LINE: Performance and Reliability Analysis Engine

User manual

Version: 2.0.0-ALPHA

Last revision: August 27, 2018

Contents

Chapter 1

Introduction

1.1 What is LINE?

LINE is an engine for performance and reliability analysis of systems based on queueing theory and stochastic modeling. Systems may either be software applications, business processes, computer networks, or else. LINE transforms 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. A key feature of LINE is that the engine decouples the model description from the solvers used for its solution. That is, the engine implements model-to-model transformations that automatically translate the model specification into the input format (or data structure) accepted by the target solver.

Target models are extended queueing networks, open or closed, and layered queueing networks. These models can be solved either via external solvers or native solvers. External solvers include Java Modelling Tools (JMT; http://jmt.sf.net) and LQNS (http://www.sce.carleton.ca/rads/lqns/). Featured native model solvers are based on:

- Continuous-time Markov chains (CTMC)
- Fluid ordinary differential equations (FLUID)
- Matrix analytic methods (MAM)
- Normalizing constant analysis (NC)
- Mean-value analysis (MVA)

Techniques implemented in these solvers include both exact and approximate solution methods and typically differ for accuracy, computational cost, and the subset of model features they support.

The above techniques can be applied to models that are either specified in MATLAB or loaded from external file formats. LINE 2.0.0-ALPHA is able to accept models specified in one of the following formats:

- LINE *modeling language (MATLAB script format)*. This is a MATLAB-based object-oriented language designed to closely resemble the abstractions available in JMT's queueing network simulator (JSIM). Among the main benefits of this language is the possibility to seamlessly transform LINE models into JMT simulation models and vice-versa. That is, LINE models can be loaded and visualized within JSIMgraph and similarly JSIMgraph models can be automatically imported into LINE.

- *Layered queueing network models (LQNS XML format).* LINE is able to solve a sub-class of LQN models, provided that they are specified using 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 JMT.
- *Performance Model Interchange Format (PMIF XML format).* LINE is able to import and solve closed queueing network models specified using PMIF v1.0.

1.2 Obtaining the latest release

This document contains the user manual for LINE version 2.0.0-ALPHA. The tool can be obtained from:

https://github.com/line-solver/line/

Contrary to earlier versions, LINE 2.0.0-ALPHA is released only as a MATLAB binary (compiled .p files). The distribution has been tested on MATLAB R2018a and R2017b and requires the *Statistics and Machine Learning Toolbox*. If you are interested to obtain a JAR, or an executable distribution for anyone of the operating systems supported by the MATLAB Compiler Runtime (MCR), please contact the maintainer.

1.3 Installation and getting started

This is the fastest way to get started with LINE:

1. Download the latest release
   - Git repository: https://github.com/line-solver/line/

   Ensure that files are cloned (or decompressed) into the desired installation folder.
2. Start MATLAB and change the active directory to the installation folder. Then add all LINE folders to the path

```
addpath(genpath(pwd))
```

3. Run the demonstrators using

```
allExamples
```

1.4 Getting help

For bugs or feature requests, please use: https://github.com/line-solver/line/issues

1.5 References

To cite LINE, we recommend to reference:

- J. F. Pérez and G. Casale. "LINE: Evaluating Software Applications in Unreliable Environments", in *IEEE Transactions on Reliability*, 2017.

The following papers discuss recent applications of LINE:

- C. Li and G. Casale. "Performance-Aware Refactoring of Cloud-based Big Data Applications", in Proceedings of 10th IEEE/ACM International Conference on Utility and Cloud Computing, 2017. *This paper uses* LINE *to model stream processing systems*.
- D. J. Dubois, G. Casale. "OptiSpot: minimizing application deployment cost using spot cloud resources", in *Cluster Computing*, Volume 19, Issue 2, pages 893-909, 2016. *This paper uses* LINE *to determine bidding costs in spot VMs*.
- R. Osman, J. F. Pérez, and G. Casale. "Quantifying the Impact of Replication on the Quality-of-Service in Cloud Databases'. Proceedings of the IEEE International Conference on Software Quality, Reliability and Security (QRS), 286-297, 2016. *This paper uses* LINE *to model the Amazon RDS database*.
- C. Müller, P. Rygielski, S. Spinner, and S. Kounev. Enabling Fluid Analysis for Queueing Petri Nets via Model Transformation, Electr. Notes Theor. Comput. Sci, 327, 71–91, 2016. *This paper uses* LINE *to analyze Descartes models used in software engineering*.
- J. F. Pérez and G. Casale. "Assessing SLA compliance from Palladio component models," in Proceedings of the 2nd Workshop on Management of resources and services in Cloud and Sky computing (MICAS), IEEE Press, 2013. *This paper uses* LINE *to analyze Palladio component models used in model-driven software engineering*.

1.6 Contact

Project coordinator and maintainer contact:

Giuliano Casale
Department of Computing
180 Queen's Gate
Imperial College London

Web: http://wp.doc.ic.ac.uk/gcasale/

1.7 Copyright and license

1.8 Acknowledgement

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

Chapter 2

Network models

Systems can be modeled in LINE using one of two available classes of stochastic models: `Network` and `LayeredNetwork` models, each specified by a corresponding MATLAB class. `Network` objects are extended queueing networks. Instead, `LayeredNetwork` objects are layered queueing networks, i.e., models consisting of layers, each corresponding to a queueing network, which interact through synchronous and asynchronous calls. Technical background on these models can be found in books such as [**?**, **?**] or in tutorials [**?**, **?**, **?**].

Throughout this chapter, we discuss the specification of `Network` models. Instead, we point to the following chapters for `LayeredNetwork` models. It is important to keep in mind that not all of the model features described in the present chapter are supported by every LINE solver. However, upon calling a solver, LINE will automatically detect if the supplied model can be analyzed by that solver and return an empty set if not. We point to the solvers chapter for tables summarizing the features supported by each solver.

