations of `getStruct()` will return a cached copy of the `NetworkStruct` representation, unless the user has called `model.refreshStruct()` or `model.reset()` in-between the invocations. The `refreshStruct` function regenerates the internal representation, while `reset` destroys it, together with all other representations and cached results stored in the `Network` object. In the case of `reset`, the internal data structure will be regenerated at the next `refreshStruct()` or `getStruct()` call.

The performance cost of updating the representation can be significant, as some of the structure array field require a dedicated algorithm to compute. For example, finding the chains in the model requires an analysis of the weakly connected components of the network routing matrix. For this reason, the `Network` class provides several functions to selectively refresh only part of the `NetworkStruct` representation, once the modification has been applied to the objects (e.g., stations, classes, ...) used to define the network. These functions are as follows:

- `refreshArrival`: this function should be called after updating the inter-arrival distribution at a `Source`.

- `refreshCapacity`: this function should be called after changing buffer capacities, as it updates the `capacity` and `classcapacity` fields.

- `refreshChains`: this function should be used after changing the routing topology, as it refreshes the `rt`, `chains`, `nchains`, `nchainjobs`, and `visits` fields.

- `refreshPriorities`: this function updates class priorities in the `classprio` field.

- `refreshScheduling`: updates the `sched`, and `schedparam` fields.

Table 4.3: 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.

- `refreshProcesses`: updates the `mu`, `phi`, `phases`, `rates` and `scv` fields.

For example, suppose we wish to update the service time distribution for class-1 at node 1 to be exponential with unit rate. This can be done efficiently as follows:

```
queue.setService(class1, Exp(1.0));
model.refreshProcesses;
```

4.6.2 Direct modification of NetworkStruct objects

For maximum performance in optimization loops, LINE provides low-level `sn_set_*` functions that directly modify `NetworkStruct` objects without requiring a `Network` object. These functions bypass the high-level API and operate directly on the internal data structure, making them ideal for sensitivity analysis and numerical optimization where many parameter variations must be evaluated quickly.

The available functions are listed in Table 4.3.

For example, suppose we wish to vary the service rate at station 1 for class 1 in an optimization loop:

```
sn = model.getStruct();  % Get NetworkStruct once
for rate = 0.5:0.1:2.0
    sn = sn_set_service(sn, 1, 1, rate);
    % Solve directly with sn
    result = solver.solve(sn);
end
```

4.6.3   Refreshing a network topology with non-probabilistic routing

The `resetNetwork` function should be used before changing a network topology with non-probabilistic routing. It will destroy by default all class switching nodes. This can be avoided if the function is called as, e.g., `model.resetNetwork(false)`. The default behavior is though shown in the next example

```
>> model = Network('model');
node{1} = ClassSwitch(model,'CSNode',[0,1;0,1]);
node{2} = Queue(model, 'Queue1', SchedStrategy.FCFS);
>> model.getNodes
ans =
  2x1 cell array
    {1x1 ClassSwitch}
    {1x1 Queue}
>> model.resetNetwork
ans =
  1x1 cell array
    {1x1 Queue}
```

As shown, `resetNetwork` updates the station indexes and the revised list of nodes that compose the topology is obtained as a return parameter. To avoid stations to change index, one may simply create `ClassSwitch` nodes as last before solving the model. This node list can be employed as usual to reinstantiate new stations or `ClassSwitch` nodes. The `addLink`, `setRouting`, and possibly the `setProbRouting` functions will also need to be re-applied as described in the previous sections.

4.6.4   Saving a network object before a change

The Network object, and its inner objects that describe the network elements, are always passed by reference. The `copy` function should be used to clone LINE objects, for example before modifying a parameter for a sensitivity analysis. This function recursively clones all objects in the model, therefore creating an independent copy of the network. For example, consider the following code

```
modelByRef = model; modelByRef.setName('myModel1');
modelByCopy = model.copy; modelByCopy.setName('myModel2');
```

Using the `getName` function it is then possible to verify that `model` has now name `myModel1`, since the first assignment was by reference. Conversely, `modelByCopy.setName` did not affect the original `model` since this is a clone of the original network.

4.7   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.

4.7.1 Chain aggregation

The `ModelAdapter.aggregateChains` 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.

```
[chainModel, alpha, deaggInfo] = ModelAdapter.aggregateChains(model);
```

4.7.2 Flow-equivalent server aggregation

The `ModelAdapter.aggregateFES` function replaces a subset of stations in a closed product-form queueing network with a single Flow-Equivalent Server (FES). The FES has state-dependent service rates where the rate for each class equals the throughput of that class in an isolated subnetwork consisting only of the aggregated stations. This technique reduces the model size while preserving exact throughput and utilization metrics.

```
[fesModel, fesStation, deaggInfo] = ModelAdapter.aggregateFES(model, {queue2, ...
queue3});
```

4.7.3 Tagged job models

The `ModelAdapter.tagChain` 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.

```
[taggedModel, taggedJob] = ModelAdapter.tagChain(model, chain, jobclass);
```

4.7.4 Class removal

The `ModelAdapter.removeClass` 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.

```
newmodel = ModelAdapter.removeClass(model, jobclass);
```

Chapter 5

Network solvers

5.1 Overview

Solvers analyze objects of class Network 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 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,'exact').avgTable()
MVA(model,'method','exact').avgTable()
opt = MVA.defaultOptions; opt.method = 'exact'; MVA(model,opt).avgTable()
```

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 Network 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 [49]. 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 [4]. 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.

- `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. 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 Network 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`) [23], Rolia's multiserver (`rolia`) [56], load-dependent MVA by Reiser-Lavenberg (`reiser`) [53], and Zhou-Woodside's multiserver approximation (`zhou`) [67]. This solver is implemented by the `QNS` class.

5.2 Solution methods

We now describe the solution methods available within the Network solvers. Table 5.1 provides a global summary. Some of the listed methods (e.g., `mg1`) are not associated to a specific solver, as they do not fall in one of the reference formalisms. A solver that runs these methods can be instantiated as follows, e.g.:

```
>> solver = LINE.load('mg1',model);
>> solver.avgTable();
```

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);
```

Table 5.1: Solution methods for Network solvers.

Solver	Method	Description	Refs.
CTMC	default	Solution based on global balance	[8, § 2.1.2]
DES	default	Discrete-event simulation using SSJ and SimPy libraries	–
FLUID	default	ODE-based mean field approximations	[50, 57]
FLUID	matrix	Alias for the default method	[50, 57]
FLUID	closing	Fluid with closing method for open classes	[8, p. 507]
FLUID	statedep	Kurtz's mean field ODEs for closed models	[50]
FLUID	softmin	Smoothed statedep with *softmin* replacing *min* functions	–
FLUID	diffusion	Diffusion approximation via Euler-Maruyama SDEs	[8, § 10.1.1]
FLUID	mfq	Markovian fluid queue for single-queue systems	[37]
JMT	default	Alias for the jsim method	–
JMT	jmva	Alias for the jmva.mva method	–
JMT	jmva.mva	Exact MVA in JMVA	[53]
JMT	jmva.recal	Exact RECAL algorithm in JMVA	[24]
JMT	jmva.comom	Exact CoMoM algorithm in JMVA	[12]
JMT	jmva.amva	Approximate MVA, alias for jmva.bs.	–
JMT	jmva.aql	AQL algorithm in JMVA	[66]
JMT	jmva.bs	Bard-Schweitzer algorithm in JMVA	[8, § 9.1.1]
JMT	jmva.chow	Chow algorithm in JMVA	[22]
JMT	jmva.dmlin	De Souza-Muntz Linearizer in JMVA	[26]
JMT	jmva.lin	Linearizer algorithm in JMVA	[21]

Table 5.1 – Solution methods for `Network` solvers. *Continued from previous page*

Solver	Method	Description	Refs.
JMT	`jmva.ls`	Logistic sampling in JMVA	[13]
JMT	`jsim`	Exact discrete-event simulation in JSIM	[4]
JMT	`replication`	Transient simulation via independent replications	–
MAM	`default`	Matrix-analytic solution of structured QBDs	[38]
MAM	`dec.source`	Decomposition with arrivals as from the source	–
MAM	`dec.poisson`	Decomposition based on Poisson arrival flows	–
MAM	`dec.mna`	Decomposition based on MNA method	[44]
MAM	`inap`	Iterative Numerical Approximation Procedure via RCAT	[15]
MAM	`inapplus`	Improved INAP with weighted rates (no normalization)	[15]
MVA	`default`	Approximate MVA, same as `qd` option	–
MVA	`amva`	Approximate MVA, same as `qd` option	–
MVA	`bs`	Bard-Schweitzer approximate MVA	[8, § 9.1.1]
MVA	`lin`	Linearizer approximate MVA	[21]
MVA	`qd`	Queue-dependent approximate MVA	[18]
MVA	`qdlin`	Queue-dependent Linearizer approximate MVA	–
MVA	`exact`	Exact solution, method depends on model	–
MVA	`mva`	Alias for the `mva.amva` method	[53], [11]
MVA	`aba.upper`	Asymptotic bound analysis (upper bounds)	[8, § 9.4]
MVA	`aba.lower`	Asymptotic bound analysis (lower bounds)	[8, § 9.4]
MVA	`bjb.upper`	Balanced job bounds (upper bounds)	[17, Table 3]
MVA	`bjb.lower`	Balanced job bounds (lower bounds)	[17, Table 3]
MVA	`gb.upper`	Geometric square-root bounds (upper bounds)	[17]
MVA	`gb.lower`	Geometric square-root bounds (lower bounds)	[17]
MVA	`pb.upper`	Proportional bounds (upper bounds)	[17, Table3]
MVA	`pb.lower`	Proportional bounds (lower bounds)	[17, Table3]
MVA	`sb.upper`	Simple bounds (upper bounds, Thm. 3.2, $n=3$)	[35, Table3]
MVA	`sb.lower`	Simple bounds (lower bounds, Eq. 1.6)	[35, Table3]
MVA	`gig1.allen`	Allen-Cunneen formula - GI/G/1	[8, § 6.3.4]
MVA	`gig1.heyman`	Heyman formula - GI/G/1	–
MVA	`gig1.kingman`	Kingman upper bound- GI/G/1	[8, § 6.3.6]
MVA	`gig1.klb`	Kramer-Langenbach-Belz formula - GI/G/1	[8, § 6.3.4]
MVA	`gig1.kobayashi`	Kobayashi diffusion approximation - GI/G/1	[8, § 10.1.1]
MVA	`gig1.marchal`	Marchal formula - GI/G/1	[8, § 10.1.3]
MVA	`gigk`	Kingman approximation - GI/G/k	

Table 5.1 – Solution methods for `Network` solvers. *Continued from previous page*

Solver	Method	Description	Refs.
MVA	mg1	Pollaczek–Khinchine formula - M/G/1	[8, § 3.3.1]
MVA	mg1.fb	Feedback/LAS scheduling - M/G/1/FB	[64]
MVA	mg1.lrpt	Longest remaining PT - M/G/1/LRPT	[64]
MVA	mg1.psjf	Preemptive SJF - M/G/1/PSJF	[64]
MVA	mg1.srpt	Shortest remaining PT - M/G/1/SRPT	[64]
MVA	mg1.setf	Shortest elapsed time first - M/G/1/SETF	[48]
MVA	mm1	Exact formula - M/M/1	[8, § 6.2.1]
MVA	mmk	Exact formula - M/M/k (Erlang-C)	
NC	default	Alias for the `adaptive` method	–
NC	adaptive	Automated choice of deterministic method	–
NC	exact	Automated choice of exact solution method.	–
NC	ca	Multiclass convolution algorithm (exact)	–
NC	comom	Class-oriented method of moments for hommo-geneous models (exact)	[12]
NC	cub	Grundmann-Moeller cubature rules	[13]
NC	mva	Product of throughputs on MVA lattice (exact)	[52, Eq. (47)]
NC	imci	Improved Monte carlo integration sampler	[63]
NC	kt	Knessl-Tier asymptotic expansion	[39]
NC	le	Logistic asymptotic expansion	[13]
NC	ls	Logistic sampling	[13]
NC	nr.logit	Norlund-Rice integral with logit transformation	[16]
NC	nr.probit	Norlund-Rice integral with probit transformation	[16]
NC	panacea	Panacea asymptotic expansion	[47], [55]
NC	rd	Reduction heuristic	[16]
NC	sampling	Automated selection of sampling method	–
NC	mem	Maximum Entropy Method for open queueing networks	[41]
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).	[1]
SSA	serial	CTMC stochastic simulation on a single core	[34]
SSA	para	Parallel simulations (independent replicas)	–

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 RCAT convergence is slow.

FLUID method selection. The default FLUID 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.

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. 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. For open networks, the `mem` method applies the Maximum Entropy Method to estimate the joint queue-length distribution.

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.

5.2.1 `AUTO`

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.

Therefore, heuristics are used to identify a solver based on structural properties of the model, such as based on the scheduling strategies used at the stations as well as the number of jobs, chains, and classes. The AUTO solver extracts structural features from the queueing network model and uses them to select the most appropriate solution method. The feature extraction process analyzes characteristics including the proportion of FCFS, processor-sharing, and delay stations, the presence of class-switching nodes, the average

number of servers per queue, the number of jobs per chain, and the distribution of service time processes. These features are normalized and combined with derived metrics that capture ratios and proportions of key structural properties. The selection mechanism supports both heuristic-based decision making and machine learning classification using trained ensemble models. In , a trained classifier stored in ONNX format can optionally be used to predict the optimal solver based on patterns observed in a training dataset of queueing network models. The classifier is loaded from disk and applied to the normalized feature vector to determine the best method among options such as MVA, CTMC, JMT, and NC solvers. This automated selection process simplifies the modeling workflow by removing the need for manual solver tuning, particularly beneficial when analyzing large collections of models or when the modeler is uncertain about the relative performance characteristics of different solvers for a given network structure. When AUTO determines that a solver is appropriate but it does not support the specific function called (e.g., `getTranAvg` might not be available in all solvers), it examines the list of feasible alternative solvers and prioritizes them in execution time order, with the fastest one on average having the higher priority. Eventually, the solver will always be able to identify a solution strategy, through at least simulation-based solvers such as `JMT` or `SSA`. Users should consider employing AUTO instead of manual solver selection when exploring new model structures, when benchmarking solver performance across model families, or when embedding LINE in automated workflows where solver expertise cannot be assumed. However, for production environments where the optimal solver is already known through empirical evaluation, directly instantiating the specific solver class avoids the overhead of feature extraction and classification. The fallback strategy ensures robustness by progressively relaxing requirements until a compatible solver is found, guaranteeing that analysis can proceed even for unusual or edge-case model configurations.

5.2.2   CTMC

The `CTMC` class solves the model by first generating the infinitesimal generator of the Network 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.

Transient analysis is carried out by numerically solving Kolmogorov's forward equations using MATLAB's ODE solvers. The range of integration is controlled by the `timespan` option. The ODE solver choice is the same as for `FLD`.

The CTMC solver heuristically limits the solution to models with no more than 6000 states. The `force` option needs to be set to `true` to bypass this control. In models with infinite states, such as networks with open classes, the `cutoff` option should be used to reduce the CTMC to a finite process. If specified as a scalar value, `cutoff` is the maximum number of jobs that a class can place at an arbitrary station. More generally, a matrix assignment of `cutoff` indicates to LINE that `cutoff` has in row i and column r the maximum number of jobs of class r that can be placed at station i.

The following methods are available for the `CTMC` solver:

- `options.method='default'`: Direct solution of the global balance equations.

- `options.method='gpu'`: GPU-accelerated solution of the global balance equations.

Details on the additional configuration options of the CTMC solver is given in the next table.

Table 5.2: CTMC configuration options (MATLAB)

Option	Value	Description
`options.config.hide_immediate`	logical	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.
`options.timestep`	double	Fixed time interval for transient analysis. If specified, generates equally-spaced time points instead of adaptive stepping.

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.getReward()`, which returns transient reward trajectories over time. Example applications include computing expected queue lengths, utilization, blocking probabilities, or custom cost functions.

Reward functions are defined using `model.setReward(name, rewardFn)`, where `name` is a string identifier and `rewardFn` is a function handle `@(state, sn)` that maps a state vector and the network structure to a scalar reward value. See `example_rewardModel_1.m` in the `rewardModel` examples folder.

5.2.3 `FLD`

This solver implements fluid/mean-field approximation methods, based on the system of fluid ordinary differential equations for INF-PS queueing networks presented in [50]. The latter is based on Kurtz's mean-field approximation theory. The fluid ODEs are normally solved with the `'NonNegative'` ODE solver option enabled. Four types of ODE solvers are used: *fast* or *accurate*, the former only if `options.iter_tol>` 10^{-3}, and *stiff* or *non-stiff*, depending on the value of `options.stiff`. The default choice of solver is stored in the following static functions:

- `Solver.accurateStiffOdeSolver`, set to MATLAB's `ode15s`.

- `Solver.accurateOdeSolver`, set to `ode45`.

- `Solver.fastStiffOdeSolver`, set to `ode23s`.

- `Solver.fastOdeSolver`, set to `ode23`.

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.

The solution of models with FCFS stations maps these stations into corresponding PS stations where the service rates across classes are set identical to each other with a service distribution given by a mixture of the service processes of the service classes. The mixture weights are determined iteratively by solving a sequence of PS models until convergence. Upon initializing FCFS queues, jobs in the buffer are all initialized in the first phase of the service.

The following methods are available for the `FLD` solver:

- `options.method='default'` or `'matrix'`: Fluid ODE solver using matrix formulation per Ruuskanen et al. [57]. Supports both closed and open queueing networks.

- `options.method='closing'` or `'statedep'`: State-dependent fluid solver with closing approximations. Useful for models with DPS scheduling.

- `options.method='diffusion'`: Diffusion approximation for closed multiclass BCMP networks using the Euler-Maruyama method for stochastic differential equations. This method models queue lengths as continuous variables with Brownian noise, extending deterministic fluid models to capture stochastic fluctuations. Presently, this method supports only product-form closed networks.

- `options.method='mfq'` (Markovian Fluid Queue): Exact steady-state analysis for single-queue open networks with phase-type arrivals and service times, based on the BUTools `FluFluQueue` implementation [38]. This method computes exact queue-length and sojourn time moments from the fluid flow equations and is restricted to single-queue networks (Source → Queue → Sink topology). Closed networks and networks with multiple queues are not supported.

5.2.4 JMT

The class is a wrapper for the JMT and consists of a model-to-model transformation from the Network 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.

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.hasProductFormSolution` 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.

5.2.5 `MAM`

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 [38]. 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 from Masuyama and Takine [46]. The number of CDF points computed can be controlled via `options.config.num_cdf_pts` (default: 10000).

