Computationally Efficient Simulation of Queues: The R Package queuecomputer

Anthony Ebert
Queensland University of Technology
ACEMS

Paul Wu
Queensland University of Technology
ACEMS

Kerrie Mengersen
Queensland University of Technology
ACEMS

Fabrizio Ruggeri
CNR-IMATI
Queensland University of Technology
ACEMS

Abstract

Large networks of queueing systems model important real-world systems such as MapReduce clusters, web-servers, hospitals, call-centers and airport passenger terminals. To model such systems accurately we must infer queueing parameters from data. Unfortunately, for many queueing networks there is no clear way to proceed with parameter inference from data. Approximate Bayesian computation could offer a straight-forward way to infer parameters for such networks if we could simulate data quickly enough.

We present a computationally efficient method for simulating from a very general set of queueing networks with the R package **queuecomputer**. Remarkable speedups of more than 2 orders of magnitude are observed relative to the popular DES packages **simmer** and **simpy**. We replicate output from these packages to validate the package.

The package is modular and integrates well with the popular R package **dplyr**. Complex queueing networks with tandem, parallel and fork/join topologies can easily be built with these two packages together. We show how to use this package with two examples: a call-centre and an airport terminal.

Keywords: queues, queueing theory, discrete event simulation, operations research, approximate Bayesian computation, R.

1. Introduction

The queues we encounter in our everyday experience, where customers wait in line to be served by a server, is a useful analogy for many other processes. We say analogy because the word *customers* could represent: MapReduce jobs (Lin, Zhang, Wierman, and Tan 2013); patients in a hospital (Takagi, Kanai, and Misue 2016); items in a manufacturing system (Dallery and Gershwin 1992); calls to a call center (Gans, Koole, and Mandelbaum 2003); shipping containers in a seaport (Kozan 1997) or even cognitive tasks (Cao 2013). Similarly *server* could represent: a compute cluster; medical staff; machinery or a customer service representative at a call centre. Queueing systems can also be networked together to form *queueing*

networks. We can use queueing networks to build models of processes such as provision of internet services (Sutton and Jordan 2011), passenger facilitation at international airports (Wu and Mengersen 2013) and emergency evacuations (Van Woensel and Vandaele 2007). Clearly queueing systems and queueing networks are useful for understanding important real-world systems.

Performance measures for a given queueing system can often only be derived through simulation. Queues are usually simulated with discrete event simulation (DES) (Rios Insua, Ruggeri, and Wiper 2012, pg. 226). In DES changes in state are discontinuous. The state is acted upon by a countable list of *events* at certain times which cause the discontinuities. If the occurrence of an event is independent of everything except simulation time it is *determined*, otherwise it is *contingent* (Nance 1981).

Popular DES software packages are available in many programming languages including: the R package **simmer** (Ucar and Smeets 2016), the Python (Van Rossum and Drake 2014) package **simpy** (Lünsdorf and Scherfke 2013) and the Java (Gosling 2000) package **JMT** (Bertoli, Casale, and Serazzi 2009). DES packages are often so expressive that they can be considered languages in their own right, indeed the programming language Simula (Dahl and Nygaard 1966) is a literal example of this.

queuecomputer (Ebert 2016) implements an algorithm that can easily be applied to a wide range of queueing systems and networks of queueing systems. It is vastly more computationally efficient than existing approaches to DES. We term this new computationally efficient algorithm queue departure computation (QDC). Computational efficiency is important because if we can simulate from queues quickly then we can embed a queue simulation within an approximate Bayesian computation (ABC) algorithm (Sunnåker, Busetto, Numminen, Corander, Foll, and Dessimoz 2013) and estimate queue parameters for very complicated queueing models in a straight forward manner.

In Section 2 we review the literature on queueing theory and develop notation used throughout this paper. In Section 3 we present the QDC algorithm and compare it to DES. We demonstrate usage of the package in Section 4. Details of implementation and usage are discussed in Section 5. The package is validated in Section 6 by replicating results from DES packages **simpy** and **simmer**. We compare computed performance measures from the output of a **queuecomputer** simulation to theoretical results for $M/M/2$ queueing systems. We benchmark the package in Section 7 and compare computation time with **simpy** and **simmer**. Examples in Section 8 are used to demonstrate how the package can be used to simulate a call centre and an international airport terminal.