2.1 Network object definition

2.1.1 Creating a network of resources

A queueing network can be described in LINE using its native modelling language, which decouples model specification from its solution. To get started, we can create our first queueing network model, called `myModel`, as follows

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

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

A node is a resource in the network that can be visited by a job. A node can either be *stateful* or *stateless*, where the notion of state includes the ability of a node to track the identity of jobs that are spending time inside it. If jobs are allowed to spend time in a stateful node, this is also said to be a *station*. For example: a sink is a stateless node; a first-come first-served queue is a station; a round-robin router is a stateful node, since it needs to keep track in its state variables of the last destination a job has been sent to, but it is not a station since dispatching is immediate.

Stations inherit from the `Station` class. In particular, the `QueueingStation` class specifies a queueing system from its name and scheduling strategy, e.g.

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

Valid scheduling strategies are specified within the `SchedStrategy` static class and include:

- First-come first-served (`SchedStrategy.FCFS`)
- Infinite-server (`SchedStrategy.INF`)
- Processor-sharing (`SchedStrategy.PS`)
- Discriminatory processor-sharing (`SchedStrategy.DPS`)
- Generalized processor-sharing (`SchedStrategy.GPS`)
- Shortest expected processing time (`SchedStrategy.SEPT`)
- Shortest job first (`SchedStrategy.SJF`)
- Head-of-line priority (`SchedStrategy.HOL`)

Infinite-server stations may be instantiated as either stations with the `SchedStrategy.INF` strategy or using the following specialized constructor

```
delay = DelayStation(model, 'ThinkTime');
```

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 more details.

2.1.2 Job classes

Jobs travel within the network placing service demands at the stations. 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. The demand placed by a job at a station depends on the class of the job.

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

```
class1 = OpenClass(model, 'Class1');
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. A network can include at most a single `Source` and a single `Sink`. LINE does not require to associate these nodes explicitly with the open classes, as this is done automatically. However, the LINE language requires to explicitly create these nodes as the routing topology needs to specify the entry and exit points of jobs.

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 class *reference station* (e.g., `node{1}`):

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

The notion of reference station is introduced in subsection 2.1.2 below.

The reference station of an open class is always automatically chosen by LINE to be the `Source`. Moreover, if the network includes only closed classes, there is no need to instantiate `Source` and `Sink`.

Reference station

Above we have shown that the specification of classes requires to choose a reference station. The reference station is a node used to calculate certain performance indexes, called *system performance indexes*, associated to *chains*. A chain defines the set of reachable classes for a job that starts in the given class r and over time changes class according to a class-switching routing strategy, which is discussed later in this section. Since class switching in LINE does not allow a class to become open if it is closed, and vice-versa, chains can also be classified into open and closed, depending on the classes that compose them.

For example, the system throughput for a chain is defined as sum of the arrival rates at the reference station for all classes in that chain. In the case of open chains, the reference station is always the `Source` and its arrival rate corresponds to the flow of jobs that return to the infinite pool in the external world through the `Sink`. If there is no class switching, each chain contain a single class, thus the per-chain and per-class performance indexes are identical.

Class priorities

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

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

2.1.3 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`). 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 `linkNetwork` function. This function will also automatically add nodes to the network to ensure the correct switching of class for jobs in transit between nodes (`ClassSwitch` nodes).

For example, consider a network with two classes and four nodes where: (i) class one visits node 1, then one at random between nodes 2,3, and 4, and finally cycles back to node 1; (ii) class two cycles through stations 1 to 4 in a sequence. We can instantiate this routing topology as follows:

```
P{class1} = zeros(4);
P{class1}(1,2:4) = [1/3, 1/3, 1/3];
P{class1}(2:4,1) = 1.0;
P{class2} = circul(4); % class 2 visits stations in order
model.linkNetwork(P);
```

When used as arguments to a cell array or matrix, class objects will be replaced by a corresponding numerical index. In LINE, the indexing of classes and nodes corresponds to the order in which they are

instantiated in the model. The `getClassIndex` and `getNodeIndex` functions return the numeric index associated to a node name, e.g., `model.getNodeIndex('Delay')`. Class and node names in a network need to be unique. The list of used names 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 than between stations, which means that elements such as the `Source` and the `Sink` are explicitly considered in the routing matrix. Note that the routing matrix assigned to a model can be summarized using the `printRoutingMatrix` function.

Other routing strategies

The above routing specification style is for probabilistic routing strategies at all nodes. A different style should be used for scheduling policies that do not require to explicit routing probabilities or when only some of the nodes use a probabilistic strategy. Currently supported strategies include:

- Round robin (`RoutingStrategy.RR`). This is a non-probabilistic 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.

For such 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, e.g.,

```
queue.setRouting(class1,RoutingStrategy.RR);
```

assigns equal probability to all outgoing links from node 1. In this specification style, for nodes with probabilistic routing strategies one should instead use

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

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

2.1.4 Class switching

In LINE, jobs can switch class while they travel between stations (including self-loops on the same station). This feature can be used to model queueing properties such as re-entrant lines, i.e., chains in which a job visiting a station a second time may require a different average demand than at its first visit. 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. The strategy to describe the class switching mechanism is integrated in the specification of the routing between stations.

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 example, assume that in the previous example a job in class 1 can switch into class 2 after departing node 4, remaining there forever. We can specify this routing strategy as follows:

```
P = cellzeros(model.getNumberOfClasses,model.getNumberOfNodes);
P{class1,class2}(4,1) = 1.0; % routing from node 4 to 1 switching from ...
    class 1 to class 2
P{class1,class1}(1,2:4) = [1/3, 1/3, 1/3]; P{1,1}(2:3,1) = 1.0; % routing ...
    for class 1 (no switching)
P{class2,class2} = circul(4); % class 2 visits nodes in a cyclic order (no ...
    switching)
model.linkNetwork(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, so that all performance metrics at the node refer to the class of the job upon arrival.

Depending on the specified probabilities, a job will be able to switch class among a subset of the available classes. Each subset is called a *chain*. Chains are determined 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 their member classes.

Class switching with non-probabilistic routing strategies

In the presence of non-probabilistic routing strategies, one needs to manually specify the class switching mechanism. This can be done through addition to the network topology of `ClassSwitchNode` elements. The constructor of this node requires to specify a probability matrix C such that $C(r, s)$ is the probability that a job of class r arriving into the `ClassSwitchNode` 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

```
C = [0, 1; 0, 1];
queue = ClassSwitchNode(self, 'CSNode',C);
```

Note that for a network with M stations, up to M^2 `ClassSwitchNode` elements may be required to implement class-switching across all possible links, including self-loops. Moreover, refreshing network parameters under non-probabilistic routing strategies may .