The following methods are available for the `MAM` solver:

- `options.method='default'` or `'dec.source'` (Arrival Stream Decomposition): Decomposes the network by approximating the arrival process at each downstream station using the distribution observed at the source station, scaled by the corresponding visit ratio. This is the default decomposition method and supports open, closed, and mixed queueing networks.

- `options.method='dec.mna'` (Matrix Network Analyzer): A decomposition method based on the Matrix Network Analyzer (MNA) algorithm [44], which computes more precise departure processes from each queue and therefore achieves higher accuracy than `dec.source`. This method is recommended for networks with Markovian arrival processes and supports FCFS and HOL scheduling disciplines.

- `options.method='dec.poisson'` (Poisson Arrival Approximation): Simplifies all inter-station arrival flows to Poisson processes, effectively ignoring burstiness in inter-station traffic. This is the fastest but least accurate decomposition method, suitable for rapid exploratory analysis or as a baseline approximation.

- `options.method='inap'` (INAP): Iterative Numerical Approximation Procedure [15] based on the Reversed Compound Agent Theorem (RCAT), which finds product-form solutions for certain queueing network configurations by iteratively solving the equilibrium equations of coupled Markovian processes.

- `options.method='inapplus'` (Improved INAP): An improved variant of INAP [15] that accumulates weighted equilibrium rates without normalizing by destination probabilities. This modification offers improved numerical stability and faster convergence compared to `inap`, and is recommended when RCAT convergence is slow.

5.2.6 `MVA`

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 [21], unless there are two or less jobs in total within closed classes, in which case the solver runs the Bard-Schweitzer algorithm [58].

Extended queueing models are handled as follows:

- Non-exponential service times in FCFS nodes are handled only in the single-server case via the method selected in the `options.config.highvar` setting. By default high variance is ignored, as the FCFS solver tends to produce good result in closed models also without specialized corrections. It is alternatively possible to handle high variance either using the Diffusion-M/G/k interpolation from [14], casted with weights $a_i = b_i = 10^{-8}$, or using the high-variance MVA (HV-MVA) corrections proposed in [9, 51]. The multi-server extension is ongoing; we point to the NC solver for a version already available.

- Multi-servers are dealt with using the methods listed in Table 5.3 for the `options.config.multiserver` option. These are coupled with a modification of the Rolia-Sevcik correction [56], where in light-load the Rolia-Sevcik correction is treated as if there was a single server.

- Non-preemptive are dealt with using the methods listed in Table 5.3 for the configuration option `options.config.np_priority`. The solver feature in particular AMVA-CL and the shadow server methods [28].

- DPS queues are analyzed with a standard method similar to the biased processor sharing approximation reported in [43, §11.4]. Here, an arriving job of class r sees a queue-length in class $s \neq r$ scaled by the correction factor w_s/w_r, where w_s is the weight of class s.

- Limited load-dependence (intended here as other than multi-server) and class-dependence are handled through the correction factors proposed in [18]. If a station is both limited load-dependent and multi-server, then if the `softmin` method is chosen the solver will suitably combine the softmin term and the limited load-dependent correcting factors. Moreover, iterative queue-length corrections such as those applied by the AQL and Linearizer methods are also applied to these terms. Limited load-dependence (queue-dependent AMVA or QD-AMVA) is handled through correction factors.

- Fork-join networks are assumed to feature a direct acyclic graph (DAG) in-between forks and joins. They are analyzed by iteratively transforming the sibling tasks into jobs belonging to independent classes, using the algorithm specified in `options.config.fork_join`. If a fork has fan out f (i.e., the fork out-degree), in the implementation of the Heidelberger-Trivedi [36] method, one artificial open class is created for each of $f - 1$ sibling task, while also retaining a task in the original class. The residence times along a branch are then treated as exponential random variables and their maximum, corresponding to the response time of the fork-join section, is computed using specialized results for

this distribution. LINE supports this method, but uses as a default a custom variant whereby in which the original and artificial classes can take with probability $1/f$ any of the outgoing branches. While the latter can result in states that do not exist in the original model, since two sibling tasks may take the same branch, it is correct in expectation and it does not treat differently the artificial classes than the original class, which can be beneficial when the original class is *closed* and thus differs significantly from an *open* artificial class.

Solver-specific configuration options are reported in Table 5.3.

Table 5.3: MVA configuration options (MATLAB)

Option	Value	Description
options.config.multiserver	'default'	Equals 'softmin' at PS queues and 'seidmann' at FCFS queues.
options.config.multiserver	'seidmann'	Seidmann's decomposition [59].
options.config.multiserver	'softmin'	QD-AMVA's softmin approximation [18].
options.config.np_priority	'default'	Non-preemptive priority handling. Equals 'cl'.
options.config.np_priority	'cl'	Chandy-Lakshmi [28].
options.config.np_priority	'shadow'	Sevcik's shadow server [60].
options.config.highvar	'default'	Ignored - no correction applied.
options.config.highvar	'interp'	Diffusion-M/G/k interpolation from [14].
options.config.highvar	'hvmva'	High-variance MVA as in [9], extended to multiclass as [29, Eq. 3.21].
options.config.fork_join	'default'	Equals 'mmt'.
options.config.fork_join	'mmt'	Mixed-model transformation [27]
options.config.fork_join	'ht'	Heidelberger-Trivedi [36]

5.2.7 NC

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 typicallly maps the problem to certain multidimensional integrals, allowing the use of numerical methods such as MonteCarlo sampling and asymptotic expansions in their approximation.

For cache models, the NC solver supports an importance sampling method ('sampling') that uses Monte Carlo techniques to estimate cache hit probabilities and miss rates. This method is particularly useful for large cache configurations where exact methods become computationally expensive. The sampling method can be enabled by setting the method option:

```
NC(model, 'method', 'sampling', 'samples', 100000).avgTable()
```

The following deterministic methods are available for the NC solver:

- `options.method='cub'` (Cubature): Numerical integration of the normalizing constant using Grundmann-Müller cubature rules over the simplex of job populations [13]. This method provides high accuracy for small to medium population networks but becomes computationally expensive as the population grows.

- `options.method='nr.logit'` (Norlund-Rice Logistic): Approximates the normalizing constant by applying a Laplace (saddle-point) approximation with a logistic sigmoid transformation of the integration variables [16]. Suitable for load-dependent networks with moderate to large populations.

- `options.method='nr.probit'` (Norlund-Rice Probit): Similar to `nr.logit` but uses the normal cumulative distribution function (probit transform) instead of the logistic sigmoid [16]. Offers an alternative parametrization of the saddle-point approximation for load-dependent networks.

- `options.method='rd'` (Reduction): An approximation method for load-dependent networks that decomposes the service rate functions into a gamma correction factor applied to the effective network demands [16]. This heuristic enables fast steady-state analysis of state-dependent queueing systems.

- `options.method='mem'` (Maximum Entropy): Estimates performance metrics for open queueing networks by maximizing the entropy of the queue-length distribution subject to mean-value constraints [41]. This method applies only to open networks and converges iteratively to a steady-state solution using flow-balance equations.

5.2.8 `SSA`

The `SSA` class is a basic stochastic simulator for continuous-time Markov chains. It reuses some of the methods that underpin `CTMC` to generate the network state space and subsequently simulates the state dynamics by probabilistically choosing one among the possible events that can incur in the system, according to the state spaces of each of node in the network. For efficiency reasons, states are tracked at the level of individual stations and, in some of the algorithms, hashed.

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 `getInitState` functions of the `Network` class.

The `SSA` solver offers two comprehensive methods: `'serial'` and `'para'` (default). The serial method runs on a single core, while the parallel methods run on multicore via MATLAB's `spmd` command.

`SSA` solver also implements the much faster next reaction method (`'nrm'`, see [1]). However, this is available only for open and closed queueing models with specific scheduling disciplines, in particular `INF` and `PS`. The `'nrm'` method is 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.

5.2.9 `Posterior`

The `Posterior` 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 `Posterior` 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 `Posterior` 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.

The constructor for the `Posterior` 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 = Posterior(model, @SolverMVA);
```

The `Posterior` solver provides several methods for retrieving results. The `getAvgTable` method returns the prior-weighted expected values of all performance metrics, effectively averaging over the parameter uncertainty. The `getPosteriorTable` method returns the complete set of results for each alternative, allowing the analyst to examine how performance varies across different parameter values. Additionally, the `getPosteriorDist` 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.

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.setService(jobClass, 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 `Posterior` 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.

5.3   Supported language features and options

5.3.1   Solver features

Once a model is specified, it is possible to use the `getUsedLangFeatures` 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.getUsedLangFeatures.list.SchedStrategy_FCFS)
...
```

Every LINE solver implements the `support` to check if it supports all language features used in a certain model

```
>> JMT.supports(model)
ans =
  logical
   1
```

5.3.2   State-dependent routing and service

LINE supports state-dependent behaviors that allow routing decisions and service rates to vary based on the current system state. State-dependent routing strategies enable dynamic job dispatching policies that adapt to queue lengths or other system conditions. The RROBIN (Round-Robin) strategy cycles through destination nodes in a fixed sequence, distributing jobs evenly across multiple servers or queues. This is particularly useful for load balancing in systems with parallel service resources. The JSQ (Join Shortest Queue) strategy routes each arriving job to the destination with the smallest queue length, providing dynamic load balancing that responds to instantaneous system conditions. The WRROBIN (Weighted Round-Robin) strategy generalizes round-robin dispatching by assigning different weights to destinations, allowing proportional load distribution. State-dependent service rates can be configured using the `setLoadDependence` method, which allows service rates to scale based on the number of jobs present at a station. This enables modeling of congestion effects, speed scaling, or batch processing behaviors. Among the solvers, CTMC provides complete support for all state-dependent features through its explicit state space representation. The JMT solver supports these features through discrete event simulation, making it suitable for validating state-dependent models. The MVA solver supports load-dependent service rates through specialized mean value analysis algorithms, though it does not handle general state-dependent routing. The following example demonstrates RROBIN routing at a Router node, where jobs are dispatched to three parallel queues in round-robin fashion.

```
router = Router(model, 'Dispatcher');
queue1 = Queue(model, 'Queue1', SchedStrategy.FCFS);
queue2 = Queue(model, 'Queue2', SchedStrategy.FCFS);
queue3 = Queue(model, 'Queue3', SchedStrategy.FCFS);
```

```
router.setRouting(jobclass, RoutingStrategy.RROBIN, ...
    {queue1, queue2, queue3});
```

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.

5.3.3 Class functions

The table below lists the steady-state and transient analysis functions implemented by the Network 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.

Table 5.4: Solver support for average performance metrics

Function	Regime	Network Solver							
		CTMC	DES	FLD	JMT	MAM	MVA	NC	SSA
avg	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avgTable	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avgChain	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avgChainTable	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avgNode	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avgNodeTable	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avgNodeChain	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avgNodeChainTable	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avgSys	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avgSysTable	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avgArvR	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avgArvRChain	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avgNodeArvRChain	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avgQLen	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avgQLenChain	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avgNodeQLenChain	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avgRespT	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avgRespTChain	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avgNodeRespTChain	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avgSysRespT	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avgTput	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avgTputChain	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avgNodeTputChain	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avgSysTput	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avgUtil	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avgUtilChain	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
avgNodeUtilChain	Steady-state	✓	✓	✓	✓	✓	✓	✓	✓
getTranAvg	Transient	✓	✓	✓	✓				

Table 5.5: Solver support for advanced metrics

Function	Regime	CTMC	DES	FLD	JMT	MAM	MVA	NC	SSA
		Network Solver							
getCdfRespT	Steady-state	✓	✓	✓	✓	✓			
getAvgAoI	Steady-state				✓				
getCdfAoI	Steady-state				✓				
getProb	Steady-state	✓	✓						
getProbAggr	Steady-state	✓	✓	✓	✓		✓	✓	
getProbSys	Steady-state	✓	✓						✓
getProbSysAggr	Steady-state	✓	✓		✓			✓	
getProbNormConstAggr	Steady-state				✓		✓	✓	
getTranCdfPassT	Transient		✓	✓					
getTranCdfRespT	Transient		✓		✓				
getTranProb	Transient	✓	✓						
getTranProbAggr	Transient	✓	✓						
getTranProbSys	Transient	✓	✓						
getTranProbSysAggr	Transient	✓	✓						
sample	Transient		✓						✓
sampleAggr	Transient		✓		✓				✓
sampleSys	Transient		✓						✓
sampleSysAggr	Transient		✓		✓				✓

The functions listed above with the `Table` suffix (e.g., `avgTable`) 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 each station and class.

- `avgChain`: returns the mean queue-length, utilization, mean response time (for one visit), and throughput for every station and chain.

- `avgSys`: returns the system response time and system throughput, as seen as the reference node, by chain.

- `getCdfRespT`: returns the distribution of response times (for one visit) for the stations at steady-state.

- `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.

- `avgNode`: 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.

- `getProb`: returns state probabilities at equilibrium at a given station.

- `getProbAggr`: returns marginal state probabilities for jobs of different classes at a given station.

- `getProbSys`: returns joint probabilities for a given system state.

- `getProbSysAggr`: returns joint probabilities for jobs of different classes at all stations.

- `getProbNormConstAggr`: returns the normalizing constant of the state probabilities for the model.

- `getTranAvg`: returns transient mean queue length, utilization and throughput for every station and chain from a given initial state.

- `getTranCdfPassT`: returns the distribution of first passage times in transient regime.

- `getTranCdfRespT`: returns the distribution of response times in transient regime.

- `sample`: returns the transient marginal state for a station from a given initial state.

- `sampleAggr`: returns the transient marginal state for jobs of different classes at a given station from a given initial state.

- `sampleSys`: returns the transient marginal system state for a station from a given initial state.

- `sampleSysAggr`: returns the transient marginal system state for jobs of different classes at a given station from a given initial state.

5.3.4 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 Sink and Source nodes, but due to low accuracy when run on open models this feature is disabled in the current release.

5.3.5 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 restart independent (`SchedStrategyType.PI`)

Table 5.6: Solver support for Network nodes

Strategy	Network Solver							
	CTMC	DES	FLD	JMT	MAM	MVA	NC	SSA
Cache	✓	✓		✓		✓	✓	✓
ClassSwitch	✓	✓	✓	✓	✓	✓	✓	✓
Delay	✓	✓	✓	✓	✓	✓	✓	✓
Fork		✓		✓		✓		
Join		✓		✓		✓		
Queue	✓	✓	✓	✓	✓	✓	✓	✓
Sink	✓	✓	✓	✓	✓	✓	✓	✓
Source	✓	✓	✓	✓	✓	✓	✓	✓

- non-preemptive (`SchedStrategyType.NP`)

- non-preemptive priority (`SchedStrategyType.NPPrio`).

The `SchedStrategyType.PR` policy class includes scheduling strategies like `LCFSPR`, where a pre-empted job resumes service from the point it was interrupted. In contrast, the `SchedStrategyType.PI` 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 scheduling strategies listed in the table include both common and specialized disciplines. Among the less common strategies, LAS (Least Attained Service) is a preemptive discipline that prioritizes the job with the smallest accumulated service time, which can be optimal for minimizing mean slowdown in certain systems. The EDD (Earliest Due Date) strategy serves jobs in order of their deadlines, making it suitable for systems with time-critical workloads. The FB (Foreground-Background) strategy is a preemptive variant of processor sharing that divides jobs into priority levels based on their attained service. The POLLING strategy encompasses several disciplines for multi-queue systems where a single server cycles through queues according to exhaustive, gated, or limited policies. The SEPT (Shortest Expected Processing Time) strategy is optimal for minimizing mean response time when job sizes are known or can be estimated, and is closely related to SJF but based on expected rather than exact service times. The LEPT (Longest Expected Processing Time) strategy serves the longest jobs first, which can be useful in certain fairness-oriented scenarios. The KCHOICES strategy implements the power-of-k-choices routing, where arriving jobs are assigned to the shortest of k randomly sampled queues, providing load balancing with low coordination overhead. Finally, the RL (Reinforcement Learning) strategy enables adaptive scheduling policies learned through interaction with the system. These specialized strategies are primarily supported by the CTMC, DES, JMT, and SSA solvers, as indicated in the feature support table that follows. Examples

Table 5.7: Solver support for scheduling strategies

Strategy	Class	Network Solver							
		CTMC	DES	FLD	JMT	MAM	MVA	NC	SSA
FCFS	NP	✓	✓	✓	✓	✓	✓	✓	✓
INF	NP	✓	✓	✓	✓	✓	✓	✓	✓
SIRO	NP	✓	✓		✓		✓	✓	✓
SEPT	NP	✓	✓		✓				✓
SRPT	NP		✓		✓				
SJF	NP		✓		✓				
FCFSPRIO	NPPrio	✓	✓		✓		✓		✓
EDD	NPPrio		✓		✓				
EDF	PRPrio		✓		✓				
PS	PR	✓	✓	✓	✓	✓	✓	✓	✓
DPS	PR	✓	✓	✓	✓		✓		✓
GPS	PR	✓	✓		✓				✓
PSPRIO	PRPrio	✓	✓		✓				✓
DPSPRIO	PRPrio	✓	✓		✓				✓
GPSPRIO	PRPrio	✓	✓		✓				✓

demonstrating these strategies can be found in the examples directory, such as the polling examples showing exhaustive and gated polling disciplines, and the DPS examples illustrating discriminatory processor sharing with class-dependent weights.

5.3.6 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.

5.3.7 Solver options

Table 5.9 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. The global defaults for the solvers can be manually adjusted by editing the `lineDefaults.m` file in the root folder.