2. Queueing theory

Queueing theory is the study of queueing systems and originated from the work of Agner Krarup Erlang in 1909 to plan infrastructure requirements for the Danish telephone system (Thomopoulos 2012, pg 2).

A *queueing system* is defined as follows. Each customer $i = 1, 2, \cdots$ has an arrival time a_i (or equivalently an inter-arrival time $\delta_i = a_i - a_{i-1}$, $a_0 = 0$) and an amount of time they require with a server, called the *service time* s_i. Typically a server can serve only one customer at a time. A server which is currently serving another customer is said to be *unavailable*, a server without a customer is *available*. If all servers are unavailable when a customer arrives

then customers must wait in the queue until a server is available. Detailed introductions to queueing systems can be found in standard texts such as Bhat (2015).

The characteristics of a queueing system is expressed with the notation of Kendall (1953). This notation has since been extended to six characteristics:

- f_δ, inter-arrival distribution;
- $f_{\mathbf{s}}$, service distribution;
- K, number of servers $\in \mathbb{N}$;
- C, capacity of system $\in \mathbb{N}$;
- n, customer population $\in \mathbb{N}$; and
- R, service discipline

Choices for inter-arrival and service distributions are denoted by "M" for exponential and independently distributed, "GI" for general and independently distributed and "G" for general without the independence assumption. The capacity of the system C refers to the maximum number of customers within the system at any one time[1]. Customers are within the system if they are being served or waiting in the queue. The customer population n is the total number of customers including those outside of the system (yet to arrive or already departed). The service discipline R defines how customers in the queue are allocated to available servers. The most common service discipline is first come first serve (FCFS). To specify a queueing system, these characteristics are placed in the order given above and separated by a forward slash "/".

The simplest queueing system is exponential in distribution for both the inter-arrival $\delta_i \overset{iid}{\sim} \exp(\lambda)\ \forall i \in 1:n$ and service processes $s_i \overset{iid}{\sim} \exp(\mu)\ \forall i \in 1:n$, where λ and μ are exponential rate parameters. Additionally, K is set to 1, C and n are infinite and R is FCFS. It is denoted by $M/M/1/\infty/\infty/FCFS$, which is shortened to $M/M/1$.

Parameter inference for this system was considered first by Clarke (1957), estimators were derived from the likelihood function. This likelihood is later used by Muddapur (1972) to derive the joint posterior distribution. Bayesian inference for queueing systems is summarised in detail by Rios Insua *et al.* (2012).

Managers and planners are less interested in parameter inference and more interested in performance measures such as: $N(t)$, the number of customers in system at time t; $\bar{B}$, the average number of busy servers; ρ, the *resource utilization*; and $\bar{\mathbf{w}}$, the average waiting time for customers. If $\lambda < K\mu$ the queueing system will eventually reach equilibrium and distributions of performance measures become independent of time.

In the case of a $M/M/K$ system equilibrium distributions for performance measures are derived analytically, they are found in standard queueing theory textbooks (Lipsky 2008; Thomopoulos 2012). For instance the limit probability of N customers in the system $\mathsf{P}(N)$ is

[1]If the system is at full capacity and new customers arrive, new customers leave the system immediately without being served.

$$\mathsf{P}(0) = \left[\frac{(K\rho)^K}{K!(1-\rho)} + 1 + \sum_{i=1}^{K-1} \frac{(K\rho)^i}{i!} \right]^{-1} \tag{1}$$

$$\mathsf{P}(N) = \begin{cases} \mathsf{P}(0)\frac{(K\rho)^n}{N!} & N \leq K \\ \mathsf{P}(0)\frac{(K\rho)^n}{K!K^{N-K}} & \text{otherwise} \end{cases} \tag{2}$$

where ρ, the resource utilization, is defined as $\frac{\lambda}{K\mu}$. For an $M/M/K$ system this is equal to the expected number of busy servers divided by the total number of servers $\frac{\mathsf{E}(B)}{K}$ (Cassandras and Lafortune 2009, pg. 451). The expected number of customers in the system is (Bhat 2015)

$$\mathsf{E}(N) = K\rho + \frac{\rho(K\rho)^K P(0)}{K!(1-\rho)^2}, \tag{3}$$

and the expected waiting time is

$$\mathsf{E}(\mathbf{w}) = \frac{(K\rho)^K P(0)}{K!K\mu(1-\rho)^2}. \tag{4}$$

If the parameters of f_δ and $f_{\mathbf{s}}$ are uncertain then we must turn to predictive distributions for estimates of performance measures. Predictive distributions of performance measures using Bayesian posterior distributions are derived by Armero (1994); Armero and Bayarri (1999).

