Package ‘R6arqas’

July 10, 2025

Type Package

Title Application in R for Queueing Analysis and Simulation (R6 Refactor)

Version 0.0.1

Description R6 refactor of the original arqas package. Provides analytical metrics for classic queueing models (M/M/1, M/M/s, etc.).

Depends R (>= 3.6)

Imports R6, distr, doParallel, foreach, stats, fitdistrplus, ggplot2, grid, graphics

Suggests roxygen2, devtools

License MIT + file LICENSE

Encoding UTF-8

LazyData false

Roxygen list(markdown = TRUE, r6 = FALSE)

RoxygenNote 7.3.2

Collate 'AuxSimulateModels.R'
'MarkovianModel.R'
'ComplexModels.R'
'DistributionAnalysis.R'
'GraphicsAux.R'
'SimpleModels.R'
'NetworModels.R'
'SimulateModels.R'
'zzz.R'

R topics documented:

`ClosedJackson` *ClosedJackson-class: Closed Jackson network (R6).*

Description

R6 class for a *closed* Jackson network with a fixed population of `n` circulating customers. Each node behaves as an `M/M/s` queue.

Arguments

`mu`	Numeric vector of service rates.
`s`	Integer vector with the number of servers per node.
`p`	Routing matrix. Rows must sum to 1.
`n`	Integer. Total number of customers in the network.

`ClosedNet` *Simulate a closed Jackson-type network*

Description

Wrapper around `ClosedNetSim$new()` plus optional parallel replicas.

Usage

```
ClosedNet(
  serviceDistribution,
  s,
  p,
  nClients = 3L,
  staClients = 100L,
  transitions = 1000L,
  historic = FALSE,
  nsim = 10L,
  nproc = 1L
)
```

Arguments

`serviceDistribution`	List of `distr` objects (one per node).
`s`	Integer vector with the number of servers at each node.
`p`	Routing matrix (`length(s)` × `length(s)`), row-stochastic.
`nClients`	Total number of circulating customers `N`.
`staClients`	Warm-up completions discarded from statistics.
`transitions`	Number of completed services counted for statistics.
`historic`	Logical, store the full trajectory?
`nsim`	Integer, number of independent replications.
`nproc`	Integer, CPU cores (1 = sequential).

Value

A `ClosedNetSim` object, or the aggregated result of `combineSimulations()` when `nsim > 1`.

ClosedNetSim	*ClosedNetSim – simulated closed Jackson-type network (R6)*

Description

Pure simulation (event-driven) of a *closed* queueing network with a fixed population of `nClients` customers. Each node is a `G/G/s` queue. After completing service a customer is routed to the next node according to the user-supplied routing matrix `p` (rows must sum to 1). The algorithm is a line-by-line port of the original S3 function, wrapped now in an R6 class.

Arguments

`serviceDistribution`	List of `distr` objects (one per node).
`s`	Integer vector with the number of servers at each node.
`p`	Routing matrix (`length(s)` × `length(s)`), row-stochastic.
`nClients`	Total number of circulating customers `N`.
`staClients`	Warm-up completions discarded from statistics.
`transitions`	Number of completed services counted for statistics.
`historic`	Logical, store the full trajectory?

CLOSED_JACKSON	*Functional constructor: Closed Jackson network.*

Description

Functional constructor: Closed Jackson network.

Usage

```
CLOSED_JACKSON(mu, s, p, n)
```

Arguments

`mu`	Numeric vector of service rates.
`s`	Integer vector with the number of servers per node.
`p`	Routing matrix. Rows must sum to 1.
`n`	Integer. Total number of customers in the network.

Value

A `ClosedJackson` object.

`combineSimulations` *Agrega un listado de simulaciones independientes*

Description

Combina varias réplicas de un modelo de colas calculando promedio, desviación típica y un resumen rápido para cada métrica de interés.

Usage

```
combineSimulations(listsims)
```

Arguments

`listsims`	`list` con objetos de clase (S3 o R6) simulada

Value

Un único objeto de la misma clase que `listsims[[1]]`, pero con los campos `$out` reemplazados por listas `mean/sd/summary`

`fitData` *Ajusta varias distribuciones a un vector numérico*

Description

Ajusta varias distribuciones a un vector numérico

Usage

```
fitData(
  data,
  ldistr = c("exp", "norm", "weibull", "unif", "lnorm", "gamma", "beta")
)
```

Arguments

`data`	Vector numérico con las observaciones
`ldistr`	`character` con nombres abreviados de distribuciones (ej. `"exp"`, `"norm"`, `"weibull"`, `...`)

Value

Lista de objetos `fitdist`. Clase extra: `"FitList"`

See Also

Other DistributionAnalysis: `goodnessFit()`, `summaryFit()`

FW — *CDF of the time in system $F\_W(x)$*

Description

CDF of the time in system $F_W(x)$

Usage

```
FW(qm, x)
```

Arguments

qm	An object that inherits from `MarkovianModel`.
x	Non-negative numeric vector.