This can be specified as an argument to the constructor of the solver. For example, the following two constructor invocations are identical

Table 5.8: Solver support for statistical distributions

Distribution	Network Solver							
	CTMC	DES	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		✓		✓				

```
s = JMT(model)
opt = JMT.defaultOptions; s = JMT(model, opt)
```

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',1e6)
opt = JMT.defaultOptions; opt.samples=1e6; s = JMT(model, opt)
```

Available solver options are as follows:

- `cache` (`logical`) if set to `true` the solver after the first invocation will return the same result upon subsequent calls, without solving again the model. This option is `true` by default. Caching can be bypassed using the `refresh` methods (see Section 4.6).

- 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. This is a mandatory option to analyze open classes using the CTMC solver.

- 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 to convergence of iterative methods. In the FLD solver this option regulates both the absolute and relative tolerance of the ODE solver.

- 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 DES, 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, DES). 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 $\pm$ half-width. Default is false (disabled).

- cimethod (string, DES only) specifies the confidence interval computation method. Valid values are 'obm' (overlapping batch means, default), 'bm' (non-overlapping batch means), or 'none' (disable CI computation).

- ciminbatch (integer ≥ 2, DES only) minimum batch size for confidence interval computation. Actual batch size is $\max(\textbf{ciminbatch}, \sqrt{n})$ where n is the sample count. Default is 10.

- ciminobs (integer ≥ 10, DES only) minimum number of post-warmup observations required before computing confidence intervals. Default is 100.

- config.variates (string, DES 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'.

- 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 language to run the chosen method, either 'matlab' (default) or 'java'. The Java codebase does not support all available methods in MATLAB, but it is faster.

- method (string) configures the internal algorithm used to solve the model.

- samples (integer ≥ 1) controls the number of samples collected *for each* performance index by simulation-based solvers. JMT requires a minimum number of samples of $5 \cdot 10^3$ samples.

- 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.

- timespan (real interval) requires the transient solver to produce a solution in the specified temporal range. If the value is set to $[\text{Inf}, \text{Inf}]$ the solver will only return a steady-state solution. For the FLD solver and in simulation, this setting has the same computational cost of $[0, \text{Inf}]$, 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, DES 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).

- mserbatch (integer ≥ 1, DES only) batch size for the MSER transient detection algorithm. Only used when tranfilter='mser5'. Default is 5 (standard MSER-5).

- warmupfrac (real $\in [0, 1)$, DES only) fraction of total events discarded as warmup when using fixed transient filtering. Only used when tranfilter='fixed'. Default is 0.2 (20% warmup).

- `obmoverlap` (real $\in [0,1]$, `DES` 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.

- `cnvgon` (`logical`, `DES` 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 $\in (0,1)$, `DES` only) convergence tolerance threshold. Simulation stops when (CI half-width/mean) < **cnvgtol** for all metrics. A value of 0.05 means 5% relative precision. Default is 0.05.

- `cnvgbatch` (integer ≥ 1, `DES` only) minimum number of batches required before checking for convergence. More batches provide more reliable confidence interval estimates. Default is 20.

- `cnvgchk` (integer ≥ 0, `DES` 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.

5.4   Solver maintenance

The following best practices can be helpful in maintaining the LINE installation:

- To install a new release of JMT, it is necessary to delete (or overwrite) the `JMT.jar` file under the `'JMT'` folder. This forces LINE to download the latest version of the JMT executable.

- To remove temporary by-products of the JMT solver it is recommended to periodically run the `jmtCleanTempDir` script. This is more important when using the `'keep'` option, which stores on disk the temporary `.jsimg` and `.jsimw` models sent to JMT.

Table 5.9: Default values of the LINE solver options and their default assignments (MATLAB)

Option	Solver default							
	MVA	CTMC	DES	FLD	JMT	MAM	NC	SSA
cache	true	true	true	true	true	true	true	true
confint			false	true	false			false
config								
cutoff		(no default)						
force	false	false	false	false	false	false	false	false
keep			false		false			
init_sol	[]			[]				
iter_max	10^3			200				
iter_tol	10^{-6}			10^{-4}		10^{-4}		
method	'default'	'default'	'default'	'default'	'default'	'default'	'default'	'default'
lang	'matlab'	'matlab'	'matlab'	'matlab'	'matlab'	'matlab'	'matlab'	'matlab'
samples			10^5		10^4			10^4
seed	rand	rand	rand	rand	rand	rand	rand	rand
stiff				true				
timespan		[Inf,Inf]	[0,Inf]	[0,Inf]	[0,Inf]		[Inf,Inf]	[0,Inf]
tol	10^{-4}	10^{-4}		10^{-4}				
verbose	1	1	1	1	1	1	1	1

Chapter 6

Layered network models

In this chapter, 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 [30] and to the LQNS user manual for an introduction [31]. 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.

6.1 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.

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.

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 either be *synchronous*, *asynchronous*, or

forwarding. At present, LINE supports only the first two kinds of activities. 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.

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 Network object, which are then iterative analyzed using different solution methods.

6.2   LayeredNetwork object definition

6.2.1   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', 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 instead 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 to 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 Network object.

6.2.2 FunctionTask

The `FunctionTask` class extends `Task` to model serverless functions and function-as-a-service (FaaS) systems. It adds two additional timing properties characteristic of serverless environments:

- Setup time (cold start): the initialization delay incurred when a function instance is started.

- Delay-off time (teardown): the delay incurred when a function instance is terminated.

A `FunctionTask` can be instantiated as follows:

```
F1 = FunctionTask(model, 'F1', 4, SchedStrategy.FCFS).on(P1);
F1.setSetupTime(Exp(1.0));      % cold start time (mean = 1.0)
F1.setDelayOffTime(Exp(2.0));   % teardown time (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. A complete example is provided in `lqn_function` in the `examples/` folder.

6.2.3 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 to 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_repl_lru` and `cache_compare_repl` in the `examples/` folder.

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

AND-fork and AND-join activity precedences play a central role in modeling parallel execution patterns within layered queueing networks, enabling representation of concurrent task execution and subsequent

synchronization. When an activity reaches an AND-fork precedence, it spawns multiple parallel execution paths that proceed independently until they converge at a corresponding AND-join. Each forked branch can contain arbitrary sequences of activities, including service demands on processors, synchronous calls to lower-layer entries, and even nested fork-join structures. The semantics of AND-fork in layered networks differ subtly from fork-join constructs in traditional queueing networks because activities are bound to tasks and processors, introducing resource contention at both the activity level and the processor level. Cross-layer synchronization occurs when activities in different branches make synchronous calls to entries located in separate layers of the model, requiring coordination between the completion of service at multiple servers before the join can proceed. The join point blocks until all branches spawned by the corresponding fork have completed their execution, at which point a single thread of control resumes and proceeds to subsequent activities. This blocking behavior at the join creates dependencies that propagate across layers, as the response time of a high-level task depends on the maximum completion time across all parallel branches, which in turn depends on the response times of lower-layer services invoked from those branches. Task completion detection in layered networks must account for the fact that a task finishes processing an entry request only after all activities in the entry's activity graph have executed, including waiting for all branches of any AND-fork to synchronize at their corresponding AND-join. Solvers analyze these fork-join structures by constructing auxiliary queueing network models where each branch between a fork and join is represented as a separate subnetwork, computing the distribution of completion times for each branch, and then determining the distribution of the maximum completion time which governs the join synchronization delay. The LN solver uses matrix-geometric methods and moment-matching techniques to approximate these maximum distributions, while the LQNS solver employs iterative methods that decompose the layered model into separate queueing networks for each layer and refine their parameterization until convergence. The `lqn_workflows` example demonstrates a realistic scenario with multiple AND-fork/AND-join pairs spanning several layers, illustrating how business process workflows with parallel task execution can be modeled and analyzed using the layered queueing network abstraction.

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.fitMeanAndOrder(1.0,2)).on(T2);
A22 = Activity(model, 'A22', Exp(1.0)).on(T2).repliesTo(E2);
T2.addPrecedence(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).

6.2.4 Debugging and visualization

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

```
plot(model)
```
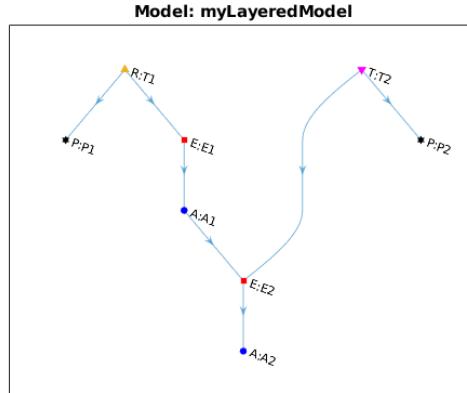
Figure 6.1: `LayeredNetwork.plot` method

An example of the result is shown in the next figure. The figure shows two processors (`P1` and `P2`), two tasks (`T1` and `T2`), and three entries (`E1`, `E2`, and `E3`) with their associated activities. Both dependencies and calls are both shown as directed arcs, with the edge weight on call arcs corresponding to the average number of calls to the target entry. For example, `A1` calls `E3` on average 2.0 times. In this figure, by clicking with the mouse on a node MATLAB will display, using a data tip, some relevant properties of the node such as scheduling or multiplicity.

When invoked as

```
plot(model,true)
```

The `jsimgView` and `jsimwView` methods can be used to visualize in JMT each layer. This can be done by first calling the `getLayers` method to obtain a cell array consisting of the Network objects, each one corresponding to a layer, and then invoking the `jsimgView` and `jsimwView` 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:

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 replyTo	Repeated synch calls
Activity with boundTo specification	Repeated asynch calls

6.3 Internals

6.3.1 Representation of the model structure

It is possible to access the internal representation of a `LayeredNetwork` model in a similar way as for Network objects, i.e.:

```
lqn = model.getStruct();
```

The return `lqn` structure, of class `LayeredNetworkStruct`, contains all the information about the specified model. It relies on relative and absolute indexing for the elements of the `LayeredNetwork`.

- A *relative* index is a number between 1 and the number of similar elements in the model, e.g., for a model with 3 tasks, the relative index t of a task would be a number in $[1,3]$.

- An *absolute* index is a number between 1 and the total number of elements (of any kind, except calls) in the model, e.g., for a model with 2 hosts, 3 tasks, 5 entries, and 8 activities, the total number of elements is `nidx`= 18 and last activity a may have an absolute index `aidx`= 18 and a relative index a= 8.

- The difference between the relative and the absolute index of an element is referred to as *shift*, e.g., in the previous example `ashift`= $18 - 8 = 10$.

- Absolute and relative indexing for calls and hosts are identical, call index `cidx` ranges in $[1, \mathbf{ncalls}]$ and host index `hidx` ranges in $[1, \mathbf{nhosts}]$.

Using the above convention, the internal representation of the model is described in Table 6.1. As in the examples above, relative and absolute indexes are differentiated by using the suffix `idx` in the latter (e.g., a vs. `aidx`). This indexing style is used throughout the codebase as well.

Table 6.1: `LayeredNetworkStruct` static properties

Field	Type	Description
nidx	integer	Total number of `LayeredNetwork` elements
nhosts	integer	Number of `Hosts` or `Processor` elements
ntasks	integer	Number of `Tasks` elements
nentries	integer	Number of `Entry` elements
nacts	integer	Number of `Activity` elements
ncalls	integer	Number of calls issued by `Activity` elements
hshift	integer	For host h, the value h+`hshift` returns its absolute index in 1...`nidx`
tshift	integer	For task t, the value t+`tshift` returns its absolute index in 1...`nidx`
eshift	integer	For entry e, the value e+`eshift` returns its absolute index in 1...`nidx`
ashift	integer	For activity a, the value a+`ashift` returns its absolute index in 1...`nidx`
cshift	integer	For call c, the value c+`cshift` returns its absolute index in 1...`ncalls`

`tasksof{`*hidx*`}`	`array`	Array of absolute indexes for tasks on host with absolute index *hidx*
`entriesof{`*tidx*`}`	`array`	Array of absolute indexes for entries on task with absolute index *tidx*
`actsof{`*eidx*`}`	`array`	Array of absolute indexes for activities reachable from entry with absolute index *eidx*
`actsof{`*tidx*`}`	`array`	Array of absolute indexes for activities on task with absolute index *tidx*
`callsof{`*aidx*`}`	`array`	Array of absolute indexes for calls on activity with absolute index *aidx*
`hostdem{`*idx*`}`	`object`	*(Deprecated)* Object for the host demand distribution for element *idx*
`hostdem_type(`*idx*`)`	`integer`	`ProcessType` id for host demand distribution for element *idx*
`hostdem_params{`*idx*`}`	`array`	Parameters array for host demand distribution for element *idx*
`hostdem_mean(`*idx*`)`	`double`	Mean of host demand distribution for element *idx*
`hostdem_scv(`*idx*`)`	`double`	Squared coefficient of variation for host demand distribution for element *idx*
`hostdem_proc{`*idx*`}`	`cell`	Process representation (PH/MAP matrices) for host demand for element *idx*
`think{`*idx*`}`	`object`	*(Deprecated)* Object for the think time distribution for element *idx*
`think_type(`*idx*`)`	`integer`	`ProcessType` id for think time distribution for element *idx*
`think_params{`*idx*`}`	`array`	Parameters array for think time distribution for element *idx*
`think_mean(`*idx*`)`	`double`	Mean of think time distribution for element *idx*
`think_scv(`*idx*`)`	`double`	Squared coefficient of variation for think time distribution for element *idx*
`think_proc{`*idx*`}`	`cell`	Process representation (PH/MAP matrices) for think time for element *idx*
`actthink{`*idx*`}`	`object`	*(Deprecated)* Object for the activity think time distribution for element *idx*
`actthink_type(`*idx*`)`	`integer`	`ProcessType` id for activity think time distribution for element *idx*
`actthink_params{`*idx*`}`	`array`	Parameters array for activity think time distribution for element *idx*
`actthink_mean(`*idx*`)`	`double`	Mean of activity think time distribution for element *idx*
`actthink_scv(`*idx*`)`	`double`	Squared coefficient of variation for activity think time distribution for element *idx*
`actthink_proc{`*idx*`}`	`cell`	Process representation (PH/MAP matrices) for activity think time for element *idx*
`sched(`*idx*`)`	`integer`	`SchedStrategy` scheduling policy id for a host or task with absolute index *idx*
`names{`*idx*`}`	`char`	Name of the element with absolute index *idx*
`hashnames{`*idx*`}`	`char`	Same as `names{`*idx*`}` with type prefix ("H:" for host, "R:" for reference task, "T:" for task, "C:" for cache task, "E:" for entry, "I:" for item entry, "A:" for activity)
`mult(`*idx*`)`	`integer`	Multiplicity for a host or task with absolute index *idx*
`repl(`*idx*`)`	`integer`	Replication factor for a host or task with absolute index *idx*
`type{`*idx*`}`	`integer`	`LayeredNetworkElement` id for an element with absolute index *idx*
`nitems(`*idx*`)`	`integer`	Total number of items in `CacheTask` or `ItemEntry` *idx*
`itemcap{`*idx*`}(`*l*`)`	`integer`	CacheTask *idx* capacity for cache list *l*
`replacement(`*tidx*`)`	`integer`	`ReplacementPolicy` id for cache task with absolute index *tidx*
`itemproc(`*idx*`)`	`object`	*(Deprecated)* `DiscreteDistrib` object for for the item popularity in `ItemEntry` *idx*
`itemproc_type(`*idx*`)`	`integer`	`ProcessType` id for item popularity distribution for element *idx*
`itemproc_params{`*idx*`}`	`array`	Parameters array for item popularity distribution for element *idx*
`itemproc_mean(`*idx*`)`	`double`	Mean of item popularity distribution for element *idx*
`itemproc_scv(`*idx*`)`	`double`	Squared coefficient of variation for item popularity distribution for element *idx*
`itemproc_proc{`*idx*`}`	`cell`	Process representation for item popularity for element *idx*
`calltype(`*cidx*`)`	`integer`	`CallType` id for call *cidx*
`callpair(`*cidx*`,`*j*`)`	`integer`	$j = 1$ is the activity *aidx* issuing call *cidx*, $j = 2$ is the entry *eidx* called by *cidx*

`callproc(`*cidx*`)`	object	*(Deprecated)* `DiscreteDistrib` object for the number of calls with absolute index *cidx*
`callproc_type(`*cidx*`)`	integer	`ProcessType` id for call count distribution for call *cidx*
`callproc_params{`*cidx*`}`	array	Parameters array for call count distribution for call *cidx*
`callproc_mean(`*cidx*`)`	double	Mean of call count distribution for call *cidx*
`callproc_scv(`*cidx*`)`	double	Squared coefficient of variation for call count distribution for call *cidx*
`callproc_proc{`*cidx*`}`	cell	Process representation for call count for call *cidx*
`callnames{`*cidx*`}`	char	Automatically assigned name to call with absolute index *cidx*
`callhashnames{`*cidx*`}`	char	Same as `callnames{`*cidx*`}` but with type prefix for source and destination
`actpretype(`*aidx*`)`	integer	`ActivityPrecedenceType` id preceding the activity with absolute index *aidx*
`actposttype(`*aidx*`)`	integer	`ActivityPrecedenceType` id after the activity with absolute index *aidx*
`graph(`idx_i, idx_j`)`	float	$\neq 0$ if idx_i "runs on", "calls" or "precedes" idx_j. Values are by default 1.0 unless they specify a mean number of calls, a probability, fork fan-out or join fan-in degrees
`dag(`idx_i, idx_j`)`	float	Similar to `graph(`idx_i, idx_j`)` but loops do not form cycles and entry-task edges are flipped direction at non-reference tasks to aid multiplicity analysis.
`parent(`*idx*`)`	integer	Return absolute index value of the parent node for the node whose absolute index value is *idx*
`replygraph(`*aidx, eidx*`)`	logical	True if activity with absolute index *aidx* on entry *eidx* replies ending the entry call
`taskgraph(`$tidx_i, tidx_j$`)`	logical	True if host or task idx_i is a caller of host or task idx_j
`iscache(`*tidx*`)`	logical	True if task *tidx* is a `CacheTask`
`iscaller(`idx_i, idx_j`)`	logical	True if task or activity idx_i calls task or entry idx_j
`issynccaller(`idx_i, idx_j`)`	logical	True if task or activity idx_i issues a synchronous call to task or entry idx_j
`isasynccaller(`idx_i, idx_j`)`	logical	True if task or activity idx_i issues an asynchronous call to task or entry idx_j
`isref(`*tidx*`)`	logical	True if task *tidx* is a reference task
`isfunction(`*tidx*`)`	logical	True if task *tidx* is a function task
`setuptime{`*idx*`}`	object	*(Deprecated)* Object for the setup time distribution for element *idx*
`setuptime_type(`*idx*`)`	integer	`ProcessType` id for setup time distribution for element *idx*
`setuptime_params{`*idx*`}`	array	Parameters array for setup time distribution for element *idx*
`setuptime_mean(`*idx*`)`	double	Mean of setup time distribution for element *idx*
`setuptime_scv(`*idx*`)`	double	Squared coefficient of variation for setup time distribution for element *idx*
`setuptime_proc{`*idx*`}`	cell	Process representation (PH/MAP matrices) for setup time for element *idx*
`delayofftime{`*idx*`}`	object	*(Deprecated)* Object for the delay-off time distribution for element *idx*
`delayofftime_type(`*idx*`)`	integer	`ProcessType` id for delay-off time distribution for element *idx*
`delayofftime_params{`*idx*`}`	array	Parameters array for delay-off time distribution for element *idx*
`delayofftime_mean(`*idx*`)`	double	Mean of delay-off time distribution for element *idx*
`delayofftime_scv(`*idx*`)`	double	Squared coefficient of variation for delay-off time distribution for element *idx*
`delayofftime_proc{`*idx*`}`	cell	Process representation (PH/MAP matrices) for delay-off time for element *idx*
`arrival{`*eidx*`}`	object	*(Deprecated)* Object for open arrival distribution for entry *eidx*
`arrival_type(`*eidx*`)`	integer	`ProcessType` id for arrival distribution for entry *eidx*
`arrival_params{`*eidx*`}`	array	Parameters array for arrival distribution for entry *eidx*
`arrival_mean(`*eidx*`)`	double	Mean of arrival distribution for entry *eidx*
`arrival_scv(`*eidx*`)`	double	Squared coefficient of variation for arrival distribution for entry *eidx*
`arrival_proc{`*eidx*`}`	cell	Process representation (PH/MAP matrices) for arrival for entry *eidx*

6.3.2 Decomposition into layers

Layers are a form of decomposition where we model the performance of one or more servers. The activity of clients not detailed in that layer is taken into account through an artificial delay station, placed in a closed loop to the servers [56]. This artificial delay is used to model the inter-arrival time between calls issued by that client.