Jackson (1957) was one of the first to consider networks of queueing systems. In a *Jackson network* there is a set of J queueing systems. After a customer is served by queueing system j they arrive at another queueing system with fixed probability $p_{j,k}$. Customers leave the system with probability $1 - \sum_{k=1}^{J} p_{j,k}$. Other examples of queueing networks include the *tandem* (Glynn and Whitt 1991), *parallel* (Hunt and Foote 1995) and the *fork/join* (Kim and Agrawala 1989) topologies.

In a tandem queueing network, customers traverse an ordered series of queues before departing the system. Real examples of such systems include airport terminals, internet services and manufacturing systems. In a parallel network, customers are partitioned into different $(\mathbf{a}, \mathbf{s})$ to be seen by separate queueing systems. In a fork/join network a task (another term for customer) is forked into a number of subtasks which are to be completed by distinct parallel servers. The difference from the parallel network is that the task can only depart the system once all subtasks have arrived at the join point.

Most models of queueing systems assume time-invariant inter-arrival and service processes. In practice many real-world queues have inter-arrival processes which are strongly time-dependent, such as: call-centres (Weinberg, Brown, and Stroud 2007; Brown, Gans, Mandelbaum, Sakov, Shen, Zeltyn, and Zhao 2005), airport runways (Koopman 1972) and hospitals (Brahimi and Worthington 1991). In the case of the $M/M/1$ queue we can adapt the notation to $M(t)/M(t)/1$ to represent exponential processes where parameters $\lambda(t)$ and $\mu(t)$ change with time. Such queueing systems are referred to as *dynamic* queueing systems.

In general, analytic solutions do not exist for dynamic queueing systems (Malone 1995; Worthington 2009). Green, Kolesar, and Svoronos (1991) showed that using stationary queueing systems to model dynamic queueing systems leads to serious error even if deviation from stationarity is slight. The problem is compounded once we consider queueing networks. Understanding long-term and transient behaviour of such queues can only be achieved with approximation methods or simulation. We now detail the QDC algorithm, a computationally efficient method for simulating queueing systems.

3. Queue departure computation

3.1. Fixed number of servers

QDC can be considered as a multi-server extension to an algorithm presented by Lindley (1952). For a single server queueing system, the departure time of the ith customer is: $d_i = \max(a_i, d_{i-1}) + s_i$, since the customer either waits for a server or the server waits for a customer. The algorithm (not the paper) was, surprisingly, not extended to multi-server systems until Krivulin (1994). However with each new customer i the algorithm must search a growing $i+1$ length vector. This algorithm therefore scales poorly, with computational complexity $O(n^2)$, where n is number of customers. Kin and Chan (2010) adapted the original algorithm of Kiefer and Wolfowitz (1955) to an $O(nK)$ algorithm for multiserver tandem queues with blocking, that is $G/G/K/C$ queueing systems where C is the maximum capacity number of customers in the queueing systems.

QDC can also be viewed as a computationally efficient solution to the set of equations presented in Sutton and Jordan (2011, pg. 259) for FCFS queueing systems. There is a single queue served by a fixed number of K servers. The ith customer observes a set of times $b_i = \{b_{ik} | k \in 1:K\}$ which represents the times when each server will next be available. The customer i selects the earliest available server $p_i = \text{argmin}(b_i)$ from b_i. The departure time for the ith customer is therefore $d_i = \max(a_i, b_{p_i}) + s_i$, since the server must wait for the customer or the customer must wait for the server. The QDC algorithm for a fixed number of servers (Algorithm 1) pre-sorts the arrival times. Rather than assigning a b_i for each customer i to form the matrix $\mathbf{b} \in \mathbf{M}^{n \times K}$, QDC considers $\mathbf{b}$ as a continually updated K length vector representing the state of the system.