FWq — *CDF of the waiting time in queue $F\_W\_q(x)$*

Description

CDF of the waiting time in queue $F_{W_q}(x)$

Usage

```
FWq(qm, x)
```

Arguments

qm	An object that inherits from `MarkovianModel`.
x	Non-negative numeric vector.

GG1 — *Simulate a G/G/1 queue*

Description

Convenience wrapper around `GG1Sim$new()` plus parallel replication.

Usage

```
GG1(
  arrivalDistribution = Exp(3),
  serviceDistribution = Exp(6),
  staClients = 100L,
  nClients = 1000L,
  historic = FALSE,
  nsim = 10L,
  nproc = 1L
)
```

Arguments

`arrivalDistribution`
: arrival distribution (object from package *distr*).

`serviceDistribution`
: service-time distribution (object from *distr*).

`staClients` integer, number of customers discarded as burn-in (stabilisation stage).

`nClients` integer, number of customers collected for statistics.

`historic` logical, record evolution of the statistics.

`nsim` integer, number of independent replications.

`nproc` integer, CPU cores to use. If `nproc = 1` the function runs sequentially.

Value

If `nsim == 1` a single `GG1Sim` object; otherwise an object of the same class containing aggregated statistics (`mean`, `sd`, etc.) produced by `combineSimulations()`.

`GG1K` *Simulate a G/G/1/K queue (wrapper)*

Description

Simulate a G/G/1/K queue (wrapper)

Usage

```
GG1K(
  arrivalDistribution = Exp(3),
  serviceDistribution = Exp(6),
  K = 2L,
  staClients = 100L,
  nClients = 1000L,
  historic = FALSE,
  nsim = 10L,
  nproc = 1L
)
```

Arguments

`arrivalDistribution, serviceDistribution`
: objects from package **distr** defining inter-arrival and service-time laws.

`K` integer, maximum queue size (>= 1). The system thus holds at most `K + 1` customers (1 in service, *K* waiting).

`staClients` warm-up customers to discard.

`nClients` accepted customers on which statistics are based.

`historic` logical, store full trajectory?

`nsim` integer, number of replications.

`nproc` integer, CPU cores (1 = sequential).

Value

GG1KSim object or aggregated result when nsim > 1.

GG1KSim *GG1KSim – simulated G/G/1/K queue (R6)*

Description

Discrete-event simulation of a single-server queue with finite capacity K (system size = K + 1, including the job in service). Arrivals finding the system full are **lost**. This is a direct R6 refactor of the legacy function G_G_1_K().

Arguments

- `arrivalDistribution, serviceDistribution`: objects from package **distr** defining inter-arrival and service-time laws.
- `K`: integer, maximum queue size (>= 1). The system thus holds at most K + 1 customers (1 in service, *K* waiting).
- `staClients`: warm-up customers to discard.
- `nClients`: accepted customers on which statistics are based.
- `historic`: logical, store full trajectory?

GG1Sim *GG1Sim – simulated G/G/1 queue (R6)*

Description

R6 class that simulates a single-server queue with a general (i.i.d.) arrival distribution and a general service distribution – commonly referred to as a *G/G/1* system. The algorithm is exactly the same as the legacy S3 implementation but the results are stored inside the object under field out.

Arguments

- `arrivalDistribution`: arrival distribution (object from package *distr*).
- `serviceDistribution`: service-time distribution (object from *distr*).
- `staClients`: integer, number of customers discarded as burn-in (stabilisation stage).
- `nClients`: integer, number of customers collected for statistics.
- `historic`: logical, record evolution of the statistics.

Stored metrics (slot `out`)

- `pn` – empirical steady-state probability vector `P{N = n}`.
- `l` – mean number of customers in the system *L*.
- `lq` – mean number of customers in the queue *Lq*.
- `w` – mean waiting time in the system *W*.
- `wq` – mean waiting time in the queue *Wq*.
- `eff` – empirical efficiency `W / (W - Wq)`.
- `rho` – empirical traffic intensity `L - Lq`.
- `historic` (optional) – matrix with the evolution of the variables during the run when `historic = TRUE`.