Contrary to the `linkNetwork` function, one cannot specify in the argument to `setProbRouting` a class switching probability. The `setProbRouting` function should instead be used to route the job through an appropriate `ClassSwitchNode` element.

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.

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

```
A = [10,20]; % A(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(A,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`, `Network.cyclicPsInf`, `Network.cyclicFcfs`, and `Network.cyclicFcfsInf`. These functions only require to replace 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)
```

2.1.5 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. Note that 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,

```
example_closedModel_3
delay.setChainCapacity([1,1])
model.refreshCapacity()
```

creates an example model with two chains and three classes (specified in `example_closedModel_3.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 `example_closedModel_3` 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 need for

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

2.1.6 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 `PhaseType` offers distributions that are analytically tractable, which are defined upon certain absorbing Markov chains consisting of one or more states (*phases*) that are called phase-type distributions. They include as special case the following distributions supported in LINE, along with their respective constructors:

- Exponential distribution: `Exp`(λ), where λ is the rate of the exponential
- n-phase Erlang distribution: `Erlang`(α, n), where α is the rate of each of the n exponential phases
- 2-phase hyper-exponential distribution: `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.
- 2-phase Coxian distribution: `Cox2`(μ_1, μ_2, ϕ_1), which assigns rates 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 $\phi_2 = 1.0$).

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.fitMoments(0.2,10));
```

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.fitMoments(0.1));
```

where we have used a single parameter in `fitMoments` since the exponential distribution does not allow to choose the SCV.

Non-Markovian distributions are also available, but typically they restrict the available network analysis techniques to simulation. They include the following distributions:

- Deterministic distribution: `Det`(μ) assigns probability 1.0 to the value μ.
- Uniform distribution: `Uniform`(a, b) assigns uniform probability $1/(b - a)$ to the interval $[a, b]$.
- Gamma distribution: `Gamma`(α, k) assigns a gamma density with shape α and scale k.
- Pareto distribution: `Pareto`(α, k) assigns a Pareto density with shape α and scale k.

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 in models with sparse routing matrices, both to ensure an efficient model solution and 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). Typically, LINE solvers will replace zero service times with small positive values ($\varepsilon = 10^{-7}$).

Fitting a distribution

The `fitMoments` function is available for all distributions that inherit from the `PhaseType` class. This function provides exact or approximate matching of the requested 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.fitMoments(1,0.75)` will return the closest approximation, e.g., a 2-phase Erlang (SCV=0.5) with unit mean. The Erlang distribution also offer 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, `fitMoments` 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.fitMoments` automatically chooses to return a probability of selecting phase 1 equal to 0.99, as this spends the degree of freedom corresponding to the (unspecified) third moment of the distribution. Compared to other choices, this particular assignment corresponds to an higher probability mass in the tail of the distribution.

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 `PhaseType` 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 `getRenewalProcess` function. The result will be a cell array where element $k+1$ corresponds to matrix D_k.

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 inherit from the `PointProcess` class, and include the 2-state Markov-modulated Poisson process (`MMPP2`) and empirical traces read from files (`Replayer`).

In particular, 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 `PhaseType` class.

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.fitMoments(0.2,10), 5.0);
```

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

2.1.7 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
```

An example is shown in Figure Figure 2.1 below. 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 transforming it into a MATLAB 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 next figures 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`