This algorithm is simple and computationally efficient. At each iteration of the loop we need only search $\mathbf{b}$, a K length vector for the minimum element in code line 8. In the language of DES, we consider $\mathbf{b}$ as the system state and $\mathbf{a}$ as the event list, which are all *determined* events. This differs from conventional DES approaches to modelling queueing systems where the queue length is the system state and both $\mathbf{a}$ and $\mathbf{d}$ constitute the event list, where the events of $\mathbf{a}$ are *determined* and the events of $\mathbf{d}$ are continually updated and therefore *contingent*.

Algorithm 1 can simulate any queue of the form $G(t)/G(t)/K/\infty/n/FCFS$ where K and n can be made arbitrarily large. Furthermore, the inter-arrival and service distributions can be of completely general form and even have a dependency structure between them. Since the arrival and service times are supplied by the user rather than sampled in-situ, the algorithm "decouples" statistical sampling from queue computation. This frees the user to simulate queues of arbitrarily complex $f_{\delta,\mathbf{s}}$, where K is fixed.

Algorithm 1 QDC for fixed K

function QDC_NUMERIC($\mathbf{a} \in \mathbb{R}_+^n, \mathbf{s} \in \mathbb{R}_+^n, K \in \mathbb{N}$)
  Sort $(\mathbf{a}, \mathbf{s})$ in terms of $\mathbf{a}$ (ascending)
  Create vector $\mathbf{p} \in \mathbb{N}^n$.
  Create vector $\mathbf{b} \in \mathbb{R}_+^K$.
  Create vector $\mathbf{d} \in \mathbb{R}_+^n$.
  $b_k \leftarrow 0 \qquad \forall k \in 1:K$
  for $i \in 1:n$ **do**
    $p_i \leftarrow \arg\min(\mathbf{b})$
    $b_{p_i} \leftarrow \max(a_i, b_{p_i}) + s_i$
    $d_i \leftarrow b_{p_i}$
  end for
  Put $(\mathbf{a}, \mathbf{d}, \mathbf{p})$ back to original (input) ordering of $\mathbf{a}$
  return $(\mathbf{d}, \mathbf{p})$
end function

3.2. Changing number of servers

Conditional case

Suppose that the number of servers that customers can use changes throughout the day. This reflects realistic situations where more servers are rostered on for busier times of the day. We say that for a certain time t, the customers have a choice of $K(t)$ *open* servers from K. This means that there are $K(t)$ servers rostered-on for time t. We define the term *closed* as the opposite of open.

We represent the number of open servers throughout the day as a step function. Time is on the positive real number line and is partitioned by L knot locations $\mathbf{x} = (x_1, \cdots, x_L) \in \mathbb{R}_+^L$ into $L+1$ epochs $(0, x_1], (x_1, x_2], \cdots, (x_L, \infty)$. The number of open servers in each epoch is represented by a $L+1$ length vector $\mathbf{y} = (y_1, \cdots, y_{L+1}) \in \mathbb{N}_0^{L+1}$. If we assume that none of the service times $\mathbf{s}$ span the length of more than one epoch $(x_l, x_{l+1}]$, formally

$$\forall i \quad [s_i < \min(x_{l+1} - x_l | l \in 1:L)]\,, \tag{5}$$

then we need consider a change in state over at most 1 knot location. This step function is determined input by the user. Like the arrival and service times $(\mathbf{a}, \mathbf{s})$ it is changeable by the user before the simulation but not during the simulation.

We close server k by writing an ∞ symbol to b_k ensuring that no customer can use that server. If the server needs to be open again at time t we write t to b_k allowing customers to use that server. Since $\mathbf{x}$ now corresponds to changes in $\mathbf{b}$ it is part of the event list along with $\mathbf{a}$. The entire event list is still determined and need not be updated mid-simulation.