`GG1_Inf_HSIMH` *Simulate a G/G/1/Inf/H queue*

Description

Simulate a G/G/1/Inf/H queue

Usage

```
GG1_Inf_HSIMH(
  arrivalDistribution = Exp(3),
  serviceDistribution = Exp(6),
  H = 5L,
  staClients = 100L,
  nClients = 1000L,
  historic = FALSE,
  nsim = 10L,
  nproc = 1L
)
```

Arguments

`arrivalDistribution, serviceDistribution`	Objects from package **distr** with the inter-arrival and service-time laws.
`H`	Integer >= 1, size of the customer population.
`staClients`	Warm-up customers discarded from statistics.
`nClients`	Customers collected for statistics.
`historic`	Logical, store the whole trajectory?
`nsim`	Integer, number of independent replications.
`nproc`	Integer, CPU cores (1 = sequential).

Value

A `GG1_Inf_HSIMHSim` object, or the aggregated result of `combineSimulations()` when `nsim > 1`.

`GG1_Inf_HSIMHSim` *GG1_Inf_HSIMHSim – simulated G/G/1/Inf/H queue (R6)*

Description

Discrete-event simulation of a single-server queue fed by a *finite* population of `H` sources. At any moment each source is either **in** the system (being served or waiting) or **outside** and generating its own inter-arrival time. The total population is constant and no arrivals are lost.

Arguments

`arrivalDistribution, serviceDistribution`	Objects from package **distr** with the inter-arrival and service-time laws.
`H`	Integer >= 1, size of the customer population.
`staClients`	Warm-up customers discarded from statistics.
`nClients`	Customers collected for statistics.
`historic`	Logical, store the whole trajectory?

`GGInf` *Simulate a G/G/Inf queue*

Description

Wrapper around `GGInfSim$new()` plus optional parallel replications.

Usage

```
GGInf(
  arrivalDistribution = Exp(3),
  serviceDistribution = Exp(6),
  staClients = 100L,
  nClients = 1000L,
  historic = FALSE,
  nsim = 10L,
  nproc = 1L
)
```

Arguments

`arrivalDistribution, serviceDistribution`	Distributions from package **distr** describing inter-arrival and service times.
`staClients`	Warm-up customers discarded from statistics.
`nClients`	Customers included in statistics.
`historic`	Logical, store the whole trajectory
`nsim`	Integer, number of independent replications.
`nproc`	Integer, CPU cores (1 = sequential).

Value

A `GGInfSim` object, or the aggregated result of `combineSimulations()` when `nsim > 1`.

GGInfSim *GGInfSim – simulated G/G/Inf queue (R6)*

Description

Discrete-event simulation of a queueing system with unlimited parallel servers: every arrival starts service immediately, therefore there is **no queue** and the waiting-time in queue is always 0. The class keeps exactly the same interface used by the other simulation models in this package.

Arguments

`arrivalDistribution, serviceDistribution`
: Distributions from package **distr** describing inter-arrival and service times.

`staClients`
: Warm-up customers discarded from statistics.

`nClients`
: Customers included in statistics.

`historic`
: Logical, store the whole trajectory

GGS *Simulate a G/G/s queue (wrapper)*

Description

Simulate a G/G/s queue (wrapper)

Usage

```
GGS(
  arrivalDistribution = Exp(3),
  serviceDistribution = Exp(6),
  servers = 2L,
  staClients = 100L,
  nClients = 1000L,
  historic = FALSE,
  nsim = 10L,
  nproc = 1L
)
```

Arguments

`arrivalDistribution, serviceDistribution`
: *distr* objects.

`servers`
: integer, number of servers *s* (>= 1).

`staClients`
: integer, warm-up customers.

`nClients`
: integer, customers collected for stats.

`historic`
: logical, record full trajectory?

`nsim`
: integer, number of replications.

`nproc`
: integer, CPU cores (1 = sequential).

Value

GGSSim object or aggregated result when nsim > 1.

GGSK *Simulate a G/G/s/K queue*

Description

Simulate a G/G/s/K queue

Usage

```
GGSK(
  arrivalDistribution = Exp(3),
  serviceDistribution = Exp(6),
  servers = 2L,
  K = 3L,
  staClients = 100L,
  nClients = 1000L,
  historic = FALSE,
  nsim = 10L,
  nproc = 1L
)
```

Arguments

arrivalDistribution, serviceDistribution	objects from package **distr** defining the inter-arrival and service-time laws.
servers	Integer >= 1 (number of parallel identical servers).
K	Integer >= 1, maximum queue size (capacity minus servers).
staClients	Warm-up customers to discard.
nClients	Customers accepted for statistics.
historic	Logical, keep full trajectory?
nsim	integer, number of independent replications.
nproc	integer, CPU workers (1 = sequential).