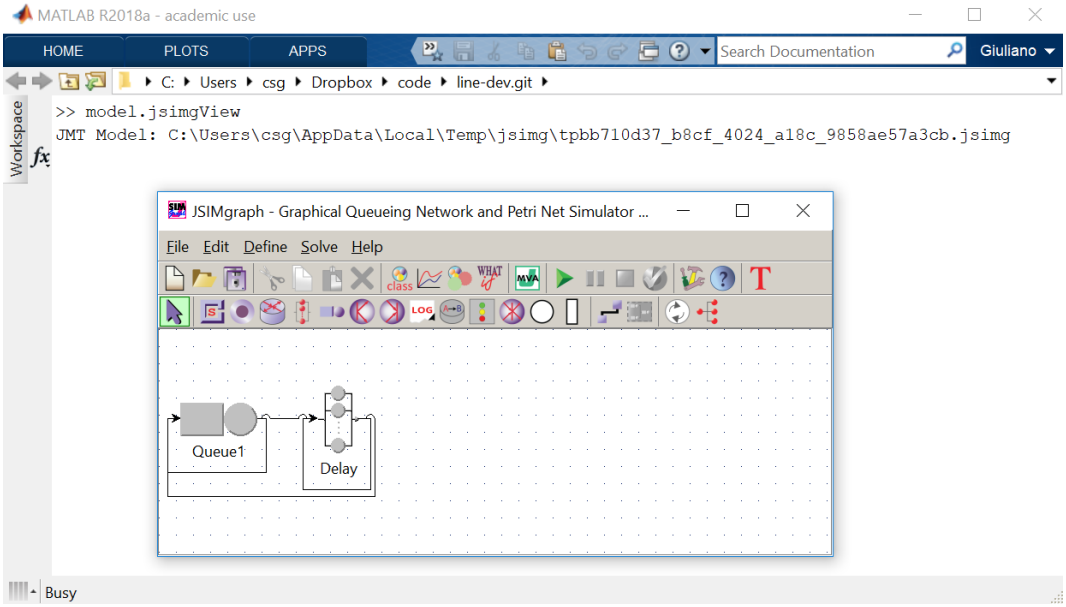

Figure 2.1: jsimgView function

are drawn as returns to the `Source`. The node topology on the right, illustrates all nodes, including certain `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'     'DelayStation'          'inf'       5           1
    'Queue1'    'QueueingStation'       'ps'        0           2
```

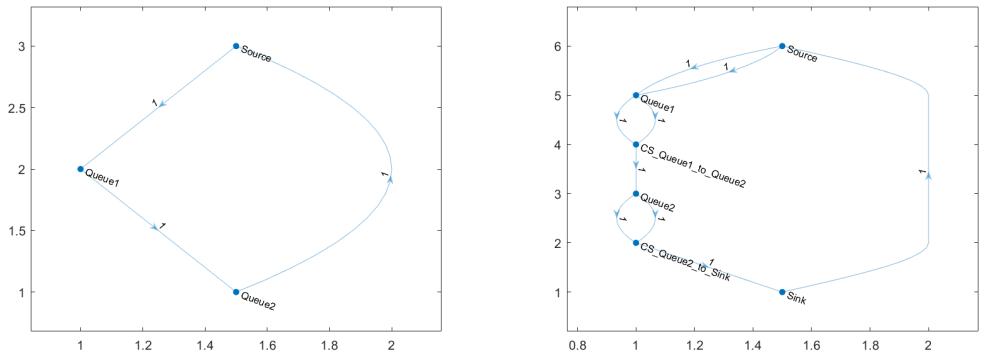

Figure 2.2: `getGraph` function: station topology (left) and node topology (right) for a 2-class tandem queueing network with class-switching.

```
>> 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 source node.

2.1.8 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.
- `PMIF2LINE` imports a XML file containing a PMIF 1.0 model.

Both scripts require in input the filename and desired model name, and return a single output, e.g.,

```
qn = PMIF2LINE([pwd,'\\examples\\data\\PMIF\\pmif_example_closed.xml'],'Mod1')
```

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

`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,

```
example_closedModel_1; LINE2SCRIPT(model, 'script.m')
```

creates a new file `script.m` with code

```
model = Network('model');
queue = DelayStation(model, 'Delay');
delay = QueueingStation(model, 'Queue1', SchedStrategy.PS);
delay.setNumServers(1);
class1 = ClosedClass(model, 'ClosedClass1', 5, queue, 0);
queue.setService(class1, Cox2.fitMoments(1.000000,1.000000));
delay.setService(class1, Cox2.fitMoments(2.000000,1.000000));
P = cell(1);
P{1,1} = [0.7 0.3;1 0];
model.linkNetwork(P);
```

that is equivalent to the model specified in `example_closedModel_1.m`.

2.1.9 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.,

```
LINE2SCRIPT(JMT2LINE('myModel.jsimg'),'myModel.m')
```

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

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

2.2 Steady-state analysis

2.2.1 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 sections.

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 = SolverJMT(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.getAvgQLen()
% UN(i,r): utilization of class r at station i
UN = solver.getAvgUtil()
% RN(i,r): mean response time of class r at station i (summed on visits)
RN = solver.getAvgRespT()
% TN(i,r): mean throughput of class r at station i
TN = solver.getAvgTput()
```

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

```
[QN,UN,RN,TN] = solver.getAvg()
```

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 with the following method

```
AvgTable = solver.getAvgTable()
```

It is also possible to aggregate results at chain level using the `getAvgByChain` method.

2.2.2 Station response time distribution

`SolverFluid` 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 = SolverFluid(model);
FC = solver.getCdfRespT();
plot(FC{2,1}(:,2),FC{2,1}(:,1)); xlabel('t'); ylabel('Pr(RespT<t)');
```

2.2.3 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 `getAvgSysByChain` and `getAvgSysByChainTable` 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.

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 in some applications one may want to avoid counting each visit as a completion of the visit to the system. In such cases, LINE allows 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.linkNetwork(P);
```

Note that LINE does not allow a chain to complete at heterogeneous stations, therefore the `completes` property of a class always refers to the reference station for the chain.

2.3 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 native support in LINE for state

modeling.

2.3.1 Station states

We begin by explaining how to specify a state s_0. State modelling is supported only for stations with scheduling policies that depend on the number of jobs running or waiting at the node. For example, it is not supported for shortest job first (`SchedStrategy.SJF`) scheduling, in which state depends on the service time samples for the jobs.

Suppose that the network has R classes and that service distributions are phase-type, i.e., that they inherit from `PhaseType`. 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 $c_1 = 0$.
- 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 `PhaseType`. 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. 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 2.1: State descriptors for Markovian scheduling policies

Sched. strategy	**Station state vector**	**State condition**
`EXT`	$[\texttt{Inf}, s_{11}, ..., s_{1K_1}, ..., s_{R1}, ..., s_{RK_R}]$	$\sum_k s_{rk} = 1, \forall r$
`FCFS`, `HOL`, `LCFS`	$[c_b, ..., c_1, s_{11}, ..., s_{1K_1}, ..., s_{R1}, ..., s_{RK_R}]$	$\sum_r \sum_k s_{rk} = 1$
`SEPT`, `RAND`	$[b_1, ..., b_R, s_{11}, ..., s_{1K_1}, ..., s_{R1}, ..., s_{RK_R}]$	$\sum_r \sum_k s_{rk} = 1$
`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 of resident jobs.

- `State.toMarginal`: extracts 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 `example_closedModel_4.m`. 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:

```
>> example_closedModel_4;
>> space = State.fromMarginalAndRunning(model,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,3,[2,1,1,1],[2,1,0,0])
space =
     4     3     2     1     0     0     0
     3     4     2     1     0     0     0
```

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,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 state to a station

Given a single or multiple states, it is possible to assign the initial state to a station using the `setState` function on that station's object. To cope with multiple states, 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 analyze the network from all those states and weight the results according to the prior probabilities. The default prior value assigned probability 1.0 to the *first* specified state. The functions `setStatePrior` and `getStatePrior` of the `Station` class can be used to check and change the prior probabilities for the supplied initial states.

2.3.2 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 with elements $\texttt{n}(i, r)$ specifying the total number of resident class-r jobs at node i and the latter two require a matrix $\texttt{s}(i, r)$ 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 server of reference stations are occupied in order of class id, i.e., jobs in the firstly created class are assigned to the server in phase 1, then spare servers are allocated to the second class in phase 1, and so forth;
- 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.

Lastly, the `initFromAvgQLen` 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.

2.3.3 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 position time $t' > t$ without ever being visited again by any job. `LINE` allows one 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 an heuristic whereby a class is considered transient if it has fewer events 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.

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

2.4.1 Computing transient averages

At present, LINE supports only transient computation of queue-lengths, throughputs and utilizations using the `CTMC` and `FLUID` solvers. Transient response times are not currently supported, as they do not always obey Little's law.

The computation of transient metrics proceeds similarly to the steady-state case. We first obtain the handles for transient averages:

```
[Qt,Ut,Tt] = model.getTransientHandlers();
```

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

```
[QNt,UNt,TNt] = solver{s}.getTransientAvg(Qt,Ut,Tt)
plot(QNt{1,1}(:,2), QNt{1,1}(:,1))
```

The transient average queue-length at node i for class r is stored within `QNt{i,r}`.

Note that the above code does not show how to specify a maximum time t for the output time series. This can be done using the `timespan` field of the options, as described later in the solvers chapter.

2.4.2 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 `getTransientCdfRespT`, implemented by `SolverJMT` 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 `getTransientCdfFirstPassT` 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 `SolverFluid` and has a similar syntax as `getCdfRespT`.

2.5 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 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 several models.

2.5.1 Internal representation of the model structure

For efficiency reasons, once a user requests to solve a `Network`, LINE calls internally generates 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 `QN` class, which describes an extended multiclass queueing network with class-switching and Coxian service times. The representation can be obtained as follows

```
qn = model.getStruct()
```

The table below presents the properties of the `QN` class.

2.5.2 Fast parameter update

Successive invocations of `getStruct()` will return a cached copy of the `QN` 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 `QN` representation, once the modification has been applied to the objects (e.g., stations, classes, ...) used to define the network. These functions are as follows:

Table 2.2: QN 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
classname{r}	string	Name of class r
classprio(r)	integer	Priority of class r (0 = highest priority)
csmask(r, s)	logical	true if class r can switch into class s at some node
isstation(i)	logical	true if node i is a station
isstateful(i)	logical	true if node i is a stateful node
mu{i, r}(k)	double	Coxian 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.
nchains	integer	Number of chains in the network
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
nservers(i)	integer	Number of servers at station i
nstations	integer	Number of stations in the network
nstateful	integer	Number of stateful nodes in the network
nodenames{i}	string	Name of node i
nodetypes{i}	string	Type of node i (e.g., NodeType.Sink)
nvars	integer	Number of local state variables at stateful nodes
phases(i, r)	integer	Number of phases for service process of class r at station i
phi{i, r}(k)	double	Coxian completion 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)
refstat(r)	integer	Index of reference station for class r
rt(idx_{ir}, idx_{js})	double	Probability of routing from station i to j, switching class from r to s where, e.g., $idx_{ir} = (i-1) *$ nclasses $+ r$.
rtnodes(idx_{ir}, idx_{js})	double	Same as rt, but i and j are nodes.
rtfun(st1, st2)	matrix	State-dependent routing table given initial (st1) and final (st2) state cell arrays. Table entries defined as in rt.
schedparam(i, r)	double	Parameter for class r strategy at station i
sched{i}	cell	Scheduling strategy at station i (e.g., SchedStrategy.PS)
schedid(i)	integer	Scheduling strategy id at station i (e.g., SchedStrategy.ID_PS)
sync{s}	struct	Data structure specifying a synchronization s among nodes
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 node i in class r
varsparam{i}	double	Parameters for local variable instantiation at stateful node i

- `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`, `schedid`, and `schedparam` fields.
- `refreshService`: 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.refreshService;
```

2.5.3 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');
queue = ClassSwitchNode(model,'CSNode',[0,1;0,1]);
delay = QueueingStation(model, 'Queue1', SchedStrategy.FCFS);
>> model.getNodes
ans =
  2x1 cell array
    {1x1 ClassSwitchNode}
    {1x1 QueueingStation}
>> model.resetNetwork
ans =
  1x1 cell array
    {1x1 QueueingStation}
```

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 `ClassSwitchNode` nodes as last before solving the model. This node list can be employed as usual to reinstantiate new stations or `ClassSwitchNode` nodes. The `addLink`, `setRouting`, and possibly the `setProbRouting` functions will also need to be re-applied as described in the previous sections.

2.5.4 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.

Chapter 3

Network solvers

3.1 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 sharing a similar overall solution strategy, they can differ significantly from each other in the way this is actually implemented and on wether the final solution is exact or approximate.

The `method` field in the options structure array passed to a solver can be used to select the desired method, e.g., `options.method='default'` requires the solver to use default options, while `options.method='exact'` requires to solve the model exactly, if an exact solution method is available.

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 2.0.0-ALPHA:

- `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 `SolverAuto` 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 `SolverCTMC` class.
- `FLUID`: 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 [**?**, **?**]. This solver is implemented by the `SolverFluid` class.
- `JMT`: This is a solver that uses a model-to-model transformation to export the LINE representation into a JMT simulation model [**?**]. This solver can analyze also non-Markovian models, in

particular those involving deterministic or Pareto distributions, or empirical traces. This solver is implemented by the `SolverJMT` 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 `SolverMAM` 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 `SolverMVA` 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 `SolverNC` class.
- `SSA`: This is a discrete-event simulation based on the CTMC representation of the model. It is fully implemented in MATLAB language and thus offer lower speed than JMT, but the model execution can be easily parallelized using MATLAB's *spmd* construct. This solver is implemented by the `SolverSSA` class.

3.2 Solvers

3.2.1 AUTO

The `SolverAuto` class provides interfaces to the core solution functions (e.g., `getAvg`, ...) that dynamically bind to one of the other solvers implemented in LINE (`CTMC`, `NC`, ...). It is often not possible 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. Such heuristics, though, are independent of the core function called, thus it is possible that the optimal solver does not support the specific function called (e.g., `getTranAvg`). In such cases `SolverAuto` determines what other solvers would be feasible and prioritizes them in execution time order, with the fastest one on average having the higher priority. Eventually, the solver will be always able to identify a solution strategy, through at least simulation-based solvers such as `JMT` or `SSA`.

3.2.2 CTMC

The `SolverCTMC` 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 `SolverFluid`.

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`(i, r) is the maximum number of jobs of class r that can be placed at station i.

3.2.3 FLUID

This solver is based on the system of fluid ordinary differential equations for INF-PS queueing networks presented in [**?**].

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.

3.2.4 JMT

The class is a wrapper for the `JMT` simulation and consists of a model-to-model transformation from the `Network` data structure into the JMT's input XML format (`.jsimg`) and a corresponding parser for JMT's results. In the transformation, 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.

Upon invocation, the `JMT` JAR archive will be searched in the MATLAB path and if unavailable automatically downloaded.

3.2.5 MAM

This is a basic solver for some Markovian open queueing systems that can be analyzed using matrix analytic methods. The solver at the moment is a basic wrapper for the BU tools library for matrix-analytic methods [**?**]. At present, it is not possible to solve a queueing network model using `SolverMAM`.

3.2.6 MVA

The solver is primarily based on the Bard-Schweitzer approximate mean value analysis (AMVA) algorithm (`options.method='default'`), but also offers and implementation of the exact MVA algorithm (`options.method='exact'`). Non-exponential service times in FCFS nodes are treated using a diffusion approximation [**?**]. Multi-server FCFS is dealt with using a slight modification of the Rolia-Sevcik method [**?**]. DPS queues are analyzed with a time-scale separation method, so that for an incoming job of class r and weight w_r, classes with weight $w_s \geq 5w_r$ are replaced by high-priority classes that are analyzed using the standard MVA priority approximation. Conversely the remaining classes are treated by weighting the queue-length seen upon arrival in class $s \neq r$ by the correction factor w_s/w_r.

3.2.7 NC

The `SolverNC` 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.

3.2.8 SSA

The `SolverSSA` class is a basic stochastic simulator for continuous-time Markov chains. It reuses some of the methods that underpin `SolverCTMC` 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 hashed. The state space is not generated upfront, but rather 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.

3.3 Supported language features and options

3.3.1 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

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

It is possible to programmatically check which solvers are available for a given model as follows