The current version of LINE adopts SRVN-type layering [31], whereby a layer corresponds to one and only one resource, either a processor or a task. The `getLayers` method returns a cell array consisting of the Network objects corresponding to each layer

```
layers = model.getLayers()
```

The decomposition is performed through the `LN` solver described later.

Within each layer, classes are used to model the time a job spends in a given activity or call, with synchronous calls being modeled by classed with label including an arrow, e.g., `'AS1=>E3'` is a closed class used represent synchronous calls from activity `AS1` to entry `E3`, whereas `'AS1->E3'` denotes an asynchronous call. Artificial delays and reference nodes are modelled as a delay station named `'Clients'`, whereas the task or processor assigned to the layer is modelled as the other node in the layer.

6.4 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 [31]. The latter requires a distribution of LQNS to be available on the operating system command line.

The LQNS solver is a mature external tool developed by the Real-time and Distributed Systems Group at Carleton University that implements a variety of solution methods for layered queueing networks. When invoked from LINE, the LQNS wrapper generates an XML input file conforming to the LQN model specification, spawns the external LQNS executable as a subprocess, and parses the XML output to extract performance metrics. The LQNS solver supports multiple solution methods selectable through solver options, including exact analytical methods for models that satisfy product-form conditions, iterative approximation methods based on mean value analysis for general layered networks, and simulation-based approaches for models with features not amenable to analytical solution. Method selection in LQNS is controlled by command-line flags passed to the external solver, with common options including the convergence tolerance for iterative methods, the maximum number of iterations, and the choice between exact Markov chain analysis versus approximate decomposition. The comparison between the LN solver (native to LINE) and LQNS reveals different strengths and trade-offs. The LN solver converts the layered network into a collection of ordinary queueing networks with artificial delay stations representing cross-layer synchronization, then employs MVA or other queueing network solvers iteratively until the delays converge to consistent values. This approach benefits from the full range of queueing network solution methods available in LINE and provides tighter integration with other LINE features such as optimization and sensitivity analysis. In contrast, LQNS

uses specialized layered network algorithms that directly exploit the hierarchical structure without intermediate conversion, potentially achieving faster convergence for deeply nested models or those with complex precedence relationships. LQNS is preferred over the LN solver when analyzing very large layered models where the conversion to queueing networks would create excessive numbers of artificial classes or when using specialized LQNS features such as quorum joins or non-exponential think times that are not yet fully supported in the LN conversion process. Conversely, the LN solver with MVA backend is preferable when the model is relatively small, when exact product-form results are desired, or when integration with LINE workflows such as automated parameter sweeps or ensemble analysis is important. Both solvers produce comparable results for standard layered queueing network models, with discrepancies typically arising only in corner cases involving unusual combinations of scheduling disciplines, replication, or activity precedence patterns.

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

```
>> AvgTable = LQNS(model).avgTable
AvgTable =
  8x6 table
    Node       NodeType        QLen         Util        RespT       Tput

    'P1'      'Processor'       NaN       0.071429      NaN          NaN
    'T1'      'Task'          0.28571     0.071429      NaN        0.071429
    'E1'      'Entry'         0.28571     0.071429       4         0.071429
    'A1'      'Activity'      0.28571     0.071429       4         0.071429
    'P2'      'Processor'       NaN       0.21429       NaN          NaN
    'T2'      'Task'          0.21429     0.21429       NaN        0.21429
    'E2'      'Entry'         0.21429     0.21429        1         0.21429
    'A2'      'Activity'      0.21429     0.21429        1         0.21429
```

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 Network 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.

6.4.1 `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, 0.95`).

Upon invocation, the `lqns` or `lqsim` commands will be searched for in the system path. If they are unavailable, the termination of LQNS will interrupt.

Remote Docker 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 Docker container can be started using the `imperialqore/lqns-rest` image extended with the REST API. 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.

6.4.2 `LN`

The native `LN` solver iteratively applies the layer updates until convergence of the steady-state measures. Since updates are parametric on the solution of each layer, `LN` can apply any of the Network solvers described in the solvers chapter to the analysis of individual layers, as illustrated in the following example for the `MVA` solver

```
options = LN.defaultOptions;
mvaopt = MVA.defaultOptions;
LN(model, @(layer) MVA(layer, mvaopt), options).avgTable
```

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`.

Methods supported by the LN solver include:

- `options.method='default'`: default recursive solution based on mean values

- `options.method='moment3'`: solution by recursive 3-moment approximation of response time distributions.

- `options.method='mol'`: Method of Layers iteration with nested inner/outer loops for task and host layers.

6.5 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.parseXML(filename)
```

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

```
model.writeXML(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 in the legacy SRVN file format[2] by means of the `writeSRVN(filename)` function.

---

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

[2] http://www.sce.carleton.ca/rads/lqns/lqn-documentation/format.pdf

6.5.1   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.

```
mergedModel = Ensemble.merge({model1, model2});
```

Chapter 7

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. In first approximation, one may simply solve independent models for each stage and weight their solution. However, depending on the frequency of transitions between stages, this incur significant errors [19]. LINE automates the evaluation of such scenarios, applying the appropriate corrections based on the environment and system characteristics.

7.1 Environment definition

7.1.1 Specifying the environment

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

```
E = 2;
envModel = Environment('UnreliableEnv', E);
```

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

```
envModel.addStage('Online', 'UP', network1);
envModel.addStage('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

```
envModel.addTransition('Online', 'Offline', Exp(1));
envModel.addTransition('Offline', 'Online', Exp(2));
```

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

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

```
>> envModel.getStageTable
ans =
  2x6 table
    Stage      Name        Type       Prob        HoldT            Model
    _____    _____    _____    _____    _____    _____

      1      'Online'     'Up'       0.83333     {1x4 cell}    [1x1 Network]
      2      'Offline'    'Down'     0.16667     {1x4 cell}    [1x1 Network]
```

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 to that stage.

7.1.2   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 Network 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.

7.1.3   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.,

```
resetRule = @(QExit) [sum(QExit,1); zeros(size(QExit,1)-1,size(QExit,2))]; % ...
  move all jobs into station 1, without changing their classes
envModel.add_transition('Online', 'Offline', Exp(1), resetRule);
```

In the above code, `QExit(i,r)` is the queue-length of class-r jobs observed at node i upon exiting the online state. The `resetRule` must produce in output a vector of the same size of `QExit`. The reset policy in this example moves instantaneously all jobs in the network into station 1 upon entering into the offline state. Note that `resetRule` can be configured differently with each stage transition and the default value is simply `resetRule = @(QExit) QExit`.

7.2 Solvers

The steady-state analysis of a system in a random environment is carried out in LINE using the blending method [19], 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.

7.2.1 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.

As in the transient analysis of Network 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.,

```
envSolver = ENV(envModel, @FLD, options);
[QN,UN,TN] = envSolver.avg()
```

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.

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 [25]. It partitions the environment state space into macro-states and computes approximate steady-state probabilities.

- *Koury-McAllister-Stewart* ('kms'): An iterative aggregation-disaggregation method [40] that refines the Courtois solution through successive approximations.

- *Takahashi's method* ('takahashi'): Another iterative aggregation-disaggregation approach [62] with different convergence properties than KMS.

- *Multigrid* ('multi'): A multi-level decomposition method that applies hierarchical aggregation.

For iterative methods ('kms' and 'takahashi'), the number of iterations can be controlled using options.config.da_iter (default: 10).
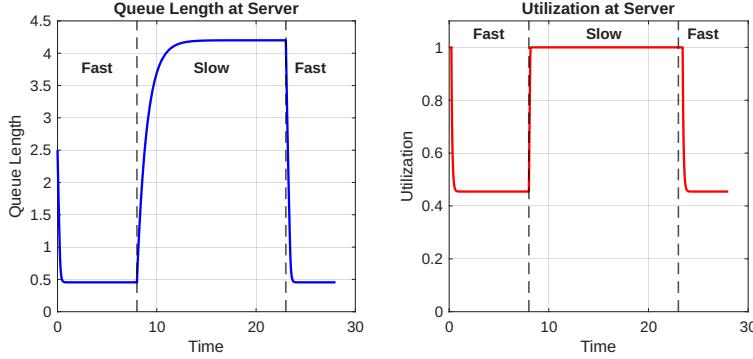
Figure 7.1: Transient queue length and utilization evolution along a sample path.

The configuration options are summarized in Table 7.2.1.

Table 7.1: ENV configuration options (MATLAB)

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	integer	Number of iterations for iterative methods (default: 10).
options.config.env_alpha	double	Alpha parameter for macro-state search (default: 0.01).

7.3   Examples

7.3.1   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:

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

% Closed class with 5 jobs
```

```
N = 5;
jobclass = ClosedClass(baseModel, '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
baseModel.link(Network.serialRouting(delay, queue));
```

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

```
env = Environment('ServerModes');

% Stage 1: Fast mode (service rate = 10.0, low utilization)
fastModel = baseModel.copy();
fastQueue = fastModel.getNodeByName('Fast/Slow Server');
fastQueue.setService(fastModel.classes{1}, Exp(10.0));
env.addStage('Fast', 'operational', fastModel);

% Stage 2: Slow mode (service rate = 0.8, high utilization)
slowModel = baseModel.copy();
slowQueue = slowModel.getNodeByName('Fast/Slow Server');
slowQueue.setService(slowModel.classes{1}, Exp(0.8));
env.addStage('Slow', 'degraded', slowModel);

% Define transitions between stages
env.addTransition('Fast', 'Slow', Exp(0.5));  % Fast->Slow
env.addTransition('Slow', 'Fast', Exp(1.0));  % Slow->Fast
```

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

```
% Initialize the environment
env.init();
stageTable = env.getStageTable()

% Solve using SolverENV with Fluid sub-solver
fldOptions = FLD.defaultOptions;
fldOptions.timespan = [0, 100];
solverFactory = @(m) FLD(m, fldOptions);

envSolver = ENV(env, solverFactory);
[QN, UN, TN] = envSolver.getAvg();

% Display environment-averaged results
envAvgTable = envSolver.getAvgTable()
```

The output shows the environment stages and performance metrics:

```
stageTable =
```

```
     2x6 table
        Stage     Name       Type          Prob         HoldT          Model

        1        Fast      operational    0.66667     {1x4 cell}    {1x1 Network}
        2        Slow      degraded       0.33333     {1x4 cell}    {1x1 Network}

envAvgTable =
   2x6 table
        Station     JobClass     QLen      Util      RespT      Tput

        ThinkTime     Jobs      2.9961    2.9961        1      2.9961
        Server        Jobs      2.0039        1      0.66796       3
```

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, `SolverENV` 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 7.1, 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 ($\approx$45%), while in the Slow stage (service rate = 0.8), the server becomes saturated ($\approx$100% utilization), causing queue buildup as arrivals accumulate faster than they can be processed.

```matlab
% Define a sample path: Fast for 8 time units, Slow for 15, Fast for 5
samplePath = {'Fast', 8.0; 'Slow', 15.0; 'Fast', 5.0};

% Compute transient metrics along the sample path
[T, segmentResults] = solver.getSamplePathTable(samplePath);

% Display the results table
```

```
disp(T);

% Access detailed transient trajectories for plotting
seg1_QLen = segmentResults{1}.QNt{2,1}.metric;  % Server queue length in ...
segment 1
seg1_time = segmentResults{1}.QNt{2,1}.t;
```

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.

7.3.2 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');
env.addNodeFailureRepair(baseModel, '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 $\rightarrow$ DOWN) with the given breakdown distribution

- Adds a repair transition (DOWN $\rightarrow$ UP) with the given repair distribution

Alternative methods for more control include `addNodeBreakdown` and `addNodeRepair`, which can be called separately. Custom reset policies can be specified as optional parameters:

```
resetBreakdown = @(q) 0*q;  % Clear queues on breakdown
resetRepair = @(q) q;       % Keep jobs on repair
env.addNodeFailureRepair(baseModel, 'Server1', Exp(0.1), Exp(1.0), ...
    Exp(0.5), resetBreakdown, resetRepair);
```

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 `getReliabilityTable` method (aliases: `relT`, `getRelT`, `relTable`, `getRelTable`):

```
>> env.getReliabilityTable
ans =
  4x4 table
     Metric          Value           Unit                    Description
    _____    _____   _____    _____

    MTTF          10               time units     Mean time to failure (UP -> DOWN)
    MTTR          1                time units     Mean time to repair (DOWN -> UP)
    MTBF          11               time units     Mean time between failures (MTTF + MTTR)
    Availability  0.90909          probability    Steady-state probability of UP state
```

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.

7.3.3   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:

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

% Closed class with 3 jobs
N = 3;
jobClass = 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.setService(jobClass, MMPP2(2.0, 5.0, 0.5, 0.3));
queue2.setService(jobClass, Exp(3.0));
```

```
model.link(Network.serialRouting(queue1, queue2));

% Convert to random environment
env = MAPQN2RENV(model);
env.printStageTable();
```

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

7.3.4   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.

```
% Create two network models for different modes
model1 = Network('Mode1');
source1 = Source(model1, 'Source');
queue1 = Queue(model1, 'Queue1', SchedStrategy.PS);
sink1 = Sink(model1, 'Sink');
jobClass1 = OpenClass(model1, 'Jobs');
source1.setArrival(jobClass1, Exp(1.0));
queue1.setService(jobClass1, Exp(2.0));
model1.link(Network.serialRouting(source1, queue1, sink1));

model2 = Network('Mode2');
source2 = Source(model2, 'Source');
queue2 = Queue(model2, 'Queue1', SchedStrategy.PS);
sink2 = Sink(model2, 'Sink');
jobClass2 = OpenClass(model2, 'Jobs');
source2.setArrival(jobClass2, Exp(1.0));
queue2.setService(jobClass2, Exp(0.8));
model2.link(Network.serialRouting(source2, queue2, sink2));

% Create environment with LogNormal transitions (non-Markovian)
envModel = Environment('SemiMarkovEnv');
envModel.addStage('Stage1', 'UP', model1);
envModel.addStage('Stage2', 'UP', model2);
envModel.addTransition('Stage1', 'Stage2', Lognormal(1.0, 0.5));  % Non-exponential
envModel.addTransition('Stage2', 'Stage1', Lognormal(0.5, 0.3));  % Non-exponential

% Enable SMP method for non-Markovian transitions
options = SolverOptions();
options.method = 'smp';
solver = SolverENV(envModel, @SolverFluid, options);
solver.runAnalyzer();
```

```
solver.result
```

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.

Appendix A

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.

A.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
```

A.2   Basic Usage

The CLI supports several commands for solving models and querying information.

A.2.1   Solving a Model

To solve a queueing network model, use the `solve` command with the mandatory `-s` (or `-solver`) option to specify the solver algorithm:

```
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
ctmc	CTMC	jsim/jsimg/jsimw	Exact, small models only
des	Discrete Event Simulation	jsim/jsimg/jsimw	Simulation-based
fld	Fluid ODE	jsim/jsimg/jsimw	Fast, approximate
jmt	Java Modelling Tools	jsim/jsimg/jsimw	External simulation
ln	Layered Network	lqnx/xml	For LQN models
lqns	LQN Solver	lqnx/xml	External LQNS tool
mam	Matrix Analytic	jsim/jsimg/jsimw	Supports Fork-Join
mva	Mean Value Analysis	jsim/jsimg/jsimw	Fast, general purpose
nc	Normalizing Constant	jsim/jsimg/jsimw	Exact analysis
ssa	Stochastic Simulation	jsim/jsimg/jsimw	State-space sampling

The input format is automatically detected from the file extension. Supported formats include:

- `.jsimg`, `.jsim`, `.jsimw`: JMT simulation model formats

- `.lqnx`, `.xml`: Layered Queueing Network Solver XML format

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
```

A.2.2 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:

- `cdf-respt`, `cdf-passt`: Response/passage time CDFs

- `perct-respt`: Response time percentiles (MAM solver)

- `prob`, `prob-aggr`, `prob-marg`: State probabilities (CTMC/SSA)

- `sample`, `sample-sys`: State trajectory sampling (SSA)

- `reward`, `reward-steady`: Reward metrics (CTMC)

To list all analysis types:

```
python line-cli.py list analysis
```

A.3 Server Modes

The CLI can run as a server to accept solve requests over the network.

A.3.1 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.

A.3.2   REST API Server

Start a REST API server for HTTP-based integration:

```
python line-cli.py rest -p 8080
```

Available endpoints:

- `POST /solve`: Submit a model for solving

- `GET /health`: Health check endpoint

A.4   Advanced Options

A.4.1   Stochastic Solver Options

For stochastic solvers (DES, SSA, JMT), a random seed can be specified for reproducibility:

```
python line-cli.py solve model.jsimg -s ssa -d 12345
```

A.4.2   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
```

A.4.3   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

A.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 REST API 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
`-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

A.6 Examples

A.6.1 Sample Output

Running the CLI on a sample open queueing network model:

```
python line-cli.py solve example.jsimg -s mva
```

produces the following output:

```
Solving example.jsimg...