Value

A GGSKSim object or the aggregated result of combineSimulations() when nsim > 1.

GGSKSim *GGSKSim – simulated G/G/s/K queue (R6)*

Description

Discrete-event simulation of a *multi-server* queue with finite capacity **K** (maximum queue length). The system can hold at most `s + K` customers (up to s in service, at most K in the waiting line). Arrivals that find the system full are **lost**.

Arguments

`arrivalDistribution, serviceDistribution`
: objects from package **distr** defining the inter-arrival and service-time laws.

`servers`
: Integer >= 1 (number of parallel identical servers).

`K`
: Integer >= 1, maximum queue size (capacity minus servers).

`staClients`
: Warm-up customers to discard.

`nClients`
: Customers accepted for statistics.

`historic`
: Logical, keep full trajectory?

GGSSim *GGSSim – simulated multiserver G/G/s queue (R6)*

Description

R6 class that simulates a queue with s identical servers, general i.i.d. inter–arrival and service–time distributions (G/G/s). It is a direct R6 port of the original S3 function `G_G_S()`. Results are stored in field `out`.

Arguments

`arrivalDistribution, serviceDistribution`
: *distr* objects.

`servers`
: integer, number of servers s (>= 1).

`staClients`
: integer, warm-up customers.

`nClients`
: integer, customers collected for stats.

`historic`
: logical, record full trajectory?

Stored metrics (slot `out`)

- `pn`, `l`, `lq`, `w`, `wq`, `rho`, `eff` – as in `GG1Sim`.
- `historic` matrix is present when `historic = TRUE`.

GGS_Inf_H — *Simulate a G/G/s/Inf/H queue*

Description

Wrapper around `GGS_Inf_HSim$new()` plus optional parallel replication.

Usage

```
GGS_Inf_H(
  arrivalDistribution = Exp(3),
  serviceDistribution = Exp(6),
  servers = 3L,
  H = 5L,
  staClients = 100L,
  nClients = 1000L,
  historic = FALSE,
  nsim = 10L,
  nproc = 1L
)
```

Arguments

`arrivalDistribution, serviceDistribution`	Objects from **distr** giving the inter-arrival and service-time laws.
`servers`	Integer >= 1, number of servers (*s*).
`H`	Integer >= 1, customer population size.
`staClients`	Warm-up customers discarded from statistics.
`nClients`	Customers collected for statistics.
`historic`	Logical, store the whole trajectory?
`nsim`	Integer, number of independent replications.
`nproc`	Integer, CPU cores (1 = sequential).

Value

A `GGS_Inf_HSim` object, or the aggregated result of `combineSimulations()` when `nsim > 1`.

GGS_Inf_HSim — *GGS_Inf_HSim – simulated G/G/s/Inf/H queue (R6)*

Description

Discrete-event simulation of a multi-server queue (`s` identical servers) fed by a *finite* population of `H` sources. Each source alternates between **inside** the system (being served or waiting) and **outside** where it generates its own inter-arrival time. The total population is constant and no arrivals are lost.

Arguments

`arrivalDistribution, serviceDistribution`	Objects from **distr** giving the inter-arrival and service-time laws.
`servers`	Integer >= 1, number of servers (*s*).
`H`	Integer >= 1, customer population size.
`staClients`	Warm-up customers discarded from statistics.
`nClients`	Customers collected for statistics.
`historic`	Logical, store the whole trajectory?

`GGS_Inf_HY` *Simulate a G/G/s/Inf/H/Y queue*

Description

Wrapper around `GGS_Inf_HYSim$new()` plus optional parallel replication.

Usage

```
GGS_Inf_HY(
  arrivalDistribution = Exp(3),
  serviceDistribution = Exp(6),
  servers = 3L,
  H = 5L,
  Y = 3L,
  staClients = 100L,
  nClients = 1000L,
  historic = FALSE,
  nsim = 10L,
  nproc = 1L
)
```

Arguments

`arrivalDistribution, serviceDistribution`	Objects from **distr** defining the inter-arrival and service-time laws.
`servers`	Integer >= 1, number of servers (*s*).
`H`	Integer >= 1, finite customer population.
`Y`	Integer >= 1, replacement threshold.
`staClients`	Warm-up customers discarded from statistics.
`nClients`	Customers collected for statistics.
`historic`	Logical, store the whole trajectory?
`nsim`	Integer, number of independent replications.
`nproc`	Integer, CPU cores (1 = sequential).

Value

A `GGS_Inf_HYSim` object, or the aggregated result of `combineSimulations()` when `nsim > 1`.