This algorithm can simulate queues of form $G(t)/G(t)/K(t)/\infty/n/FCFS$, where $K(t)$ refers to the number of open servers changing with time. As mentioned previously this algorithm is subject to Condition 5, this condition is not overly restrictive if we consider realistic systems with few changes in K. The recorded server allocations $\mathbf{p} = (p_1, \cdots, p_n)$ may not reflect the real system since Algorithm 2 does not allow the user to specify exactly which servers are

Algorithm 2 QDC for $K(t)$ (conditional)

function QDC_SERVER.STEPFUN($\mathbf{a} \in \mathbb{R}_+^n, \mathbf{s} \in \mathbb{R}_+^n, \mathbf{x} \in \mathbb{R}_+^L, \mathbf{y} \in \mathbb{N}_0^{L+1}$)
  Sort $(\mathbf{a}, \mathbf{s})$ in terms of $\mathbf{a}$ (ascending)
  $x_{L+1} \leftarrow \infty$
  $y_{L+2} \leftarrow 1$
  $K \leftarrow \max(\mathbf{y})$
  Create vector $\mathbf{b} \in \mathbb{R}_+^K$.
  $b_k \leftarrow \infty \quad \forall k \in 1 : K$
  $b_k \leftarrow 0 \quad \forall k \in 1 : y_0$
  Create vector $\mathbf{p} \in \mathbb{N}^n$.
  Create vector $\mathbf{d} \in \mathbb{R}_+^n$.
  $l \leftarrow 1$
  $p_1 \leftarrow 1$
  for $i \in 1 : n$ **do**

    // Adjustments to $\mathbf{b}$ *with change in epoch*
    if $\forall k \in 1 : K \quad [b_k \geq x_{l+1}]$ OR $a_i \geq x_{l+1}$ **then**
      if $y_{l+1} - y_l > 0$ **then**
        for $k \in (y_l + 1 : y_{l+1})$ **do**
          $b_k \leftarrow x_{l+1}$
        end for
      end if
      if $y_{l+1} - y_l < 0$ **then**
        for $k \in (y_{l+1} + 1 : y_l)$ **do**
          $b_k \leftarrow \infty$
        end for
      end if
      $l \leftarrow l + 1$
    end if
    // End of adjustments to $\mathbf{b}$ *with change in epoch*

    $p_i \leftarrow \arg\min(\mathbf{b})$
    $b_{p_i} \leftarrow \max(a_i, b_{p_i}) + s_i$
    $d_i \leftarrow b_{p_i}$

    // Extra loop if current size is zero so that customer i *can be processed in next epoch*
    if $y_l = 0$ **then**
      $i \leftarrow i - 1$
    end if

  end for
  Put $(\mathbf{a}, \mathbf{d}, \mathbf{p})$ back to original (input) ordering of $\mathbf{a}$
  return $(\mathbf{d}, \mathbf{p})$
end function

open in each epoch, only how many are open and closed. If this output is needed or in cases where Condition 5 does not hold, we must use the less computationally efficient but more general unconditional algorithm below.

Unconditional case

If Condition 5 does not hold or if otherwise we wish to control exactly which servers are open at what time then we must use a less computationally efficient algorithm (Algorithm 4). Each server k has its own partition of L_k knot locations $\mathbf{x_k} = (x_{k,1}, \cdots, x_{k,L_k}) \in \mathbb{R}_+^{L_k}$ and each

$\mathbf{y_k} = (y_{k,1}, \cdots, y_{k,L_k+1})$ is an alternating sequence of 0 and 1s of length L_k indicating whether the server is open or closed respectively for the associated epoch. The vector $\mathbf{c}$ is used slightly differently to how it is used in Sutton and Jordan (2011). We use it to represent the time at which each server is next available for the current customer i, given the current system state $\mathbf{b}$. It is the output of the `next_fun` function.

Algorithm 3 Next function

function NEXT_FUN($t, \mathbf{x} \in \mathbb{R}_+^L, \mathbf{y}$)
  Find l such that $x_l < t \leq x_{l+1}$.
  if $y_{l+1} = 0$ **then**
    return x_{l+1}
  else
    return t
  end if
end function