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  Tput    2
Source 1  Class B  QLen    0
Source 1  Class B  Tput    1
Queue 1   Class A  QLen    1.33318
Queue 1   Class A  Util    0.4
Queue 1   Class A  RespT   0.66659
Queue 1   Class A  Tput    2
Queue 1   Class B  QLen    0.99989
Queue 1   Class B  Util    0.3
Queue 1   Class B  RespT   0.99989
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  Tput    3


System Metrics
==============
Station  Class                      Metric    Value    Unit
-----------------------------------------------------------
Chain1   Class A Class B Class C  SysRespT  0.77769
Chain1   Class A Class B Class C  SysTput   3

Execution time: 0.252s
```

Appendix B

API Reference - jline.api Package

This chapter provides comprehensive documentation for the `jline.api` package, which contains high-level algorithmic implementations organized by computational area. The API layer provides specialized algorithms for various performance modeling and analysis tasks.

B.1 Package Overview

The `jline.api` package is organized into the following packages:

- **CACHE** - Caching algorithms and cache performance analysis

- **FJ** - Fork-Join queueing utilities

- **LOSSN** - Loss network algorithms

- **LSN** - Load-dependent stochastic network algorithms

- **MAM** - Matrix Analytic Methods for stochastic processes

- **MAPQN** - Markovian Arrival Process Queueing Networks

- **MC** - Markov Chain algorithms (CTMC/DTMC)

- **MEASURES** - Information theory and probability distance measures

- **NPFQN** - Non-Product-Form Queueing Networks

- **PFQN** - Product-Form Queueing Networks

- **POLLING** - Polling system algorithms

- **QSYS** - Single queue system analysis

- **SN** - Stochastic Network utility functions

- **TRACE** - Trace analysis and fitting algorithms

- **WKFLOW** - Workflow analysis algorithms

B.2   Cache Analysis (jline.api.cache)

This package provides algorithms for analyzing cache performance, replacement policies, and cache hierarchies.

B.2.1   Key Algorithms

- `Cache_miss_*` - Cache miss probability calculations for various replacement policies

- `Cache_prob_*` - Cache hit/miss probability analysis

- `Cache_ttl_*` - Time-to-live cache analysis algorithms

- `Cache_rayint` - Ray integration methods for cache analysis

- `Cache_mva` - Mean value analysis for cache systems

B.2.2   Example Usage

```
// Example: Calculate cache miss probability for LRU policy
Matrix hitProb = Cache_miss_rayint.cache_miss_rayint(
    arrival_rates,
    cache_capacity,
    item_popularities
);
```

B.3   Matrix Analytic Methods (jline.api.mam)

This package implements matrix analytic methods for analyzing stochastic processes, particularly Markovian arrival processes (MAPs) and phase-type distributions.

B.3.1 Key Algorithm Categories

MAP Algorithms

- `Map_*` - Single MAP analysis (moments, distributions, transformations)

- `Mmap_*` - Marked MAP analysis for multi-class arrivals

- `Mmpp_*` - Markov Modulated Poisson Process algorithms

Phase-Type Distributions

- `Aph_*` - Acyclic phase-type distribution fitting and analysis

- `Aph2_*` - Order-2 acyclic phase-type specific algorithms

Fitting Algorithms

- `*_fit_*` - Various fitting algorithms for different arrival processes

- `*_fit_trace` - Trace-based fitting methods

- `*_fit_gamma` - Gamma distribution fitting

B.3.2 Example Usage

```
// Example: Fit a MAP to trace data
Matrix trace = loadTraceData();
Matrix[] mapParams = Map2_fit.map2_fit(trace, options);
Matrix D0 = mapParams[0]; // Background rates
Matrix D1 = mapParams[1]; // Arrival rates
```

B.4 Markov Chain Analysis (jline.api.mc)

This package provides algorithms for continuous-time (CTMC) and discrete-time (DTMC) Markov chain analysis.

B.4.1 CTMC Algorithms

- `Ctmc_solve` - Steady-state solution methods

- `Ctmc_transient` - Transient analysis algorithms

- `Ctmc_randomization` - Uniformization methods

- `Ctmc_stochcomp` - Stochastic complementation

- `Ctmc_simulate` - Discrete-event simulation

B.4.2 DTMC Algorithms

- `Dtmc_solve` - Eigenvalue-based solution methods

- `Dtmc_simulate` - Markov chain simulation

- `Dtmc_stochcomp` - State space reduction techniques

B.4.3 Example Usage