`GGS_Inf_HYSim` *GGS_Inf_HYSim – simulated G/G/s/Inf/H queue with Y replacements (R6)*

Description

Multi-server queue (`s` identical servers) fed by a *finite* population of `H` sources **plus** a replacement rule: when the number of customers *inside* the system is <= `Y` the source that just left is immediately replaced by a new one (fresh inter-arrival time is generated). For `Y` >= `H` the behaviour degenerates to the plain `G/G/s/Inf/H` case.

Arguments

`arrivalDistribution, serviceDistribution`	Objects from **distr** defining the inter-arrival and service-time laws.
`servers`	Integer >= 1, number of servers (s).
`H`	Integer >= 1, finite customer population.
`Y`	Integer >= 1, replacement threshold.
`staClients`	Warm-up customers discarded from statistics.
`nClients`	Customers collected for statistics.
`historic`	Logical, store the whole trajectory?

`goodnessFit` *Pruebas chi-cuadrado y KS para cada ajuste*

Description

Pruebas chi-cuadrado y KS para cada ajuste

Usage

```
goodnessFit(lfitdata)
```

Arguments

`lfitdata`	Lista producida por `fitData`

Value

`data.frame` con: distribución, estadísticos y *p-values*

See Also

Other DistributionAnalysis: `fitData()`, `summaryFit()`

MarkovianModel — `MarkovianModel` *(abstract R6 class)*

Description

Abstract R6 base class for queueing systems whose inter-arrival *and* service times follow exponential (Markovian) distributions.

Arguments

`arrivalDistribution`
: Object created with `distr::Exp(rate)`.

`serviceDistribution`
: Object created with `distr::Exp(rate)`.

`n, x, ...`
: Arguments passed on to the corresponding methods (see details above).

Format

An `R6ClassGenerator` object.

Public fields

`arrivalDistribution` An object of class `distr::Exp` that represents inter-arrival times.

`serviceDistribution` An object of class `distr::Exp` that represents service times.

`servers` `integer`. Number of parallel servers.

`out` A `list` of performance metrics (ρ, L, W, etc.) produced by subclasses.

Public methods

`$initialize()` Constructor.

`$lambda()` / `$mu()` Return arrival and service rates.

`$Pn(n)` Steady-state probability $\Pr\{N = n\}$.

`$FW(x)` CDF of the time *in the system*, $F_W(x)$.

`$FWq(x)` CDF of the *waiting* time in queue, $F_{W_q}(x)$.

`$Qn(n)` Probability that the queue length equals n.

`$maxCustomers()` Practical upper bound for the number of customers the model can hold (may be ∞).

`$print()` Pretty printer for the console.

See Also

MM1, Pn, FW

Examples

```
m <- MM1$new(4, 6)
m$Pn(0:3)
FWq(m, 1)
```

`maxCustomers`	*Upper bound on the number of customers supported*

Description

Upper bound on the number of customers supported

Usage

```
maxCustomers(qm)
```

Arguments

`qm`	An object that inherits from `MarkovianModel`.

`MM1`	*MM1-class: Class MM1*

Description

R6 class for the single-server exponential queue (M/M/1).

Arguments

`lambda`	Arrival rate (> 0).
`mu`	Service rate (> 0).

Format

An \codeR6Class generator.

Public methods

`$initialize()` Constructor.

`$lambda()` / `$mu()` Return arrival and service rates.

`$Pn(n)` Steady-state probability $\Pr\{N = n\}$.

`$FW(x)` CDF of the time *in the system*, $F_W(x)$.

`$FWq(x)` CDF of the *waiting* time in queue, $F_{W_q}(x)$.

`$Qn(n)` Probability that the queue length equals n.

`$maxCustomers()` Practical upper bound for the number of customers the model can hold (may be ∞).

`$print()` Pretty printer for the console.

Examples

```
mm1 <- MM1$new(5, 8)
mm1$out$lq
Pn(mm1, 0:2)
FW(mm1, 1)
```

`MM1InfH` *MM1InfH-class: M/M/1/Inf/H finite-population queue*

Description

R6 class for a single-server Markovian queue with a finite population of size \eqnH. Kendall notation: \emphM/M/1/Inf/H.

Arguments

`lambda`	Mean arrival rate (> 0).
`mu`	Mean service rate (> 0).
`h`	Population size (integer >= 1).

Format

An \codeR6Class generator.

Public methods

`$initialize()` Constructor.

`$lambda()` / `$mu()` Return arrival and service rates.

`$Pn(n)` Steady-state probability $\Pr\{N = n\}$.