```
>> Solver.getAllFeasibleSolvers(model)
ans =
  1x6 cell array
    {1x1 SolverCTMC}    {1x1 SolverJMT}    {1x1 SolverSSA}    {1x1 ...
       SolverFluid}    {1x1 SolverMVA}    {1x1 SolverNC}
```

In the example, `SolverMAM` is not feasible for the considered model and therefore not returned. Note that `SolverAuto` is never included in the list returned by this methods since this is a wrapper for other solvers.

3.3.2 Class functions

The table below lists the steady-state and transient analysis functions implemented by the `Network` solvers. Since the features of the `AUTO` solver are the union of the features of the other solvers, in what follows it will be omitted from the description.

Table 3.1: Solver support for scheduling strategies

		Network Solver						
Function	**Regime**	CTMC	FLUID	JMT	MAM	MVA	NC	SSA
getAvg	Steady-state	✓	✓	✓	✓	✓	✓	✓
getAvgArvR	Steady-state	✓	✓	✓	✓	✓	✓	✓
getAvgByChain	Steady-state	✓	✓	✓	✓	✓	✓	✓
getAvgByChainTable	Steady-state	✓	✓	✓	✓	✓	✓	✓
getAvgQLen	Steady-state	✓	✓	✓	✓	✓	✓	✓
getAvgRespT	Steady-state	✓	✓	✓	✓	✓	✓	✓
getAvgSysByChain	Steady-state	✓	✓	✓	✓	✓	✓	✓
getAvgSysByChainTable	Steady-state	✓	✓	✓	✓	✓	✓	✓
getAvgTable	Steady-state	✓	✓	✓	✓	✓	✓	✓
getAvgTput	Steady-state	✓	✓	✓	✓	✓	✓	✓
getAvgUtil	Steady-state	✓	✓	✓	✓	✓	✓	✓
getCdfRespT	Steady-state		✓					
getProbState	Steady-state	✓					✓	
getProbStateSys	Steady-state	✓		✓			✓	
getTranAvg	Transient		✓	✓				
getTranCdfPassT	Transient		✓					
getTranCdfRespT	Transient			✓				
getTranState	Transient			✓				
getTranStateSys	Transient			✓				

The functions listed above with the `Table` suffix (e.g., `getAvgTable`) provide results in tabular format corresponding to the corresponding core function (e.g., `getAvg`). The features of the core functions are as follows:

- `getAvg`: returns the mean queue-length, utilization, mean response time (for one visit), and throughput for each station and class.
- `getAvgByChain`: returns the mean queue-length, utilization, mean response time (for one visit), and throughput for every station and chain.
- `getAvgSysByChain`: 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.
- `getProbState`: returns joint and marginal state probabilities for jobs of different classes for each station at steady-state.
- `getProbStateSys`: returns joint probabilities for the system state for each class at steady-state.
- `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.
- `getTranState`: returns the transient marginal state for every stations and class from a given initial state.
- `getTranStateSys`: returns the transient marginal system state from a given initial state.

3.3.3 Scheduling strategies

The table below shows the supported scheduling strategies within LINE queueing stations. Each strategy belongs to a policy class: preemptive resume (`SchedPolicy.PR`), non-preemptive (`SchedPolicy.NP`), non-preemptive priority (`SchedPolicy.NPPrio`).

3.3.4 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.

3.3.5 Solver options

Solver options are encoded in LINE in a structure array that is internally passed to the solution algorithms. This can be specified as an argument to the constructor of the solver. For example, the following two constructor invocations are identical

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

Table 3.2: Solver support for scheduling strategies

		Network Solver						
Strategy	**Policy Class**	CTMC	FLUID	JMT	MAM	MVA	NC	SSA
FCFS	NP	✓	✓	✓	✓	✓	✓	✓
INF	NP	✓	✓	✓		✓	✓	✓
RAND	NP	✓		✓		✓	✓	✓
SEPT	NP	✓		✓				✓
SJF	NP			✓				
HOL	NPPrio	✓		✓				✓
PS	PR	✓	✓	✓		✓	✓	✓
DPS	PR	✓	✓	✓		✓		✓
GPS	PR	✓		✓				✓

Table 3.3: Solver support for statistical distributions

	Network Solver						
Distribution	CTMC	FLUID	JMT	MAM	MVA	NC	SSA
Cox2	✓	✓	✓	✓	✓	✓	✓
Exp	✓	✓	✓	✓	✓	✓	✓
Erlang	✓	✓	✓	✓	✓	✓	✓
HyperExp	✓	✓	✓	✓	✓	✓	✓
Disabled	✓	✓	✓	✓	✓	✓	✓
Det			✓				
Gamma			✓				
Pareto			✓				
Replayer			✓				
Uniform			✓				

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 = SolverJMT(model,'method','exact')
opt = SolverJMT.defaultOptions; opt.method='exact'; s = SolverJMT(model, opt)
```