```
// Example: Solve CTMC for steady-state probabilities
Matrix infinitesimalGenerator = buildCTMC();
Matrix steadyState = Ctmc_solve.ctmc_solve(infinitesimalGenerator);
```

B.5   Product-Form Queueing Networks (jline.api.pfqn)

This package implements algorithms for product-form queueing networks, organized into three main solution approaches.

B.5.1   Mean Value Analysis (pfqn.mva)

- `Pfqn_mva` - Exact mean value analysis

- `Pfqn_mvams` - Multi-server MVA

- `Pfqn_mvamx` - Mixed network MVA

- `Pfqn_linearizer*` - Various linearization approximations

B.5.2   Normalizing Constant (pfqn.nc)

- `Pfqn_nc` - Convolution algorithm

- `Pfqn_le` - Linear equation methods

- `Pfqn_ca` - Conditional arrival methods

- `Pfqn_mom` - Method of moments

B.5.3 Load-Dependent Services (pfqn.ld)

- `Pfqn_gld` - General load-dependent services

- `Pfqn_mvald` - MVA for load-dependent networks

- `Pfqn_ncld` - Normalizing constant for load-dependent services

B.6 Queueing System Analysis (jline.api.qsys)

This package provides algorithms for analyzing single queueing systems with various arrival and service processes.

B.6.1 Single Server Systems

- `Qsys_mm1` - M/M/1 queue analysis

- `Qsys_mg1` - M/G/1 queue analysis

- `Qsys_gm1` - G/M/1 queue analysis

- `Qsys_gg1` - G/G/1 queue approximations

B.6.2 Multi-Server Systems

- `Qsys_mmk` - M/M/k queue analysis

- `Qsys_gigk_*` - G/G/k approximation methods

B.6.3 Approximation Methods

- `Qsys_gig1_approx_*` - Various G/G/1 approximations (Allen-Cunneen, Gelenbe, Heyman, etc.)

- `Qsys_gigk_approx_*` - Multi-server approximations (Cosmetatos, Kingman, Whitt)

B.7 Stochastic Network Utilities (jline.api.sn)

This package provides utility functions for analyzing stochastic networks and determining model properties.

B.7.1 Network Property Detection

- `SnHas*` - Functions to detect network features (multiclass, product-form, etc.)

- `SnIs*` - Functions to determine network type (open, closed, mixed)

B.7.2   Performance Metrics

- `SnGet*` - Functions to extract performance metrics from solutions

- `SnRefresh*` - Functions to update network parameters

B.8   Information Theory and Probability Distance Measures (jline.api.measures)

This package provides a comprehensive collection of statistical measures for comparing probability distributions and analyzing information content. The measures are widely used in model fitting, validation, and comparison tasks throughout LINE.

B.8.1   Information-Theoretic Measures

The package implements standard information-theoretic quantities for discrete and continuous probability distributions. Shannon entropy (`Ms_entropy`) quantifies the uncertainty in a probability distribution and serves as a foundation for other measures. Conditional entropy (`Ms_condentropy`) measures the remaining uncertainty in one random variable given knowledge of another, while joint entropy (`Ms_jointentropy`) quantifies the total uncertainty in multiple random variables considered together. Mutual information (`Ms_mutinfo`) measures the amount of information shared between two random variables, representing the reduction in uncertainty about one variable when the other is observed.

B.8.2   Divergence Measures

Several algorithms compute divergence between probability distributions, quantifying how one distribution differs from another. The Kullback-Leibler divergence (`Ms_kullbackleibler`) is an asymmetric measure of the information loss when one distribution is used to approximate another. The Jensen-Shannon divergence (`Ms_jensenshannon`) provides a symmetric and bounded variant of KL divergence, useful for clustering and classification tasks. Additional divergence measures include relative entropy (`Ms_relatentropy`) and various chi-squared divergences (`Ms_pearsonchisquared`, `Ms_neymanchisquared`, `Ms_additivesymmetricchisquared`).

B.8.3   Statistical Distance Metrics

The package includes traditional statistical distance measures for comparing distributions. The Hellinger distance (`Ms_hellinger`) provides a metric that is symmetric and bounded, making it suitable for optimization problems. The Bhattacharyya distance (`Ms_bhattacharyya`) measures the similarity between two probability distributions and is closely related to the Hellinger distance. The Wasserstein distance (`Ms_wasserstein`) measures the minimum cost of transforming one distribution into another and is particularly useful for comparing empirical distributions.

B.8.4 Geometric Distance Measures

Standard geometric distances are provided for vector-based distribution comparisons. Euclidean distance (`Ms_euclidean`) computes the straight-line distance between distribution vectors, while Manhattan distance (`Ms_manhattan`) sums the absolute differences. The Minkowski distance (`Ms_minkowski`) generalizes both measures through a parameterizable norm. Other geometric measures include Lorentzian distance (`Ms_lorentzian`) and various wave-based distances (`Ms_wavehegdes`).

B.8.5 Similarity Measures

Several algorithms compute similarity rather than distance between distributions. Cosine similarity (`Ms_cosine`) measures the angle between distribution vectors, independent of magnitude. Jaccard similarity (`Ms_jaccard`) and Tanimoto similarity (`Ms_tanimoto`) quantify overlap between sets or distributions. Additional similarity measures include Dice coefficient (`Ms_sorensen`), Czekanowski similarity (`Ms_czekanowski`), and various intersection-based measures (`Ms_intersection`, `Ms_harmonicmean`).

B.8.6 Goodness-of-Fit Statistics

The package provides statistical tests for distribution comparison and model validation. The Anderson-Darling statistic (`Ms_anderson_darling`) tests whether a sample comes from a specified distribution, with particular sensitivity to differences in the distribution tails. Additional goodness-of-fit measures support model selection and validation tasks across LINE solvers.

B.9 Trace Analysis (jline.api.trace)

This package provides algorithms for analyzing and fitting empirical data traces.

B.9.1 Trace Statistics

- `Mtrace_*` - Multivariate trace analysis

- `Trace_mean`, `Trace_var` - Basic trace statistics

- `Mtrace_moment` - Higher-order moment analysis

B.9.2 Trace Manipulation

- `Mtrace_split`, `Mtrace_merge` - Trace partitioning and combination

- `Mtrace_bootstrap` - Bootstrap resampling methods

B.10   Workflow Analysis (jline.api.wf)

This package provides algorithms for analyzing workflow patterns and business process models.

B.10.1   Pattern Detection

- `Wf_branch_detector` - Branch pattern identification

- `Wf_loop_detector` - Loop pattern detection

- `Wf_parallel_detector` - Parallel execution detection

- `Wf_sequence_detector` - Sequential pattern analysis

B.10.2   Workflow Management

- `WorkflowManager` - Central workflow coordination

- `Wf_analyzer` - Comprehensive workflow analysis

- `Wf_pattern_updater` - Dynamic pattern modification

B.11   Fork-Join Utilities (jline.api.fj)

This package provides specialized utilities for analyzing fork-join queueing models, where arriving jobs are split into parallel tasks that must all complete before the job exits the system. Fork-join models are fundamental for analyzing parallel processing systems, MapReduce frameworks, and distributed computing architectures.

The topology detection function `Fj_isfj` validates whether a queueing network has the canonical fork-join structure consisting of a Source node, Fork node, multiple parallel Queue stations, Join node, and Sink node. This validation ensures that subsequent analysis algorithms can safely assume the required structural properties. The distribution conversion function `Fj_dist2fj` transforms service time distributions from standard LINE representations into the specialized format required by the FJ_codes library, which computes response time percentiles for fork-join systems. Parameter extraction is handled by `Fj_extract_params`, which analyzes a fork-join network model and extracts the key parameters including arrival rates, service time distributions for each parallel queue, and the number of parallel branches. These utilities enable seamless integration between LINE network models and specialized fork-join analysis algorithms that provide percentile-based performance predictions beyond mean value estimates.

B.12 Loss Networks (jline.api.lossn)

This package provides algorithms for analyzing loss networks, which are finite-capacity systems where arriving jobs that find insufficient resources are immediately rejected rather than queued. Loss networks are fundamental models for circuit-switched telecommunications systems, bandwidth allocation in communication networks, and admission control in cloud computing environments.

The primary algorithm `Lossn_erlangfp` implements an Erlang fixed-point approximation for multi-resource loss networks. This method computes blocking probabilities and utilization metrics for networks where each job class may require resources from multiple links simultaneously. The algorithm accepts arrival rates for each job class, resource capacity requirements specified as a matrix mapping links to classes, and available capacity at each link. It returns mean queue lengths per class, loss probabilities per class, and blocking probabilities per link. The fixed-point iteration approach provides efficient computation even for large-scale networks where exact solution methods become computationally prohibitive. This package enables capacity planning studies where system designers must dimension network resources to meet target blocking probability constraints under specified traffic loads.

B.13 Load-Dependent Stochastic Networks (jline.api.lsn)

This package provides algorithms for analyzing layered stochastic networks, which model hierarchical systems with multiple layers of service interactions. Layered stochastic networks extend traditional queueing network models by capturing inter-layer dependencies and resource contention across architectural tiers.

The key function `LsnMaxMultiplicity` computes the maximum multiplicity values for tasks and processors in layered network models. Multiplicity represents the number of concurrent instances of a software task or hardware processor, and determining appropriate multiplicity values is critical for capacity planning in multi-tier service systems. The algorithm analyzes task interactions, processor allocations, and service demands to compute multiplicity bounds that ensure system stability and meet performance targets. This capability supports architectural design studies where engineers must determine appropriate parallelism levels and resource allocations to satisfy response time requirements while minimizing infrastructure costs. The layered network abstraction is particularly valuable for modeling modern cloud applications with microservice architectures, where services span multiple deployment tiers with complex calling patterns and resource sharing.

B.14 MAP Queueing Networks (jline.api.mapqn)

This package implements algorithms for computing performance bounds in queueing networks with Markovian Arrival Process (MAP) arrivals. MAP-based queueing networks extend classical product-form networks by supporting bursty and correlated arrival patterns that commonly arise in computer systems, communication networks, and transaction processing workloads. Since MAP queueing networks generally lack

product-form solutions, this package focuses on efficient bounding techniques that provide rigorous upper and lower bounds on performance metrics.

The package implements several complementary bounding approaches organized around linear reduction methods. The general linear reduction algorithm `Mapqn_bnd_lr` formulates the bounding problem as a linear program that maximizes or minimizes target performance metrics subject to flow balance and traffic constraints. Variants include `Mapqn_bnd_lr_mva` which integrates mean value analysis approximations, and `Mapqn_bnd_lr_pf` which exploits product-form subnetworks when present. Alternative formulations based on queue-response decomposition are provided by `Mapqn_bnd_qr`, `Mapqn_bnd_qr_delay`, and `Mapqn_bnd_qr_ld` for load-dependent services.

Supporting infrastructure includes `Mapqn_parameters` and `Mapqn_parameters_factory` for constructing and validating the parameter structures required by bounding algorithms, and `Mapqn_lpmodel` which provides the linear programming framework used across multiple bound computation methods. Additional algorithms `Mapqn_qr_bounds_bas` and `Mapqn_qr_bounds_rsrd` implement specialized queue-response bounds using different decomposition strategies. These bounding techniques enable rapid performance prediction for MAP queueing networks where exact solution methods are computationally infeasible, supporting capacity planning and performance debugging in systems with realistic correlated workloads.

B.15   Polling Systems (jline.api.polling)

This package provides exact analytical algorithms for computing mean waiting times in cyclic polling systems with switchover times. A polling system consists of a single server that visits N queues in cyclic order, spending switchover time to move between queues. Three polling disciplines are supported, each differing in how many customers are served per visit:

- **Exhaustive** (`polling_qsys_exhaustive`): The server continues serving a queue until it becomes empty before moving to the next queue. This minimizes the mean waiting time but can lead to starvation of other queues under heavy load. Based on [61], eq. (15).

- **Gated** (`polling_qsys_gated`): The server serves all customers present at the beginning of a visit period; customers arriving during the visit wait for the next cycle. This provides more predictable service than exhaustive polling. Based on [61], eq. (20).

- **1-Limited** (`polling_qsys_1limited`): The server serves at most one customer per queue visit, regardless of queue length. This provides the fairest service distribution across queues but results in higher mean waiting times. Based on [10].

All three functions accept arrays of Markovian Arrival Processes (MAPs) for arrivals, service times, and switchover times at each queue. Each MAP is specified as a pair of matrices (D_0, D_1), where D_0 captures phase transitions without events and D_1 captures transitions with events. The functions return an array of mean waiting times, one per queue. The system must satisfy the stability condition $\sum_i \lambda_i b_i < 1$, where λ_i and b_i are the arrival rate and mean service time at queue i, respectively.

These algorithms are automatically invoked by the MVA solver when a network contains queues configured with `PollingType.EXHAUSTIVE`, `PollingType.GATED`, or `PollingType.KLIMITED` scheduling, combined with switchover times specified via `setSwitchover`. Examples demonstrating polling systems are provided in the `examples/advanced/cyclicPolling/` directory.

B.16 Usage Guidelines

B.16.1 Algorithm Selection

The API provides multiple algorithms for similar tasks. Selection criteria include:

- **Accuracy** - Exact vs. approximate methods

- **Computational Cost** - Time and memory complexity

- **Model Support** - Supported feature sets

- **Numerical Stability** - Robustness to edge cases

B.16.2 Integration Patterns

- **Direct API Calls** - Use algorithms directly for specialized analysis

- **Solver Integration** - Algorithms are automatically selected by solvers

- **Custom Workflows** - Combine multiple algorithms for complex analysis

B.16.3 Error Handling

API functions follow consistent error handling patterns:

- Return null for invalid inputs

- Throw `RuntimeException` for computational failures

- Provide fallback algorithms when primary methods fail

B.17 Developer Notes

B.17.1 Java 8 Compatibility

All API implementations maintain Java 8 compatibility:

- Use `Collectors.toList()` instead of `Stream.toList()`

- Avoid `var` keyword and factory methods

- Traditional control flow structures

B.17.2   Performance Considerations

- Matrix operations are optimized for large-scale problems

- Algorithms provide both exact and approximate variants

- Memory usage is optimized through sparse representations

- Parallel implementations available for computationally intensive methods

Bibliography

[1] 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.

[2] S. Asmussen and M. Bladt. Point processes with finite-dimensional conditional probabilities. *Stochastic Processes and their Applications*, 82(1):127–142, 1999.

[3] 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.

[4] 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.

[5] 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.

[6] Mogens Bladt and Bo Friis Nielsen. *Matrix-Exponential Distributions in Applied Probability*. Springer, New York, 2017.

[7] A. Bobbio, A. Horváth, M. Scarpa, and M Telek. Acyclic discrete phase type distributions: properties and a parameter estimation algorithm. *Perform. Eval.*, 54(1):1–32, 2003.

[8] G. Bolch, S. Greiner, H. de Meer, and K. S. Trivedi. *Queueing Networks and Markov Chains*. Wiley, 2006.

[9] A. B. Bondi and W. Whitt. The influence of service-time variability in a closed network of queues. *Perform. Eval.*, 6:219–234, 1986.

[10] O. J. Boxma and W. P. Groenendijk. Waiting times in discrete-time cyclic-service systems. *IEEE Transactions on Communications*, 36(2):164–170, 1988.

[11] 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.

[12] G. Casale. CoMoM: Efficient class-oriented evaluation of multiclass performance models. *IEEE Trans. Software Engineering*, 35(2):162–177, 2009.

[13] G. Casale. Accelerating performance inference over closed systems by asymptotic methods. In *Proc. of ACM SIGMETRICS*. ACM Press, 2017.

[14] G. Casale. Integrated Performance Evaluation of Extended Queueing Network Models with Line. In *2020 Winter Simulation Conference (WSC)*, pages 2377–2388. IEEE, dec 2020.

[15] 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.

[16] G. Casale, P.G. Harrison, and O.W. Hong. Facilitating load-dependent queueing analysis through factorization. *Perform. Eval.*, 2021.

[17] 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.

[18] G. Casale, J. F. Pérez, and W. Wang. QD-AMVA: Evaluating systems with queue-dependent service requirements. In *Proceedings of IFIP PERFORMANCE*, 2015.

[19] G. Casale, M. Tribastone, and P. G. Harrison. Blending randomness in closed queueing network models. *Perform. Eval.*, 82:15–38, 2014.

[20] 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.

[21] K. M. Chandy and D. Neuse. Linearizer: A heuristic algorithm for queuing network models of computing systems. *Commun. ACM*, 25(2):126–134, 1982.

[22] W.-M. Chow. Approximations for large scale closed queueing networks. *Perform. Eval*, 3(1):1–12, 1983.

[23] A. E. Conway. *Fast Approximate Solution of Queueing Networks with Multi-Server Chain-Dependent FCFS Queues*, pages 385–396. Springer US, Boston, MA, 1989.

[24] 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.

[25] P. J. Courtois. *Decomposability: queueing and computer system applications*. Academic Press, New York, 1977.

[26] 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.

[27] 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.

[28] D. L. Eager and J. N. Lipscomb. The AMVA priority approximation. *Perform. Eval.*, 8(3):173–193, 1988.

[29] G. Franks. *Performance Analysis of Distributed Server Systems*. PhD thesis, Carleton, 1996.

[30] 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.

[31] G. Franks, P. Maly, C. M. Woodside, D. C. Petriu, A. Hubbard, and M. Mroz. *Layered Queueing Network Solver and Simulator User Manual*, 2012.

[32] 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.

[33] 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.

[34] D. T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *J. Phys. Chem.*, 81(25):2340–2361, 1977.

[35] A. Harel, S. Namn, and J. Sturm. Simple bounds for closed queueing networks. *Queueing Systems*, 31(1-2):125–135, 1999.

[36] P. Heidelberger and K. Trivedi. Queueing network models for parallel processing with asynchronous tasks. *IEEE Trans. Computers*, 100(11):1099–1109, 1982.

[37] G. Horváth and M. Telek. Sojourn times in fluid queues with independent and dependent input and output processes. *Performance Evaluation*, 79:160–181, 2014.

[38] 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).

[39] C. Knessl and C. Tier. Asymptotic expansions for large closed queueing networks with multiple job classes. *IEEE Trans. Computers*, 41(4):480–488, 1992.

[40] 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.

[41] D.D. Kouvatsos. Entropy maximisation and queueing network models. *Annals of Operations Research*, 48:63–126, 1994.

[42] S. S. Lavenberg. A perspective on queueing models of computer performance. *Perform. Eval.*, 10(1):53–76, 1989.

[43] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative System Performance*. Prentice-Hall, 1984.

[44] 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.

[45] KT Marshall. Some relationships between the distributions of waiting time, idle time and interoutput time in the gi/g/1 queue. *SIAM Journal on Applied Mathematics*, 16(2):324–327, 1968.

[46] H. Masuyama and T. Takine. Sojourn time distribution in a MAP/M/1 processor-sharing queue. *Oper. Res. Lett.*, 31(6):406–412, 2003.

[47] 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.

[48] M. Nuyens and A. Wierman. The foreground-background queue: A survey. *Performance Evaluation*, 65(3-4):286–307, 2008.

[49] J. F. Pérez and G. Casale. Assessing SLA compliance from Palladio component models. In *Proceedings of the 2nd MICAS*, 2013.

[50] J. F. Pérez and G. Casale. Line: Evaluating software applications in unreliable environments. *IEEE Trans. Reliability*, 66(3):837–853, Sept 2017.

[51] M. Reiser. A queueing network analysis of computer communication networks with window flow control. *IEEE Trans. Communications*, 27(8):1199–1209, 1979.

[52] M. Reiser. Mean-value analysis and convolution method for queue-dependent servers in closed queueing networks. *Perform. Eval.*, 1:7–18, 1981.

[53] M. Reiser and S. Lavenberg. Mean-value analysis of closed multichain queuing networks. *J. ACM*, 27:313–322, 1980.

[54] A. Riska and E. Smirni. ETAQA solutions for infinite Markov processes with repetitive structure. *INFORMS Journal on Computing*, 19(2):215–228, 2007.

[55] T. G. Robertazzi. *Computer Networks and Systems*. Springer, 2000.

[56] J. A. Rolia and K. C. Sevcik. The method of layers. *IEEE Trans. Software Engineering*, 21(8):689–700, August 1995.

[57] 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.

[58] 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.

[59] 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.

[60] K. Sevcik. Priority scheduling disciplines in queuing network models of computer systems. In *IFIP Congress*, 1977.

[61] H. Takagi. Queuing analysis of polling models. *ACM Computing Surveys*, 20(1):5–28, 1988.

[62] 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.

[63] 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.

[64] 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.

[65] M. Woodside. *Tutorial Introduction to Layered Modeling of Software Performance*. Carleton University, February 2013.

[66] 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.

[67] 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 scripts available under the `examples/` folder.

Table A.1: Examples

Example	Problem
`ag_closed_network`	Closed network agent model
`ag_jackson_network`	Jackson network agent model
`ag_multiclass_closed`	Multiclass closed agent model
`ag_tandem_open`	Open tandem agent model
`cache_replc_rr`	A small cache model with an open arrival process
`cache_replc_fifo`	A small cache model with a closed job population
`cache_replc_lru`	A layered network with a caching layer
`cache_compare_replc`	A layered network with a caching layer having a multi-level cache
`cache_replc_routing`	A caching model with state-dependent output routing
`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
`fcr_lossn`	Loss network analysis using finite capacity regions
`cqn_repairmen`	Solving a single-class exponential closed queueing network
`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_multiserver`	Local state space generation for a station in a closed network
`cqn_oneline`	1-line exact MVA solution of a cyclic network of PS and INF stations
`cqn_twoclass_erl`	Closed network with round robin scheduling
`cqn_bcmp_theorem`	Comparison of different scheduling policies that preserve the product-form solution
`cqn_repairmen_multi`	Multi-server closed queueing network with repairmen
`cqn_twoqueues_multi`	Closed queueing network with two multi-server queues
`cqn_twoqueues`	Simple closed network with two queues
`cs_implicit`	Class switching with implicit routing

Continued on next page

Table A.1 – Examples. *Continued from previous page*

Example	Problem
cs_multi_diamond	Class switching with multiple diamond patterns
cs_single_diamond	Class switching with single diamond pattern