`$FW(x)` CDF of the time *in the system*, $F_W(x)$.

`$FWq(x)` CDF of the *waiting* time in queue, $F_{W_q}(x)$.

`$Qn(n)` Probability that the queue length equals n.

`$maxCustomers()` Practical upper bound for the number of customers the model can hold (may be ∞).

`$print()` Pretty printer for the console.

Examples

```
mm <- MM1InfH$new(0.5, 12, 5)
mm$out$l; Pn(mm, 0:3); FWq(mm, 1)
```

`MM1K` *MM1K-class: Class MM1K*

Description

Single-server queue with finite capacity K (M/M/1/K).

Arguments

`lambda`	Arrival rate (> 0).
`mu`	Service rate (> 0).
`k`	Buffer size K (integer >= 0).

Public methods

`$initialize()` Constructor.

`$lambda()` / `$mu()` Return arrival and service rates.

`$Pn(n)` Steady-state probability $\Pr\{N = n\}$.

`$FW(x)` CDF of the time *in the system*, $F_W(x)$.

`$FWq(x)` CDF of the *waiting* time in queue, $F_{W_q}(x)$.

`$Qn(n)` Probability that the queue length equals n.

`$maxCustomers()` Practical upper bound for the number of customers the model can hold (may be ∞).

`$print()` Pretty printer for the console.

`MMInf` *MMInf-class: M/M/Inf infinite-server queue*

Description

MMInf-class: M/M/Inf infinite-server queue

Arguments

`lambda`	Mean arrival rate (> 0).
`mu`	Mean service rate (> 0).

Public methods

`$initialize()` Constructor.

`$lambda()` / `$mu()` Return arrival and service rates.

`$Pn(n)` Steady-state probability $\Pr\{N = n\}$.

`$FW(x)` CDF of the time *in the system*, $F_W(x)$.

`$FWq(x)` CDF of the *waiting* time in queue, $F_{W_q}(x)$.

`$Qn(n)` Probability that the queue length equals n.

`$maxCustomers()` Practical upper bound for the number of customers the model can hold (may be ∞).

`$print()` Pretty printer for the console.

`MMS` *MMS-class: Class MMS*

Description

Multi-server exponential queue (M/M/s).

Arguments

`lambda`	Arrival rate (> 0).
`mu`	Service rate (> 0).
`s`	Number of servers (integer >= 1).

Public methods

`$initialize()` Constructor.

`$lambda()` / `$mu()` Return arrival and service rates.

`$Pn(n)` Steady-state probability $\Pr\{N = n\}$.

`$FW(x)` CDF of the time *in the system*, $F_W(x)$.

`$FWq(x)` CDF of the *waiting* time in queue, $F_{W_q}(x)$.

`$Qn(n)` Probability that the queue length equals n.

`$maxCustomers()` Practical upper bound for the number of customers the model can hold (may be ∞).

`$print()` Pretty printer for the console.

`MMSInfH` *MMSInfH-class: M/M/s/Inf/H finite-population queue*

Description

MMSInfH-class: M/M/s/Inf/H finite-population queue

Arguments

`lambda`	Mean arrival rate (> 0).
`mu`	Mean service rate (> 0).
`s`	Servers (integer >= 1).
`h`	Population size (>= s).

Public methods

`$initialize()` Constructor.

`$lambda()` / `$mu()` Return arrival and service rates.

`$Pn(n)` Steady-state probability $\Pr\{N = n\}$.

`$FW(x)` CDF of the time *in the system*, $F_W(x)$.

`$FWq(x)` CDF of the *waiting* time in queue, $F_{W_q}(x)$.

`$Qn(n)` Probability that the queue length equals n.

`$maxCustomers()` Practical upper bound for the number of customers the model can hold (may be ∞).

`$print()` Pretty printer for the console.

`MMSInfHY` *MMSInfHY-class: M/M/s/Inf/H with Y replacements*

Description

MMSInfHY-class: M/M/s/Inf/H with Y replacements

Arguments

`lambda`	Mean arrival rate (> 0).
`mu`	Mean service rate (> 0).
`s`	Servers (>= 1).
`h`	Population size (> 0).
`y`	Number of replacements (>= 1).

Public methods

`$initialize()` Constructor.

`$lambda()` / `$mu()` Return arrival and service rates.

`$Pn(n)` Steady-state probability $\Pr\{N = n\}$.

`$FW(x)` CDF of the time *in the system*, $F_W(x)$.

`$FWq(x)` CDF of the *waiting* time in queue, $F_{W_q}(x)$.

`$Qn(n)` Probability that the queue length equals n.