Algorithm 4 QDC for $K(t)$ (unconditional)

function QDC_SERVER.LIST($\mathbf{a} \in \mathbb{R}_+^n, \mathbf{s} \in \mathbb{R}_+^n, \underline{\mathbf{x}} = (\mathbf{x_1}, \cdots, \mathbf{x_K}), \underline{\mathbf{y}} = (\mathbf{y_1}, \cdots, \mathbf{y_K})$)
  Sort $(\mathbf{a}, \mathbf{s})$ in terms of $\mathbf{a}$ (ascending)
  $\forall k \in 1:K \quad x_{k,L_k+1} \leftarrow \infty$
  $\forall k \in 1:K \quad y_{k,L_k+2} \leftarrow 1$
  $K \leftarrow$ length($\underline{\mathbf{x}}$)
  Create vector $\mathbf{c} \in \mathbb{R}_+^K$
  Create vector $\mathbf{b} \in \mathbb{R}_+^K$
  $b_k \leftarrow 0 \quad \forall k \in 1:K$
  Create vector $\mathbf{p} \in \mathbb{N}^n$.
  Create vector $\mathbf{d} \in \mathbb{R}_+^n$.
  for $i \in 1:n$ **do**
    for $k \in 1:K$ **do**
      $c_k \leftarrow$ NEXT_FUN($\max(b_k, a_i), \mathbf{x_k}, \mathbf{y_k}$)
    end for
    $p_i \leftarrow \arg\min(\mathbf{b})$
    $b_{p_i} \leftarrow c_{p_i} + s_i$
    $d_i \leftarrow b_{p_i}$
  end for
  Put $(\mathbf{a}, \mathbf{d}, \mathbf{p})$ back to original (input) ordering of $\mathbf{a}$
  return $(\mathbf{d}, \mathbf{p})$
end function

This algorithm can simulate queueing systems of form $G(t)/G(t)/K(t)/\infty/n/FCFS$, where $K(t)$ refers to the number of open servers changing with time. In addition we can specify which particular servers are available when, not just how many and we are not bound by Condition 5. Once again we note that $\mathbf{b}$ can be considered as the system state and the event list is formed by $\mathbf{a}$ and the elements of $\underline{\mathbf{x}}$. This function can be called with the `queue_step` function in **queuecomputer** by supplying a `server.list` object to the `servers` argument. For the rest of this paper we focus on Algorithms 1 and 2 for their relative conceptual simplicity

and computational efficiency.

3.3. Discussion

With the algorithms so far presented we can simulate from a very general set of queueing systems $G(t)/G(t)/K(t)/\infty/n/FCFS$ in a computationally efficient manner. In contrast to the algorithm of Kin and Chan (2010), the state vector $\mathbf{b}$ is written over in each iteration. The memory usage for QDC therefore scales with $O(n)$ rather than $O(nK)$.

Tandem queueing networks can be simulated by using the output of one queueing system as the input to the next queueing system. We demonstrate this idea with the Airport Simulation examples in Section 8.2. Fork/join queueing networks are addressed in the next section where we explain the implementation details of **queuecomputer** with regards to the QDC algorithm.

4. Usage

The purpose of the package **queuecomputer** is to compute, deterministically, the output of a queueing system given the arrival and service times for all customers. The most important function is `queue_step`. The first argument to `queue_step` is a vector of arrival times, the second argument is a vector of service times and the third argument specifies the servers available.

```
R> library("queuecomputer")
R> arrivals <- cumsum(rexp(100))
R> head(arrivals)
```

```
[1] 0.693512 1.693399 2.425550 3.952405 3.961906 4.405492
```

```
R> service <- rexp(100)
R> departures <- queue_step(arrivals, service = service, servers = 2)
R> departures
```

```
# A tibble: 100 × 6
   arrivals     service departures      waiting system_time server
      <dbl>       <dbl>      <dbl>        <dbl>       <dbl>  <dbl>
1  0.693512 0.830158956   1.523671 0.000000e+00 0.830158956      1
2  1.693399 0.817648174   2.511047 1.110223e-16 0.817648174      2
3  2.425550 0.002675641   2.428226 2.138047e-16 0.002675641      1
4  3.952405 0.667180991   4.619586 4.440892e-16 0.667180991      1
5  3.961906 0.551920432   4.513827 4.440892e-16 0.551920432      2
6  4.405492 1.069236762   5.583063 1.083341e-01 1.177570886      2
7  4.594253 1.110448926   5.730035 2.533279e-02 1.135781711      1
8  4.993053 0.766944956   6.350008 5.900099e-01 1.356954853      2
9  6.047412 0.805061421   6.852474 1.110223e-16 0.805061421      1
10 6.856338 1.317802131   8.174140 0.000000e+00 1.317802131      2
# ... with 90 more rows
```