cs_transient_class	Class switching with transient classes
fj_basic_open	A simple single class open fork-join network
fj_twoclasses_forked	A multiclass open fork-join network
fj_basic_nesting	A closed model with nested forks and joins
fj_nojoin	An open model with a fork but without a join
fj_basic_closed	A simple single class closed fork-join network
fj_serialfjs_open	Two open fork-joins subsystems in tandem
fj_cs_postfork	Two-class fork-join with a class that switches into the other after the fork
fj_cs_multi_visits	Two fork-joins 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_complex_serial	Complex serial fork-join network
fj_threebranches	Fork-join network with three branches
fj_cs_prefork	Fork-join network with class-switching before fork
fj_deep_nesting	Fork-join network with deep nesting
fj_serialfjs_closed	Closed serial fork-join network
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.
lqn_serial	Analyze a layered network specified in a LQNS XML file
lqn_multi_solvers	Specifying and solving a basic layered network
lqn_init	Specifying and solving a basic layered network with initialization
lqn_twotasks	Layered network with two tasks
lqn_bpmn	BPMN to layered network transformation
lqn_workflows	Workflow modeling in layered networks
lqn_function	Layered network function modeling
lqn_basic	Basic layered network example
ld_multiserver_fcfs	Solving a single-class load-dependent closed model
ld_multiserver_ps_twoclasses	Solving a two-node multiclass load-dependent closed model
ld_multiserver_ps	Solving a three-node multiclass load-dependent closed model
ld_class_dependence	Load-dependent model with class dependence
cqn_scheduling_dps	Parameterization of a discriminatory processor sharing (DPS) station
cqn_mmpp2_service	Automatic detection of solvers that cannot analyze the model
mqn_basic	Solving a queueing network model with both closed and open classes
mqn_multiserver_ps	A difficult mixed model with sparse routing among multi-server nodes
mqn_multiserver_fcfs	Mixed model with multiserver FCFS nodes
mqn_singleserver_fcfs	Mixed model with single server FCFS
mqn_singleserver_ps	Mixed model with single server PS
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_trace_driven	Trace-driven simulation of an M/M/1 queue
oqn_vsinks	A model illustrating the emulation of multiple sinks

Table A.1 – Examples. *Continued from previous page*

Example	Problem
`oqn_fourqueues`	A large multiclass example with PS and FCFS
`prio_hol_open`	A multiclass example with PS, SIRO, FCFS, FCFSPRIO priority
`prio_hol_closed`	A high-load multiclass example with PS, SIRO, FCFS, FCFSPRIO priority
`prio_psprio`	A repaimen model with PS priority scheduling.
`prio_identical`	Priority model with identical classes
`renv_twostages_repairmen`	Solving a model in a 2-stage random environment with exponential rates
`renv_fourstages_repairmen`	Solving a model in a 4-stage random environment with Coxian rates
`renv_threestages_repairmen`	Solving a model in a 3-stage random environment with Erlang rates
`renv_node_breakdown`	Using simplified breakdown/repair API for node failures in random environments
`renv_node_breakdown_example`	Using simplified breakdown/repair API for node failures in random environments
`example_rewardModel_1`	Reward-based CTMC analysis using custom reward functions (setReward, getReward, getAvgReward)
`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
`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`	JMT simulation of a simple stochastic Petri net model
`spn_open_sevenplaces`	JMT simulation of a complex stochastic Petri net model
`spn_twomodes`	Stochastic Petri net with two modes
`spn_fourmodes`	Stochastic Petri net with four modes
`spn_inhibiting`	Stochastic Petri net with inhibiting transitions
`spn_closed_fourplaces`	Closed Stochastic Petri net with four places
`spn_closed_twoplaces`	Closed Stochastic Petri net with two places
`spn_basic_closed`	Basic closed stochastic Petri net
`tut01_mm1_basics`	M/M/1 queue basics
`tut02_mg1_multiclass_solvers`	M/G/1 multiclass solvers
`tut03_repairmen`	Repairmen model
`tut04_lb_routing`	Load balancing routing
`tut05_completes_flag`	Complete flag usage
`tut06_cache_lru_zipf`	Cache LRU Zipf distribution
`tut07_respt_cdf`	Response time CDF analysis
`tut08_opt_load_balancing`	Optimal load balancing
`tut09_dep_process_analysis`	Dependent process analysis
`tut10_lqn_basics`	Layered queueing network basics
`tut12_posterior_analysis`	Posterior analysis and parameter inference
`lcq_singlehost`	Single host layered cache queueing
`lcq_threehosts`	Three hosts layered cache queueing

Table A.1 – Examples. *Continued from previous page*

Example	Problem
swt_basic	Basic switchover times model
polling_exhaustive_exp	Exhaustive polling with exponential times
polling_gated	Gated polling
polling_klimited	K-limited polling
polling_exhaustive_det	Exhaustive polling with deterministic times

Appendix B

API Function Reference

This appendix provides a comprehensive catalog of all API functions available in the `jline.api` package. The table lists each function name, its organizational package, and a brief description of its purpose.

Table B.1: Complete API Function Reference

Function Name	Package	Description
`amap2_fit_gamma`	mam	Fits AMAP(2) distributions to match moments and correlation characteristics
`amap2_fit_gamma_map`	mam	Fits AMAP(2) by approximating arbitrary-order MAP with preserved correlation structure
`amap2_fit_gamma_trace`	mam	Fits AMAP(2) from empirical traces while preserving autocorrelation characteristics
`aph2_adjust`	mam	Adjusts moments to ensure feasibility bounds for APH(2) fitting procedures
`aph2_assemble`	mam	Constructs APH(2) transition matrices from specified rates and transition probabilities
`aph2_fit`	mam	Fits APH(2) distributions to match given moments with automatic feasibility adjustment
`aph2_fit_map`	mam	Fits APH(2) distributions by approximating arbitrary-order MAP processes
`aph2_fit_trace`	mam	Fits APH(2) distributions from empirical inter-arrival time traces
`aph2_fitall`	mam	Fits multiple APH(2) distributions to match given moments with exhaustive parameter search
`aph_bernstein`	mam	Constructs APH distributions using Bernstein exponential approximation methods
`aph_fit`	mam	Fits APH distributions to specified moments using optimization and approximation techniques
`aph_rand`	mam	APH random generation algorithms
`aph_simplify`	mam	Simplifies and combines APH distributions using structural pattern operations
`cache_erec`	cache	Implements exact recursive (EREC) algorithms for cache system analysis

Table B.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
`cache_gamma`	cache	Computes cache access factors from request arrival rates and routing matrices
`cache_gamma_lp`	cache	Computes cache access factors using linear programming optimization methods
`cache_miss`	cache	Provides general-purpose algorithms for computing cache miss rates across
`cache_miss_asy`	cache	Provides asymptotic approximation methods for cache miss rate analysis
`cache_miss_fpi`	cache	Computes cache miss probabilities using fixed-point iteration methods
`cache_miss_rayint`	cache	Estimates cache miss rates using ray method for partial differential equations
`cache_mva`	cache	Implements Mean Value Analysis algorithms for cache system performance
`cache_mva_miss`	cache	Implements Mean Value Analysis (MVA) algorithms for computing cache miss
`cache_prob_erec`	cache	Computes exact cache state probabilities using recursive methods based on
`cache_prob_fpi`	cache	Computes cache state probabilities using fixed-point iteration algorithms
`cache_prob_rayint`	cache	Computes cache state probabilities using ray integration methods for
`cache_rayint`	cache	Implements ray integration techniques for cache system analysis using
`cache_rrm_meanfield_ode`	cache	Implements the mean field ordinary differential equation system for Random
`cache_t_hlru`	cache	Computes response time metrics for Hierarchical Least Recently Used (H-LRU)
`cache_t_lrum`	cache	Computes response time metrics for Least Recently Used with Multiple servers
`cache_t_lrum_map`	cache	Analyzes response times for LRUM cache systems with Markovian Arrival
`cache_ttl_hlru`	cache	Implements TTL approximation for Hierarchical LRU (H-LRU) cache systems
`cache_ttl_lrua`	cache	Implementation of Time-To-Live (TTL) approximation for LRU(A) cache systems
`cache_ttl_lrum`	cache	Implementation of Time-To-Live approximation for Least Recently Used with
`cache_ttl_lrum_map`	cache	Combines TTL approximation with LRUM cache policies and Markovian Arrival
`cache_ttl_tree`	cache	Tree-based TTL cache analysis implementation for the LINE solver framework
`cache_xi_bvh`	cache	Computes cache xi terms using the iterative method from Gast-van Houdt
`cache_xi_fp`	cache	Estimates cache xi terms using fixed-point algorithms. Xi terms represent

Continued on next page

Table B.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
`ctmc_courtois`	mc	Courtois decomposition for nearly completely decomposable CTMCs
`ctmc_kms`	mc	Koury-McAllister-Stewart aggregation-disaggregation method for CTMCs
`ctmc_makeinfgen`	mc	Constructs and validates infinitesimal generator matrices for continuous-time
`ctmc_multi`	mc	Multi-level aggregation method for CTMCs
`ctmc_pseudostochcomp`	mc	API function for mc operations
`ctmc_rand`	mc	Generates random infinitesimal generator matrices for continuous-time Markov chains
`ctmc_randomization`	mc	Converts a continuous-time Markov chain into an equivalent discrete-time chain
`ctmc_relsolve`	mc	Equilibrium distribution of a continuous-time Markov chain re-normalized with respect to
`ctmc_simulate`	mc	Generates sample paths for CTMCs using the standard simulation algorithm with
`ctmc_solve`	mc	Computes the steady-state probability distribution for CTMCs by solving the linear
`ctmc_solve_reducible`	mc	Solve reducible CTMCs by converting to DTMC via randomization
`ctmc_ssg`	mc	API function for mc operations
`ctmc_ssg_reachability`	mc	CTMC State Space Generator for Reachability Analysis
`ctmc_stmonotone`	mc	Computes the stochastically monotone upper bound for a CTMC
`ctmc_stochcomp`	mc	Implements stochastic complementarity analysis for CTMCs to identify strongly
`ctmc_takahashi`	mc	Takahashi's aggregation-disaggregation method for CTMCs
`ctmc_testpf_kolmogorov`	mc	Test if a CTMC has product form using Kolmogorov's criteria
`ctmc_timereverse`	mc	Computes the infinitesimal generator of the time-reversed continuous-time
`ctmc_transient`	mc	Computes transient probabilities for CTMCs using numerical integration of the
`ctmc_uniformization`	mc	CTMC Transient Analysis via Uniformization
`dtmc_isfeasible`	mc	Check if a matrix represents a feasible DTMC transition matrix
`dtmc_makestochastic`	mc	Converts non-negative matrices into valid discrete-time Markov chain transition
`dtmc_rand`	mc	Generates random stochastic transition matrices for discrete-time Markov chains
`dtmc_simulate`	mc	Generates sample trajectories for DTMCs by sampling from the transition probability
`dtmc_solve`	mc	Computes the steady-state probability distribution for DTMCs by converting the
`dtmc_solve_reducible`	mc	Result class for DTMC solve reducible
`dtmc_stochcomp`	mc	Returns the stochastic complement of a DTMC
`dtmc_timereverse`	mc	Compute the infinitesimal generator of the time-reversed DTMC
`dtmc_uniformization`	mc	Result class for DTMC uniformization analysis containing the probability vector and maximum iterations used

Continued on next page

Table B.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
`lossn_erlangfp`	lossn	Implements fixed-point algorithms for analyzing loss networks using Erlang
`lsn_max_multiplicity`	lsn	Computes maximum multiplicity constraints for load sharing network (LSN)
`m3pp22_fitc_approx_cov`	mam.m3pp	Implements parameter fitting for second-order Marked Markov Modulated Poisson Process
`m3pp22_fitc_approx_cov_multiclass`	mam.m3pp	Implements constrained optimization for fitting M3PP(2,2) parameters given an underlying
`m3pp22_interleave_fitc`	mam.m3pp	Implements lumped superposition of multiple M3PP(2,2) processes using interleaved
`m3pp2m_fitc`	mam.m3pp	Implements exact fitting of second-order Marked Markov Modulated Poisson Process
`m3pp2m_fitc_approx`	mam.m3pp	Implements approximation-based fitting for M3PP(2,m) using optimization methods
`m3pp2m_fitc_approx_ag`	mam.m3pp	Implements auto-gamma approximation method for M3PP(2,m) parameter fitting
`m3pp2m_fitc_approx_ag_multiclass`	mam.m3pp	Implements multiclass auto-gamma fitting for M3PP(2,m) with variance and covariance
`m3pp2m_interleave`	mam.m3pp	Implements interleaved superposition of multiple M3PP(2,m) processes to construct
`m3pp_interleave_fitc`	mam.m3pp	Implements fitting and interleaving of k second-order M3PP processes with varying
`m3pp_interleave_fitc_theoretical`	mam.m3pp	Implements theoretical MMAP fitting through M3PP interleaving using analytical
`m3pp_interleave_fitc_trace`	mam.m3pp	Implements M3PP interleaving and fitting directly from empirical trace data
`m3pp_rand`	mam.m3pp	Implements random generation of Markovian Multi-class Point Processes (M3PP)
`m3pp_superpos_fitc`	mam.m3pp	Implements superposition-based fitting of k second-order M3PP processes into
`m3pp_superpos_fitc_theoretical`	mam.m3pp	Implements superposition fitting of k second-order M3PP processes using theoretical
`mamap22_fit_gamma_fs_trace`	mam	Fits MAMAP(2,2) from trace data using gamma autocorrelation and forward-sigma characteristics
`mamap22_fit_multiclass`	mam	Fits MAMAP(2,2) processes for two-class systems with forward moments and sigma characteristics
`mamap2m_coefficients`	mam	Computes coefficients for MAMAP(2,m) fitting formulas in canonical forms
`mamap2m_fit`	mam	Fits MAMAP(2,m) processes matching moments, autocorrelation, and class characteristics
`mamap2m_fit_fb_multiclass`	mam	Fits MAMAP using combined forward and backward moment characteristics for multiclass systems
`mamap2m_fit_gamma_fb_mmap`	mam	Fits MAMAP with autocorrelation control using forward-backward moments from MMAP input
`mamap2m_fit_mmap`	mam	Fits MAPH/MAMAP(2,m) by approximating characteristics of input MMAP processes

Table B.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
`mamap2m_fit_trace`	mam	Fits MAMAP(2,m) processes from empirical trace data with inter-arrival times and class labels
`map2_fit`	mam	Fits MAP(2) processes to match specified moments and autocorrelation decay rates
`map2mmpp`	mam	Converts MAP representations to MMPP format for compatibility with MMPP-specific algorithms
`map_acf`	mam	Computes autocorrelation function (ACF) values for MAP inter-arrival times at specified lags
`map_acfc`	mam	Computes ACFC values for MAP counting processes over time intervals, measuring temporal correlation
`map_block`	mam	Constructs MAP(2) representations from moment and autocorrelation parameters using fallback
`map_ccdf_derivative`	mam	Computes derivatives of MAP complementary cumulative distribution functions at zero
`map_cdf`	mam	Computes CDF values for MAP inter-arrival times using CTMC uniformization techniques
`map_checkfeasible`	mam	Comprehensive validation of MAP matrices including stochastic properties, numerical stability,
`map_count_mean`	mam	Computes mean number of arrivals in MAP counting processes over specified time intervals
`map_count_moment`	mam	Computes power moments of MAP counting processes using moment generating functions and
`map_count_var`	mam	Computes variance of MAP counting processes over specified time intervals using matrix
`map_embedded`	mam	Computes embedded DTMC matrices from MAP representations by extracting transition
`map_erlang`	mam	Constructs MAP representations of Erlang-k processes with specified means and phases
`map_exponential`	mam	Creates MAP representations of exponential inter-arrival time distributions with specified
`map_feasblock`	mam	Constructs feasible MAP representations when exact moment matching fails by adjusting
`map_feastol`	mam	Provides standard tolerance values for numerical feasibility checks in MAP algorithms
`map_gamma`	mam	Computes gamma parameter measuring autocorrelation decay rates in MAP processes
`map_gamma2`	mam	Computes largest non-unit eigenvalue of embedded DTMC for MAP correlation characterization
`map_hyperexp`	mam	Constructs MAP representations of two-phase hyperexponential renewal processes
`map_idc`	mam	Computes asymptotic index of dispersion for MAP counting processes, measuring long-term
`map_infgen`	mam	Computes infinitesimal generator matrix of underlying CTMC by combining MAP transition
`map_isfeasible`	mam	Provides convenient interface for MAP feasibility validation with configurable tolerance

Table B.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
`map_joint`	mam	Computes joint moments of MAP inter-arrival times for advanced statistical characterization
`map_jointpdf_derivative`	mam	Computes partial derivatives of MAP joint probability density functions at origin
`map_kpc`	mam	Computes Kronecker product composition of multiple MAPs for building complex arrival processes
`map_kurt`	mam	Computes kurtosis of MAP inter-arrival times measuring tail heaviness and distribution shape
`map_lambda`	mam	MAP arrival rate computation algorithms
`map_largemap`	mam	Provides size thresholds for determining when MAP algorithms should switch to
`map_mark`	mam	Creates Marked MAP (MMAP) representations by adding class labels to MAP arrivals
`map_max`	mam	Computes MAP representation of maximum inter-arrival times from independent MAP processes
`map_mean`	mam	MAP mean inter-arrival time computation algorithms
`map_mixture`	mam	Creates probabilistic mixtures of MAP processes with specified mixture probabilities
`map_moment`	mam	Computes raw moments of MAP inter-arrival times using matrix inversion techniques
`map_normalize`	mam	Sanitizes MAP matrices by ensuring non-negativity constraints and proper diagonal adjustments
`map_pdf`	mam	Computes PDF values for MAP inter-arrival times using matrix exponential techniques
`map_pie`	mam	Computes steady-state probability vector of embedded discrete-time Markov chain
`map_piq`	mam	Computes steady-state probability vector of underlying continuous-time Markov chain
`map_pntiter`	mam	Computes exact arrival probabilities using iterative numerical methods based on Neuts and Li
`map_pntquad`	mam	Computes MAP point process probabilities using ODE quadrature methods with Runge-Kutta integration
`map_prob`	mam	Computes equilibrium probability distribution of underlying CTMC for MAP analysis
`map_rand`	mam	Generates random MAP representations for testing, simulation, and statistical analysis
`map_randn`	mam	Generates random MAP samples with added numerical noise for robustness testing
`map_renewal`	mam	Creates renewal MAP by removing correlations to obtain memoryless arrival processes
`map_sample`	mam	Generates random samples from MAP distributions for simulation and empirical analysis
`map_scale`	mam	Rescales MAP inter-arrival time distributions to achieve specified mean values
`map_scv`	mam	Computes SCV of MAP inter-arrival times as normalized dispersion measure

Table B.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
`map_skew`	mam	Computes skewness of MAP inter-arrival times measuring asymmetry in distributions
`map_stochcomp`	mam	Performs state elimination through stochastic complementation while preserving MAP properties
`map_sum`	mam	Computes MAP representations of sums of identical MAP processes for load scaling
`map_sumind`	mam	Computes MAP representations of sums of independent MAP processes for modeling
`map_super`	mam	Creates superposition of MAP processes using Kronecker product techniques
`map_timereverse`	mam	Computes time-reversed MAP by adjusting transition rates based on stationary distributions
`map_var`	mam	MAP variance computation algorithms
`map_varcount`	mam	Computes variance of event counts in MAP processes over specified time intervals
`maph2m_fit`	mam	Fits MAPH(2,m) processes to match ordinary moments, class probabilities, and backward moments
`maph2m_fit_mmap`	mam	Fits MAPH(2,m) by approximating characteristics of input MMAP processes
`maph2m_fit_multiclass`	mam	Fits MAPH(2,m) models to multiclass characteristics with class-specific parameters
`maph2m_fit_trace`	mam	Fits MAPH(2,m) from empirical trace data for multiclass service time modeling
`maph2m_fitc_approx`	mam	Fits MAPH(2,m) using approximation methods for count statistics when exact solutions fail
`maph2m_fitc_theoretical`	mam	Fits MAPH(2,m) using theoretical count statistics for precise parameter estimation
`mapqn_bnd_lr`	mapqn	Implements general linear reduction methods for computing performance
`mapqn_bnd_lr_mva`	mapqn	Implements linear reduction bounds for MAP queueing networks using Mean
`mapqn_bnd_lr_pf`	mapqn	Implements linear reduction bounds specialized for product-form MAP
`mapqn_bnd_qr`	mapqn	Implements general quadratic reduction methods for computing performance
`mapqn_bnd_qr_delay`	mapqn	Implements quadratic reduction bounds for delay systems in MAP queueing
`mapqn_bnd_qr_ld`	mapqn	Implements quadratic reduction bounds for load-dependent MAP queueing
`mapqn_lpmodel`	mapqn	Base class for representing MAP queueing network linear programming models
`mapqn_parameters`	mapqn	Defines the base parameter structure for MAP queueing network analysis
`mapqn_parameters_factory`	mapqn	Factory class for creating parameter objects for MAP queueing network

Table B.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
`mapqn_qr_bounds_bas`	mapqn	Implements Queue-Router bounds using the Balanced Asymptotic Scaling (BAS)
`mapqn_qr_bounds_rsrd`	mapqn	Implements Queue-Router (QR) bounds using the Randomized Simultaneous
`mmap_backward_moment`	mam	Computes backward moments of MMAP inter-arrival times for each marked class
`mmap_compress`	mam	Compresses MMAP using various approximation methods including mixture, matching,
`mmap_count_idc`	mam	Computes IDC values for each marked class in MMAP counting processes
`mmap_count_lambda`	mam	Computes arrival rate vectors for each marked class in MMAP processes
`mmap_count_mcov`	mam	Computes count covariance matrices between marked classes in MMAP processes
`mmap_count_mean`	mam	Computes mean count vectors for each marked class in MMAP counting processes
`mmap_count_var`	mam	Computes variance vectors for counting processes of each marked class in MMAP
`mmap_cross_moment`	mam	Computes cross-moment matrices between different marked classes in MMAP processes
`mmap_embedded`	mam	Computes embedded discrete-time Markov chain for MMAP processes
`mmap_exponential`	mam	Constructs MMAP with exponential inter-arrival distributions for each marked class
`mmap_forward_moment`	mam	Computes forward moments of MMAP inter-arrival times for each marked class
`mmap_hide`	mam	Hides specified arrival classes in MMAP processes by removing observable events
`mmap_idc`	mam	Computes asymptotic IDC for each marked class in MMAP as time approaches infinity
`mmap_isfeasible`	mam	Validates mathematical feasibility of MMAP representations including stochastic
`mmap_issym`	mam	Checks if an MMAP is symmetric
`mmap_lambda`	mam	MMAP arrival rate computation algorithms
`mmap_maps`	mam	Extracts individual MAP processes for each marked class from MMAP representations
`mmap_mark`	mam	Converts a Markovian Arrival Process with marked arrivals (MMAP) into a new MMAP with redefined classes based on a given probability matrix
`mmap_max`	mam	Computes element-wise maximum of MMAP processes for synchronization analysis
`mmap_mixture`	mam	Creates probabilistic mixtures of MMAP processes with specified weights
`mmap_mixture_fit`	mam	Fits a mixture of Markovian Arrival Processes (MMAPs) to match the given cross-moments

Table B.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
`mmap_mixture_fit_mmap`	mam	Fits a mixture of Markovian Arrival Processes (MMAPs) to match the given moments
`mmap_mixture_order2`	mam	Creates a second-order MMAP mixture from a collection of MMAPs
`mmap_modulate`	mam	Modulates an MMAP by another MMAP, creating a compound arrival process
`mmap_normalize`	mam	Normalizes MMAP matrices to ensure feasibility and mathematical validity
`mmap_pc`	mam	Computes the proportion of counts (PC) for each type in a Markovian Arrival Process with marked arrivals (MMAP)
`mmap_pie`	mam	Computes steady-state probability vectors for each marked class in MMAP processes
`mmap_rand`	mam	Generates random MMAP representations for testing and simulation purposes
`mmap_sample`	mam	Generates random samples from MMAP distributions for each marked class
`mmap_scale`	mam	Rescales MMAP inter-arrival distributions to achieve specified mean values
`mmap_shorten`	mam	Converts an MMAP representation from M3A format to BUTools format
`mmap_sigma`	mam	Computes one-step class transition probabilities for a Marked Markovian Arrival Process (MMAP)
`mmap_sigma2`	mam	Computes two-step class transition probabilities for a Markovian Arrival Process (MMAP)
`mmap_sum`	mam	Computes superposition of MMAP processes creating independent multiclass arrival streams
`mmap_super`	mam	Combines multiple MMAP processes into superposed multiclass arrival streams
`mmap_super_safe`	mam	API function for mam operations