`$maxCustomers()` Practical upper bound for the number of customers the model can hold (may be ∞).

`$print()` Pretty printer for the console.

`MMSK` *MMSK-class: Class MMSK*

Description

Multi-server queue with capacity K (M/M/s/K).

Arguments

`lambda`	Arrival rate (> 0).
`mu`	Service rate (> 0).
`s`	Servers (integer >= 1).
`k`	Capacity K (integer >= 0).

Public methods

`$initialize()` Constructor.

`$lambda()` / `$mu()` Return arrival and service rates.

`$Pn(n)` Steady-state probability $\Pr\{N = n\}$.

`$FW(x)` CDF of the time *in the system*, $F_W(x)$.

`$FWq(x)` CDF of the *waiting* time in queue, $F_{W_q}(x)$.

`$Qn(n)` Probability that the queue length equals n.

`$maxCustomers()` Practical upper bound for the number of customers the model can hold (may be ∞).

`$print()` Pretty printer for the console.

`M_M_1` *Functional constructor for MM1*

Description

Functional constructor for MM1

Usage

```
M_M_1(lambda = 3, mu = 6)
```

Arguments

`lambda`	Arrival rate (> 0).
`mu`	Service rate (> 0).

M_M_1_INF_H *Functional constructor for MM1InfH*

Description

Functional constructor for MM1InfH

Usage

```
M_M_1_INF_H(lambda = 0.5, mu = 12, h = 5L)
```

Arguments

`lambda`	Mean arrival rate (> 0).
`mu`	Mean service rate (> 0).
`h`	Population size (integer >= 1).

M_M_1_K *Functional constructor for MM1K*

Description

Functional constructor for MM1K

Usage

```
M_M_1_K(lambda = 3, mu = 6, k = 2L)
```

Arguments

`lambda`	Arrival rate (> 0).
`mu`	Service rate (> 0).
`k`	Buffer size K (integer >= 0).

M_M_INF *Functional constructor for MMInf*

Description

Functional constructor for MMInf

Usage

```
M_M_INF(lambda = 3, mu = 6)
```

Arguments

`lambda`	Mean arrival rate (> 0).
`mu`	Mean service rate (> 0).

`M_M_S` *Functional constructor for MMS*

Description

Functional constructor for MMS

Usage

```
M_M_S(lambda = 3, mu = 6, s = 2L)
```

Arguments

`lambda`	Arrival rate (> 0).
`mu`	Service rate (> 0).
`s`	Number of servers (integer >= 1).

`M_M_S_INF_H` *Functional constructor for MMSInfH*

Description

Functional constructor for MMSInfH

Usage

```
M_M_S_INF_H(lambda = 0.5, mu = 12, s = 2L, h = 5L)
```

Arguments

`lambda`	Mean arrival rate (> 0).
`mu`	Mean service rate (> 0).
`s`	Servers (integer >= 1).
`h`	Population size (>= s).

M_M_S_INF_H_Y *Functional constructor for MMSInfHY*

Description

Functional constructor for MMSInfHY

Usage

```
M_M_S_INF_H_Y(lambda = 3, mu = 6, s = 3L, h = 5L, y = 3L)
```

Arguments

lambda	Mean arrival rate (> 0).
mu	Mean service rate (> 0).
s	Servers (>= 1).
h	Population size (> 0).
y	Number of replacements (>= 1).

OpenJackson *OpenJackson-class: Open Jackson network (R6).*

Description

R6 class that represents an *open* Jackson network with independent external Poisson arrivals. Each node behaves as an `M/M/s` queue (implemented internally with `MMS`).

Arguments

lambda	Numeric vector of external arrival rates.
mu	Numeric vector of service rates.
s	Integer vector with the number of servers per node.
p	Routing matrix (square, rows sum <= 1).

Fields (read-only)

- `lambda_vec` – external arrival rates (numeric vector).
- `mu_vec` – service rates at nodes.
- `servers_vec`– servers per node.
- `routing` – routing matrix `P`.

Key methods

- `$Pn(n)` – joint steady-state probability `Pr{N = n}`.
- `$node(i)`– returns the `MMS` model of node *i*.
- `$print()`– console summary (overrides default).

OpenNet *Simulate an open Jackson-type network*

Description

Wrapper around `OpenNetSim$new()` plus optional parallel replication.