Available solver options are as follows:

- `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 `FLUID` 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 `FLUID` 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 `FLUID`, this is a model-dependent vector with the values of all the variables used within the ODE system that underpins the fluid approximation.
- `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.
- `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.
- `timestamp` (`real interval`) requires the transient solver to produce a solution in the specified temporal range. If the value is set to `[Inf, Inf]` the solver will only return a steady-state solution. In the case of the `FLUID` solver and in simulation, `[Inf, Inf]` has the same computational cost of `[0, Inf]` therefore the latter is used as default.
- `verbose` controls the verbosity level of the solver. Supported levels are 0 for silent, 1 for standard verbosity, 2 for debugging.

3.4 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 the `JMT.jar` file under thr `'SolverJMT'` folder. This forces LINE to download the latest version of the JMT executable.
- Periodically running the `jmtCleanTempDir` script can help removing temporary by-products of the JMT solver. This is strongly encouraged under repeated uses of the `keep=1` option, as this stores on disk the temporary models sent to JMT.

Chapter 4

Layered network models

In this chapter, we present the definition of the `LayeredNetwork` class, which encodes the support in LINE for layered queueing networks. These models are extended queueing networks where servers, in order to process jobs, can issue synchronous and asynchronous calls among each other. The topology of call dependencies makes it possible to partition the model into a set of layers, each consisting of a subset of the resources. We point to [**?**] and to the LQNS user manual for an introduction [**?**].

4.1 LayeredNetwork object definition

4.1.1 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 service demands required at the underpinning processor, or calls to entries exposed by some remote tasks.

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', 1, SchedStrategy.INF).on(P2);
E1 = Entry(model, 'E1').on(T1);
E2 = Entry(model, 'E2').on(T2);
```

Here, 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 number of servers in the underpinning queueing system for `T1`). Note that both processors and tasks can be associated to the standard LINE scheduling strategies, with the exception that `SchedStrategy.REF` should be used to denote the reference task, which has a similar meaning to the reference node in the `Network` object.

4.1.2 Describing service times of entries

The service demands placed by an entry on the underpinning processor 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 of 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 final activity, which upon completion terminates the execution of the entry.

For example, we can associate an activity to each entry as follows:

```
A1 = Activity(model, 'A1', Exp(1.0)).on(T1).boundTo(E1).synchCall(E2,3.5);
A2 = Activity(model, 'A2', Exp(2.0)).on(T2).boundTo(E2).repliesTo(E2);
```

Here, `A1` is a task activity for `T1`, acts as initial activity for `E1`, consumes an exponential 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 service time on the processor underneath `T2` given by an exponential distribution with rate $\lambda = 2.0$.

At present, LINE 2.0.0-ALPHA supports only synchronous calls. Support for asynchronous calls is available in older versions, e.g. LINE 1.0.0. Extension of LINE 2.0.0-ALPHA to asynchronous calls

Activity graphs

Often, it is useful to structure the sequence of activities carried out by an entry in a graph. Currently, LINE supports this feature only for activities places in series. For example, we may replace the specification of the activities underpinning a call to `E2` as

```
A20 = Activity(model, 'A20', Exp(1.0)).on(T2).boundTo(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).

4.1.3 Debugging and visualization

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

```
plot(model)
```

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. As in the case of the `Network` class, the `getGraph` method can be called to inspect the structure of the `LayeredNetwork` object.

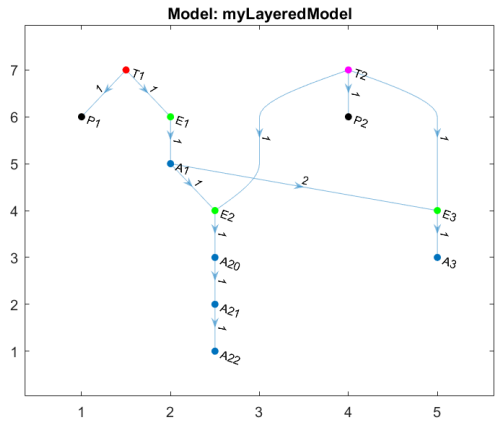

Figure 4.1: `LayeredNetwork.plot` method

Lastly, 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.

4.2 Decomposition into layers

Layers are a form of decomposition where the influence of resources not explicitly represented in that layer is taken into account through an artificial delay station, placed in a closed loop to the resources [**?**]. This artificial delay is used to model the inter-arrival time between calls from resources that belong to other layers.

4.2.1 Running a decomposition

The current version of LINE adopts SRVN-type layering [**?**], whereby a layer corresponds to one and only one resource, either a processor or a task. The only exception are reference tasks, which can only appear as clients to their processors. The `getLayers` method returns a cell array consisting of the `Network` objects corresponding to each layer

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

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`. 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.

4.2.2 Initialization and update

In general, the parameters of a layer will depend on the steady-state solution of an other layer, causing a cyclic dependence that can be broken only after the model is analyzed by a solver. In order

to assign parameters within each layer prior to its solution, the LayeredNetwork class uses the initDefault method, which sets the value of the artificial delay to simple operational analysis bounds [?].

The layer parameterization depends on a subset of performance indexes stored in a param structure array within the LayeredNetwork class. After initialization, it is possible to update the layer parameterization for example as follows

```
layers = model.getLayers();
for l=1:model.getNumberOfLayers()
    AvgTableByLayer{l} = SolverMVA(layers{l}).getAvgTable;
end
model.updateParam(AvgTableByLayer);
model.refreshLayers;
```

Here, the refreshParam method updates the param structure array from a cell array of steady-state solutions for the Network objects in each layer. Subsequently, the refreshLayers method enacts the new parameterization across the Network objects in each layer.