`mmap_timereverse`	mam	Computes the time-reversed version of a Markovian Arrival Process with marked arrivals (MMAP)
`mmpp2_fit`	mam	Fits MMPP(2) models to match specified moments and correlation characteristics
`mmpp2_fit1`	mam	Fits MMPP(2) models using simplified single-parameter approach for specific scenarios
`mmpp2_fitc`	mam	Fits MMPP(2) models using count statistics and index of dispersion criteria
`mmpp2_fitc_approx`	mam	Fits MMPP(2) using optimization-based approximation methods for count statistics
`mmpp_rand`	mam	Generates random MMPP models with diagonal D1 matrices for testing and simulation
`ms_additivesymmetricchisquared`	measures	Additive symmetric chi-squared distance between two probability distributions
`ms_adtest`	measures	Implements the Anderson-Darling test for assessing whether a sample comes from

Table B.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
ms_avgl1linfty	measures	Average L1 L-infinity distance between two probability distributions
ms_bhattacharyya	measures	Computes the Bhattacharyya distance measuring the similarity between probability
ms_canberra	measures	Canberra distance between two probability distributions
ms_chebyshev	measures	Chebyshev Distance for Probability Distributions
ms_chisquared	measures	Squared chi-squared distance between two probability distributions
ms_cityblock	measures	Implements the City block distance between probability distributions
ms_clark	measures	Clark distance between two probability distributions
ms_condentropy	measures	Computes conditional entropy measuring the remaining uncertainty
ms_cramer_von_mises	measures	Implements the Cramer-von Mises test statistic for comparing two empirical distributions
ms_cosine	measures	Computes cosine distance (1 - cosine similarity) measuring the angle between two
ms_czekanowski	measures	Czekanowski distance between two probability distributions
ms_dice	measures	Dice distance between two probability distributions
ms_divergence	measures	Divergence distance between two probability distributions
ms_entropy	measures	Implements Shannon entropy for discrete random variables
ms_euclidean	measures	Implements the standard Euclidean distance between probability distributions
ms_fidelity	measures	Fidelity distance between two probability distributions
ms_gower	measures	Gower distance between two probability distributions
ms_harmonicmean	measures	Harmonic mean distance between two probability distributions
ms_hellinger	measures	Computes the Hellinger distance measuring dissimilarity between probability distributions
ms_intersection	measures	Intersection distance between two probability distributions
ms_jaccard	measures	Jaccard distance between two probability distributions
ms_jeffreys	measures	Jeffreys divergence between two probability distributions
ms_jensendifference	measures	Jensen difference divergence between two probability distributions
ms_jensenshannon	measures	Implements Jensen-Shannon divergence, a symmetric and bounded version of
ms_jointentropy	measures	Computes joint entropy measuring the uncertainty of joint distributions
ms_kdivergence	measures	K-divergence between two probability distributions
ms_kolmogorov_smirnov	measures	Implements the Kolmogorov-Smirnov test for determining if a sample follows
ms_kuiper	measures	Implements the Kuiper test statistic, a rotation-invariant variant of the Kolmogorov-Smirnov
ms_kulczynskid	measures	Kulczynski d distance between two probability distributions
ms_kulczynskis	measures	Kulczynski s distance between two probability distributions
ms_kullbackleibler	measures	Implements the Kullback-Leibler divergence measuring distribution differences

Table B.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
ms_kumarhassebrook	measures	Kumar-Hassebrook distance between two probability distributions
ms_kumarjohnson	measures	Kumar-Johnson distance between two probability distributions
ms_lorentzian	measures	Lorentzian distance between two probability distributions
ms_matusita	measures	Matusita distance between two probability distributions
ms_minkowski	measures	Minkowski Distance for Probability Distributions
ms_motyka	measures	Motyka distance between two probability distributions
ms_mutinfo	measures	Computes mutual information measuring the amount of shared information
ms_neymanchisquared	measures	Neyman chi-squared distance between two probability distributions
ms_nmi	measures	Computes normalized mutual information providing a scale-invariant measure
ms_nvi	measures	Computes normalized variation information measuring the normalized distance
ms_pearsonchisquared	measures	Pearson chi-squared distance between two probability distributions
ms_probsymmchisquared	measures	Probabilistic symmetry chi-squared distance between two probability distributions
ms_relatentropy	measures	Computes relative entropy measuring the information difference
ms_product	measures	Product distance between two probability distributions
ms_ruzicka	measures	Ruzicka distance between two probability distributions
ms_soergel	measures	Soergel distance between two probability distributions
ms_sorensen	measures	Sorensen distance between two probability distributions
ms_squaredchord	measures	Squared chord distance between two probability distributions
ms_squaredeuclidean	measures	Squared Euclidean distance between two probability distributions
ms_taneja	measures	Taneja distance between two probability distributions
ms_tanimoto	measures	Tanimoto distance between two probability distributions
ms_topsoe	measures	Topsoe distance between two probability distributions
ms_wasserstein	measures	Implements the Wasserstein distance measuring the minimum cost to transform
ms_wavehegdes	measures	Wave-Hedges distance between two probability distributions
mtrace_backward_moment	trace	Computes backward moments of a multi-class trace
mtrace_bootstrap	trace	Implements bootstrap resampling methods for multi-class empirical trace data
mtrace_count	trace	Computes count statistics from a multi-class trace over specified time windows
mtrace_cov	trace	Computes the covariance matrix for multi-type traces
mtrace_cross_moment	trace	Computes the k-th order moment of the inter-arrival time between an event
mtrace_forward_moment	trace	Computes the forward moments of a marked trace
mtrace_iat2counts	trace	Computes the per-class counting processes of T, i.e., the counts after
mtrace_joint	trace	Given a multi-class trace, computes the empirical class-dependent joint

Table B.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
mtrace_mean	trace	Computes per-class means for multi-class empirical trace data. Enables separate
mtrace_merge	trace	Merges two traces in a single marked (multiclass) trace
mtrace_moment	trace	Computes empirical class-dependent statistical moments for multi-class trace data
mtrace_moment_simple	trace	Computes the k-th order moment of the inter-arrival time between an event
mtrace_pc	trace	Computes the probabilities of arrival for each class
mtrace_sigma	trace	Computes the empirical probability of observing a specific 2-element
mtrace_sigma2	trace	Computes the empirical probability of observing a specific 3-element
mtrace_split	trace	Given a multi-class trace with inter-arrivals T and labels L,
mtrace_summary	trace	Computes summary statistics for multiple trace analysis, providing
npfqn_nonexp_approx	npfqn	Implements approximation methods for non-product-form queueing networks
npfqn_traffic_merge	npfqn	Implements traffic merging algorithms for non-product-form queueing networks
npfqn_traffic_merge_cs	npfqn	Implements traffic merging algorithms for non-product-form queueing networks
npfqn_traffic_split_cs	npfqn	Implements traffic splitting algorithms for non-product-form queueing networks
pfqn_ab	pfqn.ld	Implements the Akyildiz-Bolch linearizer method for analyzing closed product-form queueing networks
pfqn_aql	pfqn.mva	Implements the Aggregate Queue Length (AQL) approximation method for analyzing closed
pfqn_bs	pfqn.mva	Implements the classic Bard-Schweitzer approximate MVA algorithm for closed queueing networks
pfqn_ca	pfqn.nc	Convolution Algorithm for Product-Form Networks
pfqn_cdfun	pfqn.ld	Provides functionality to evaluate class-dependent scaling functions in load-dependent queueing networks
pfqn_comomrm	pfqn.nc	Implements the Convolution Method of Moments specialized for repairman queueing models
pfqn_comomrm_ld	pfqn.ld	Implements the Convolution Method of Moments (COMOM) for computing normalizing constants in
pfqn_conwayms	pfqn.mva	Implements the Conway-Maxwell approximation method for analyzing closed queueing networks
pfqn_cub	pfqn.nc	Implements the cubature (multi-dimensional integration) approach for computing normalizing
pfqn_egflinearizer	pfqn.mva	Implements the Extended General-Form linearizer approximation for closed queueing networks
pfqn_fnc	pfqn.ld	Computes scaling factors for load-dependent functional servers in product-form queueing networks
pfqn_gflinearizer	pfqn.mva	Implements the general-form linearizer approximation for closed queueing networks with

Table B.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
pfqn_gld	pfqn.ld	Implements the generalized convolution algorithm for computing normalizing constants in
pfqn_gld_complex	pfqn.ld	Extends the generalized load-dependent convolution algorithm to handle complex-valued
pfqn_gldsingle	pfqn.ld	Provides specialized auxiliary function for computing normalizing constants in single-class
pfqn_gldsingle_complex	pfqn.ld	Provides specialized auxiliary function for computing normalizing constants in single-class
pfqn_kt	pfqn.nc	Implements the Knessl-Tier asymptotic expansion using the ray method for computing
pfqn_le	pfqn.nc	Implements the Laguerre expansion approach for computing normalizing constants in
pfqn_le_fpi	pfqn.nc	Implements the fixed-point iteration algorithm used in the Laguerre expansion method
pfqn_le_fpiz	pfqn.nc	Implements the fixed-point iteration algorithm for the Laguerre expansion method
pfqn_le_hessian	pfqn.nc	Computes the Hessian matrix used in the Laguerre expansion method for second-order
pfqn_le_hessianz	pfqn.nc	Computes the Hessian matrix used in the Laguerre expansion method for closed queueing
pfqn_linearizer	pfqn.mva	Linearizer Approximate MVA for Product-Form Networks
pfqn_linearizerms	pfqn.mva	Implements the multi-server version of Krzesinski's linearizer approximation for closed
pfqn_linearizermx	pfqn.mva	Implements linearizer-based approximation methods for mixed queueing networks with
pfqn_linearizerpp	pfqn.mva	Implements the Linearizer++ algorithm for closed queueing networks with enhanced accuracy
pfqn_lldfun	pfqn.ld	Evaluates limited load-dependent (LLD) scaling functions using spline interpolation for
pfqn_ls	pfqn.nc	Implements the logistic sampling approach for computing normalizing constants in
pfqn_mci	pfqn.nc	Implements Monte Carlo integration approaches including Importance Monte Carlo Integration
pfqn_mmint2	pfqn.nc	Implements numerical integration for computing normalizing constants in multi-class
pfqn_mmint2_gausslegendre	pfqn.nc	Implements Gauss-Legendre quadrature integration for computing normalizing constants
pfqn_mmsample2	pfqn.nc	Implements importance sampling for computing normalizing constants in multi-class
pfqn_mom	pfqn.nc	Implements the Method of Moments using exact arithmetic with BigFraction for computing
pfqn_mu_ms	pfqn.ld	Computes load-dependent scaling factors for multi-server queueing stations with finite
pfqn_mushift	pfqn.ld	Provides utility function for shifting load-dependent scaling vectors by one position,
pfqn_mva	pfqn.mva	Mean Value Analysis for Product-Form Queueing Networks

Continued on next page

Table B.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
pfqn_mvald	pfqn.ld	Load-Dependent Mean Value Analysis
pfqn_mvaldms	pfqn.ld	Provides wrapper functionality for load-dependent Mean Value Analysis with automatic
pfqn_mvaldmx	pfqn.ld	Implements Mean Value Analysis for mixed queueing networks with both open and closed classes
pfqn_mvaldmx_ec	pfqn.ld	Provides auxiliary functionality for computing EC terms used in load-dependent Mean Value
pfqn_mvams	pfqn.mva	Provides comprehensive MVA solution for mixed queueing networks with multi-server stations
pfqn_mvamx	pfqn.mva	Implements MVA for mixed networks containing both open and closed classes without multi-server
pfqn_nc	pfqn.nc	Normalizing Constant Methods for Product-Form Networks
pfqn_nc_sanitize	pfqn.nc	Sanitizes and preprocesses parameters for product-form queueing network models to
pfqn_nca	pfqn.nc	Implements the Normalizing Constant Approximation method for single-class closed
pfqn_ncld	pfqn.ld	Provides the main entry point for computing normalizing constants in load-dependent
pfqn_nrl	pfqn.nc	Implements the Normal Random Lattice approach for computing normalizing constants
pfqn_nrp	pfqn.nc	Implements the Normal Random Permutation approach for computing normalizing constants
pfqn_panacea	pfqn.nc	Implements the PANACEA approximation method for computing normalizing constants in
pfqn_pff_delay	pfqn.nc	Computes the product-form factor for delay stations in closed queueing networks
pfqn_procomom2	pfqn.ld	Implements the probabilistic class-oriented method of moments for analyzing
pfqn_propfair	pfqn.nc	Implements the proportionally fair allocation method using convex optimization
pfqn_qzgblow	pfqn.mva	Computes the lower Geometric Bound (GB) for queue lengths in closed single-class queueing
pfqn_qzgbup	pfqn.mva	Computes the upper Geometric Bound (GB) for queue lengths in closed single-class queueing
pfqn_rd	pfqn.nc	Implements the Random Discretization approach for computing normalizing constants in
pfqn_recal	pfqn.nc	Implements the RECAL (Recursive Calculation) algorithm for computing normalizing constants
pfqn_schmidt	pfqn.ld	Schmidt method for load-dependent MVA with multi-server stations
pfqn_sqni	pfqn.mva	Implements the Single Queue Network Interpolation method for analyzing multi-class closed
pfqn_stdf	pfqn.nc	Implements McKenna's 1987 method for computing sojourn time distributions at
pfqn_stdf_heur	pfqn.nc	Implements a heuristic variant of McKenna's 1987 method for computing sojourn time

Continued on next page

Table B.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
pfqn_xia	pfqn.ld	Implements Xia's asymptotic approximation method for computing normalizing constants
pfqn_xzabalow	pfqn.mva	Computes the lower ABA bound for throughput in closed single-class queueing networks
pfqn_xzabaup	pfqn.mva	Computes the upper ABA bound for throughput in closed single-class queueing networks
pfqn_xzgsblow	pfqn.mva	Computes the lower GSB for throughput in closed single-class queueing networks using
pfqn_xzgsbup	pfqn.mva	Computes the upper GSB for throughput in closed single-class queueing networks using
ph_reindex	mam	Reindexes phase-type distribution maps for network models using integer station and class indices
polling_qsys_1limited	polling	Implements analysis algorithms for 1-limited polling systems where the
polling_qsys_exhaustive	polling	Implements analysis algorithms for exhaustive polling systems where the
polling_qsys_gated	polling	Implements analysis algorithms for gated polling systems where the server
qbd_bmapbmap1	mam	Analyzes batch arrival and service systems using QBD matrix methods
qbd_mapmap1	mam	Analyzes MAP/MAP/1 queueing systems using QBD matrix analytic methods
qbd_r	mam	QBD R-matrix computation algorithms
qbd_r_logred	mam	Computes QBD R-matrix using logarithmic reduction method for numerical stability
qbd_raprap1	mam	Analyzes RAP/RAP/1 queueing systems using QBD methods with rational arrival processes
qbd_rg	mam	Computes fundamental R and G matrices for QBD analysis of MAP/MAP/1 queues
qbd_setupdelayoff	mam	Analyzes queueing systems with server setup delays and switch-off mechanisms
qsys_gg1	qsys	Provides comprehensive analysis of G/G/1 queues with general arrival and service processes
qsys_gig1_approx_allencunneen	qsys	Implements the widely-used Allen-Cunneen approximation for general G/G/1 queueing
qsys_gig1_approx_gelenbe	qsys	G/G/1 queue approximation using Gelenbe's method
qsys_gig1_approx_heyman	qsys	Analyzes a G/G/1 queueing system using Heyman's approximation
qsys_gig1_approx_kimura	qsys	G/G/1 queue approximation using Kimura's method
qsys_gig1_approx_klb	qsys	Analyzes a G/G/1 queueing system using the Kramer-Langenbach-Belz (KLB) approximation
qsys_gig1_approx_kobayashi	qsys	Analyzes a G/G/1 queueing system using Kobayashi's approximation
qsys_gig1_approx_marchal	qsys	Analyzes a G/G/1 queueing system using Marchal's approximation
qsys_gig1_approx_myskja	qsys	G/G/1 queue approximation using Myskja's method

Table B.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
qsys_gig1_approx_myskja2	qsys	G/G/1 queue approximation using enhanced Myskja's method
qsys_gig1_lbnd	qsys	G/G/1 queue lower bounds
qsys_gig1_ubnd_kingman	qsys	Calculates an upper bound on the waiting time for a G/G/1 system using Kingman's formula
qsys_gigk_approx	qsys	Analyzes a G/G/k queueing system using an approximation method
qsys_gigk_approx_cosmetatos	qsys	G/G/k queue approximation using Cosmetatos method
qsys_gigk_approx_kingman	qsys	Analyzes a G/G/k queueing system using Kingman's approximation
qsys_gigk_approx_whitt	qsys	G/G/k queue approximation using Whitt's method
qsys_gm1	qsys	G/M/1 Queueing System Analysis
qsys_mg1	qsys	Implements the Pollaczek-Khinchine formula for M/G/1 queues with Poisson arrivals
qsys_mg1_prio	qsys	M/G/1 queueing system with non-preemptive class priorities
qsys_mg1k_loss	qsys	M/G/1/K loss probability calculation
qsys_mg1k_loss_mgs	qsys	M/G/1/K loss probability using MacGregor Smith approximation
qsys_mginf	qsys	M/G/inf queue analysis (infinite servers)
qsys_mm1	qsys	Implements exact analytical solutions for the M/M/1 queue (Poisson arrivals, exponential
qsys_mm1k_loss	qsys	M/M/1/K loss probability calculation
qsys_mmk	qsys	Implements exact analytical solutions for M/M/k queues with Poisson arrivals,
qsys_mapg1	qsys	Analyzes MAP/G/1 queues with MAP arrivals and general service times using BUTools
qsys_mapm1	qsys	MAP/M/1 queueing system analysis with MAP arrivals and exponential service
qsys_mapmap1	qsys	Analyzes MAP/MAP/1 queues with MAP arrivals and MAP service times
qsys_mapmc	qsys	MAP/M/c multiserver queueing system analysis with MAP arrivals
qsys_mapph1	qsys	Analyzes MAP/PH/1 queues with MAP arrivals and phase-type service distributions
qsys_phph1	qsys	Analyzes PH/PH/1 queues with phase-type arrivals and service distributions
randp	mam	Provides random value selection based on relative probability distributions
rl_env	rl	Provides a reinforcement learning environment interface for queueing networks
rl_env_general	rl	Provides a general reinforcement learning environment for queueing networks
rl_td_agent	rl	Implements a temporal difference learning agent for queueing network control
rl_td_agent_general	rl	Implements a general-purpose temporal difference learning agent for queueing
sn_deaggregate_chain_results	sn	Calculate class-based performance metrics for a queueing network based on performance measures of its chains

Table B.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
`sn_get_arv_r_from_tput`	sn	Calculates the average arrival rates at each station from the network throughputs
`sn_get_demands_chain`	sn	Calculate new queueing network parameters after aggregating classes into chains
`sn_get_node_arv_r_from_tput`	sn	API function for sn operations
`sn_get_node_tput_from_tput`	sn	APIs to process NetworkStruct objects
`sn_get_product_form_chain_params`	sn	Calculate the parameters at class and chain level for a queueing network model
`sn_get_product_form_params`	sn	Extracts essential parameters (service demands, populations, visit ratios) from
`sn_get_residt_from_respt`	sn	Calculates the residence times at each station from the response times
`sn_get_state_aggr`	sn	Aggregates the state of the network
`sn_has_class_switching`	sn	Checks if the network uses class-switching
`sn_has_closed_classes`	sn	Checks if the network has one or more closed classes
`sn_has_dps`	sn	Check if the network includes a node with DPS scheduling
`sn_has_dps_prio`	sn	Check if the network includes a node with DPSPRIO scheduling
`sn_has_fb`	sn	Check if the network includes a node with FB (LAS) scheduling
`sn_has_fcfs`	sn	Identifies queueing networks using First-Come-First-Served scheduling disciplines
`sn_has_fork_join`	sn	Checks if the network uses fork and/or join nodes
`sn_has_fractional_populations`	sn	Checks if the network has closed classes with non-integer populations
`sn_has_gps`	sn	Check if the network includes a node with GPS scheduling
`sn_has_gps_prio`	sn	Check if the network includes a node with GPSPRIO scheduling
`sn_has_hol`	sn	Check if the network includes a node with HOL scheduling
`sn_has_homogeneous_scheduling`	sn	Checks if the network uses an identical scheduling strategy at every station
`sn_has_inf`	sn	Check if the network includes a node with INF scheduling
`sn_has_lcfs`	sn	Check if the network includes a node with LCFS scheduling
`sn_has_lcfs_pr`	sn	Check if the network includes a node with LCFSPR scheduling
`sn_has_lcfs_pi`	sn	Check if the network includes a node with LCFSPI scheduling
`sn_has_lept`	sn	Check if the network includes a node with LEPT scheduling
`sn_has_ljf`	sn	Check if the network includes a node with LJF scheduling
`sn_has_lrpt`	sn	Check if the network includes a node with LRPT scheduling
`sn_has_load_dependence`	sn	Checks if the network has a station with load-dependent service process
`sn_has_mixed_classes`	sn	Checks if the network has both open and closed classes
`sn_has_multi_chain`	sn	Check if the network has multiple chains
`sn_has_multi_class`	sn	Identifies queueing networks with multiple job classes, which require specialized
`sn_has_multi_class_fcfs`	sn	API function for sn operations
`sn_has_multi_class_heter_exp_fcfs`	sn	Checks if the network has one or more stations with multiclass heterogeneous FCFS
`sn_has_multi_class_heter_fcfs`	sn	Checks if the network has one or more stations with multiclass heterogeneous FCFS

Continued on next page

Table B.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
sn_has_multi_server	sn	Check if the network includes a multi-server node
sn_has_multiple_closed_classes	sn	Checks if the network has one or more closed classes
sn_has_open_classes	sn	Checks if the network has one or more open classes
sn_has_polling	sn	Check if the network includes a node with polling
sn_has_priorities	sn	Checks if the network uses class priorities
sn_has_product_form	sn	Determines if a queueing network has a known product-form solution by validating
sn_has_product_form_not_het_fcfs	sn	Checks if the network satisfies product-form assumptions (does not have heterogeneous FCFS)
sn_has_ps	sn	Check if the network includes a node with PS scheduling
sn_has_ps_prio	sn	Check if the network includes a node with PSPRIO scheduling
sn_has_psjf	sn	Check if the network includes a node with PSJF scheduling
sn_has_sept	sn	Check if the network includes a node with SEPT scheduling
sn_has_single_chain	sn	Check if the network has a single chain
sn_has_single_class	sn	Check if the network has a single class
sn_has_siro	sn	Check if the network includes a node with SIRO scheduling
sn_has_sjf	sn	Check if the network includes a node with SJF scheduling
sn_has_srpt	sn	Check if the network includes a node with SRPT scheduling
sn_is_closed_model	sn	Identifies closed queueing network models with finite job populations and no external
sn_is_mixed_model	sn	Checks if the network is a mixed model
sn_is_open_model	sn	Identifies open queueing network models with external arrivals and infinite
sn_is_population_model	sn	Checks if the model is a population model (only specific scheduling strategies without priorities or fork-join)
sn_is_state_valid	sn	Stochastic Network State Validation Utility
sn_print	sn	Prints comprehensive information about a NetworkStruct
sn_print_routing_matrix	sn	Prints the routing matrix of the network, optionally for a specific job class
sn_refresh_visits	sn	Stochastic Network Visit Ratio Calculator
sn_rtnodes_to_rtorig	sn	Converts routing matrices from nodes to original format, specifically handling class switching nodes
trace_mean	trace	Computes the arithmetic mean of empirical trace data. Fundamental statistical
trace_skew	trace	Computes the skewness of the trace data using Apache Commons Math
trace_var	trace	Computes sample variance and related statistics for empirical trace data
wf_analyzer	wf	Provides comprehensive workflow analysis capabilities including pattern
wf_auto_integration	wf	Provides automatic integration capabilities for workflow analysis with
wf_branch_detector	wf	Implements algorithms for detecting branching patterns in workflow traces
wf_loop_detector	wf	Implements algorithms for detecting loop and iterative patterns in workflow

Table B.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
`wf_parallel_detector`	wf	Implements algorithms for detecting parallel execution patterns in workflow
`wf_pattern_updater`	wf	Implements dynamic pattern updating algorithms for workflow analysis
`wf_sequence_detector`	wf	Implements algorithms for detecting sequential patterns in workflow traces

Index