Usage

```
OpenNet(
  arrivalDistribution,
  serviceDistribution,
  s,
  p,
  staClients = 100L,
  transitions = 1000L,
  historic = FALSE,
  nsim = 10L,
  nproc = 1L
)
```

Arguments

`arrivalDistribution`	List of `distr` objects (or `no_distr()`) giving the external arrival law for every node.
`serviceDistribution`	List of `distr` objects with service-time laws.
`s`	Integer vector, servers per node.
`p`	Routing matrix; rows must sum to ≤ 1. The extra probability \(1 - $\sum_j p_{ij}$\) is interpreted as leaving the network from node *i*.
`staClients`	Warm-up completions discarded from statistics.
`transitions`	Number of completed services counted for statistics.
`historic`	Logical, collect whole trajectory?
`nsim`	Integer, number of independent replications.
`nproc`	Integer, CPU cores (1 = sequential).

Value

A `OpenNetSim` object, or the aggregated result of `combineSimulations()` when `nsim > 1`.

`OpenNetSim` *OpenNetSim – simulated open Jackson-type network (R6)*

Description

Discrete-event simulation of an *open* queueing network with external arrivals at one or more nodes and probabilistic routing between nodes. Each node behaves as a `G/G/s` queue. The internal logic is delegated to the legacy helper `OpenNetwork_secuential()` so the numerical results remain identical to the original S3 code, but all outputs are exposed through the field `out` of the R6 object.

Arguments

`arrivalDistribution`	List of `distr` objects (or `no_distr()`) giving the external arrival law for every node.
`serviceDistribution`	List of `distr` objects with service-time laws.
`s`	Integer vector, servers per node.
`p`	Routing matrix; rows must sum to ≤ 1. The extra probability \(1 - $\sum_j p_{ij}$\) is interpreted as leaving the network from node *i*.
`staClients`	Warm-up completions discarded from statistics.
`transitions`	Number of completed services counted for statistics.
`historic`	Logical, collect whole trajectory?

`OPEN_JACKSON` *Functional constructor: Open Jackson network.*

Description

Functional constructor: Open Jackson network.

Usage

```
OPEN_JACKSON(lambda, mu, s, p)
```

Arguments

`lambda`	Numeric vector of external arrival rates.
`mu`	Numeric vector of service rates.
`s`	Integer vector with the number of servers per node.
`p`	Routing matrix (square, rows sum <= 1).

Value

An `OpenJackson` object.

`ParallelizeSimulations` *Ejecuta en paralelo* `nsim` *simulaciones de un modelo*

Description

Ejecuta en paralelo `nsim` simulaciones de un modelo

Usage

```
ParallelizeSimulations(modelfunction, parameters, nsim = 1L, nproc = 1L)
```

Arguments

`modelfunction`	Función que genera **una** réplica (debe retornar un objeto con campo `$out`)
`parameters`	`list` con los argumentos que recibe `modelfunction`
`nsim`	Número de réplicas a lanzar
`nproc`	Núcleos a utilizar (>= 1). Si vale 1 $\rightarrow$ secuencial

Value

Una lista con las réplicas, o la única réplica si `nsim == 1`

`plot_history` *Quick historic plot for any simulated queue or network*

Description

Quick historic plot for any simulated queue or network

Usage

```
plot_history(x, var = "L", ...)
```

Arguments

`x`	A single simulation object (`GG1Sim`, `OpenNetSim`, ...) **or** a list of such objects.
`var`	One of `"L"`, `"Lq"`, `"W"`, `"Wq"`, `"Clients"`, `"Intensity"`.
`...`	Extra parameters forwarded to the internal helpers (e.g., `minrange`, `maxrange`, `depth`, `showMean`, `showValues`).

Value

A **ggplot2** object.

`Pn` *Steady-state probability* $\Pr\{N = n\}$

Description

S3 frontend that delegates to `qm$Pn(n)`.

Usage

```
Pn(qm, n)
```

Arguments

`qm`	An object that inherits from `MarkovianModel`.
`n`	Non-negative integer vector.

`Qn` *Probability that the queue length equals* n

Description

Probability that the queue length equals n

Usage

```
Qn(qm, n)
```

Arguments

`qm`	An object that inherits from `MarkovianModel`.
`n`	Non-negative integer vector.

`summaryFit` *Resumen gráfico (densidad, CDF y Q-Q plot)*

Description

Resumen gráfico (densidad, CDF y Q-Q plot)

Usage

```
summaryFit(
  lfitdata,
  graphics = c("ggplot2", "graphics"),
  show = c("all", "dens", "cdf", "qq")
)
```

Arguments

`lfitdata`	Salida de `fitData`
`graphics`	`"graphics"` o `"ggplot2"`
`show`	`"all"`, `"dens"`, `"cdf"` o `"qq"`

See Also

Other DistributionAnalysis: `fitData()`, `goodnessFit()`

Index