4.3 Solvers

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

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

```
>> AvgTable = SolverLQNS(model).getAvgTable
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 getAvgTable 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.

4.3.1 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='std'` or `'lqns'`: LQNS analytical solver with default settings.
- `options.method='exact'`: 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='srvnexact'`: the solver will execute the standard LQNS analytical solver with SRVN layering and the exact MVA method.
- `options.method='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 `SolverLQNS` will interrupt.

4.3.2 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 = SolverLN.defaultOptions;
mvaopt = SolverMVA.defaultOptions;
SolverLN(model, @(layer) SolverMVA(layer, mvaopt), options).getAvgTable
```

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`.

4.4 Model import and export

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

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

```
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.

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

Chapter 5

Random environments

Systems modeled with LINE can be described as operating in an environment whose state affects the way the system operates. To distinguish the states of the environment from the ones of the system within it, we shall refer to the former as the environment *stages*. In particular, LINE 2.0.0-ALPHA supports the definition of a class of random environments subject to three assumptions:

- The stage of the environment evolves independently of the state of the system.
- The dynamics of the environment stage can be described by a continuous-time Markov chain.
- The topology of the system is independent of the environment stage.

The above definitions are in particular appropriate to describe systems whose input parameters change with the environment stage. For example, an environment with two stages, say normal load and peak load, may differ for the number of servers that are available in a queueing station, i.e., the system controller may add more servers during peak load. Upon a stage change in the environment, the model parameters will instantaneously change, but the state reached during the previous stage will be used to initialize the system in the new stage.

Although in a number of cases the system performance may be similar to a weighted combination of the average performance in each stage, this is not true in general, especially if the system dynamic (i.e., the rate at which jobs arrived and get served) and the environment dynamic (i.e., the rate at which the environment changes active stage) happen at similar timescales [**?**].

5.1 Environment object definition

5.1.1 Specifying the environment transitions

To specify an environment, it is sufficient to define a cell array with entries describing the distribution of time before the environment jumps to a given target state. For example

```
env{1,1} = Exp(0);
env{1,2} = Exp(1);
env{2,1} = Exp(1);
env{2,2} = Exp(0);
```

describes an environment consisting of two stage, where the time before a transition to the other stage is exponential with unit rate. If we were to set instead

```
env{2,2} = Erlang.fitMeanAndOrder(1,2);
```

this would cause a race condition between two distributions in stage two: the exponential transition back to stage 1, and the Erlang-2 distributed transition with unit rate that remains in stage 2. The latter means that periodically the system will be re-initialized in stage 2, meaning that jobs in execution at a server are required all to restart execution.

In LINE, an environment is internally described by a Markov renewal process (MRP) with transition times belonging to the `PhaseType` class. A MRP is similar to a Markov chain, but state transitions are not restricted to be exponential. Although the time spent in each state of the MRP is not exponential, the MRP can be easily transformed into an equivalent continuous-time Markov chain (CTMC) to enable analysis, a task that LINE performs automatically. In the example above, the underpinning CTMC will therefore consider the distribution of the minimum between the exponential and the Erlang-2 distribution, in order to decide the next stage transition.

State space explosion may occur in the definition of an environment if the user specifies a large number of non-exponential transition. For example, a race condition among n Erlang-2 distribution translates at the level of the CTMC into a state space with 2^n states. In such situations, it is recommended to replace some of the distributions with exponential ones.

5.1.2 Specifying the system models

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 `SolverFluid`.

However, we note that the model definition can be somewhat simplified if the user describes the system model in a separate MATLAB function, accepting the stage-specific parameters in input to the function. This enables reuse of the system topology across stages, while creating independent model objects. An example of this specification style is given in `example_randomEnvironment_1.m` under LINE's example folder.

5.2 Solvers

The steady-state analysis of a system in a random environment is carried out in LINE using the blending method [**?**], 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 `SolverEnv` class, which is described next.

5.2.1 ENV

The `SolverEnv` class 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 `getAvg` method on the `SolverEnv` object, e.g.,

```
models = {model1, model2, model3, model4};
envSolver = SolverEnv(models, env, @SolverFluid,options);
[QN,UN,TN] = envSolver.getAvg()
```

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.

Appendix A

Examples

In this appendix, we summarize the features exercised in the LINE scripts under the `examples` folder.

Table A.1: Examples

Example	Problem
example_cdfRespT_1	Station response time distribution in a single-class single-job closed network
example_cdfRespT_2	Station response time distribution in a multi-chain closed network
example_cdfRespT_3	Station response time distribution in a multi-chain open network
example_cdfRespT_4	Simulation-based station response time distribution analysis
example_cdfRespT_5	Station response time distribution under increasing job populations
example_closedModel_1	Solving a single-class exponential closed queueing network
example_closedModel_2	Solving a closed queueing network with a multi-class FCFS station
example_closedModel_3	Solving exactly a multi-chain product-form closed queueing network
example_closedModel_4	Local state space generation for a station in a closed network
example_closedModel_5	1-line exact MVA solution of a cyclic network of PS and INF stations
example_closedModel_6	Closed network with round robin scheduling
example_initState_1	Specifying an initial state and prior in a single class model.
example_initState_2	Specifying an initial state and prior in a multiclass model.
example_layeredModel_1	Analyze a layered network specified in a LQNS XML file
example_layeredModel_2	Specifying and solving a basic layered network
example_misc_1	Use of performance indexes handles
example_misc_2	Update and refresh of service times
example_misc_3	Parameterization of a discriminatory processor sharing (DPS) station
example_misc_4	Automatic detection of solvers that cannot analyze the model
example_mixedModel_1	Solving a queueing network model with both closed and open classes
example_openModel_1	Solving a queueing network model with open classes
example_openModel_2	1-line solution of a tandem network of PS and INF stations
example_randomEnvironment_1	Solving a model in a 2-stage random environment
example_randomEnvironment_2	Solving a model in a 4-stage random environment
example_stateProbabilities_1	Computing marginal state probabilities
example_stateProbabilities_2	Computing marginal state probabilities