The output of a `queue_step` function is a `queue_list` object. We built a summary method for objects of class `queue_step`, which we now demonstrate.

```
R> summary(departures)

Total customers:
 100
Missed customers:
 0
Mean waiting time:
 0.246
Mean response time:
 1.11
Utilization factor:
 0.53
Mean queue length:
 0.301
Mean number of customers in system:
 1.36
```

If the last element of **y** is zero it is possible that some customers will never be served, this is the "Missed customers" output. The performance measures that follow are the mean waiting time $\bar{w}$, the mean response time $\bar{r} = d - a$, the observed utilization factor $\bar{B}/K$, the mean queue length and the mean number of customers in the system respectively. The utilization factor $\bar{B}/K$ takes into account the changing number of open servers $K(t)$ where Algorithm 2 is used. We now explain the implementation details of package.

5. Implementation

The `for` loops within Algorithms 1 and 2 are written in C++ with the **Armadillo** library (Sanderson and Curtin 2016). The C++ **for** loops are called using the R packages **Rcpp** (Eddelbuettel, François, Allaire, Chambers, Bates, and Ushey 2011) and **RcppArmadillo** (Eddelbuettel and Sanderson 2014). We use R to provide wrapper functions for the C++ code.

The `queue_step` calls the more primitive `queue` function which is a wrapper for S3 methods which implement Algorithms 1, 2 or 4 depending on the class of the object supplied to the `server` argument of `queue_step`. If `class(server)` is numeric then `queue` runs Algorithm 1, if it is a `server.stepfun` then `queue` runs Algorithm 2, if it is a `server.list` then `queue` runs Algorithm 4. The `queue` function computes departure times **d** and server allocations **p** and the `queue_step` function adds additional output such as waiting times and queue lengths which are used in summary and plot methods.

To simulate fork/join networks the **queuecomputer** function `wait_step` provides a simple wrapper to the **base** function `pmax.int`, this function computes the maximum of each row for a set of two equal length numeric vectors. The vectors represent the departure times for each subjob and the departure time for the entire job is the maximum of each subjob.

In **simmer** and **simpy** users supply generator functions for simulating δ and service times $\mathbf{s}$, the user enters the set of input parameters θ_I for these generator functions and starts the simulation. The inter-arrival time is resampled after each arrival and the service time is sampled when the server begins with a new customer. This makes it difficult to model queues where distributions for inter-arrival times do not make sense: like the immigration counter for an airport, where multiple flights generate customers; or when arrival times and service times are not independent. In **queuecomputer** sampling is "decoupled" from computation, the user samples $\mathbf{a}$ and service times $\mathbf{s}$ using any method. The outputs $\mathbf{d}$ and $\mathbf{p}$ are then computed deterministically.

We now demonstrate the validity of **queuecomputer**'s output by replicating results from the DES packages **simmer** and **simpy**. We then replicate equilibrium analytic results of performance measures for the $M/M/2$ queue.

6. Validation

6.1. Comparison with simmer and simpy

To demonstrate the validity of the algorithm we consider a $M/M/2/\infty/1000/FCFS$ queue. If QDC is valid for any $M/M/K$ queueing system then it is valid for any $G(t)/G(t)/K$ queueing system. This is because any non-zero $(\mathbf{a}, \mathbf{s})$ could conceivably come from two exponential distributions, even if the probability of the particular realization is vanishingly small. We replicate exact departure times computed with the **simmer** and **simpy** packages using **queuecomputer**. First we generate $\mathbf{a}$ and $\mathbf{s}$ to be used as input to all three packages.

```
R> set.seed(1)
R> n_customers <- 10^4
R> lambda_a <- 1/1
R> lambda_s <- 1/0.9
R> interarrivals <- rexp(n_customers, lambda_a)
R> arrivals <- cumsum(interarrivals)
R> service <- rexp(n_customers, lambda_s)
```

We now input these objects into the three scripts using **queuecomputer**, **simmer**, or **simpy**. First we run the **queuecomputer** script. The `queuecomputer_output` object is sorted in ascending order so that the departure times can be compared to the DES packages.

```
R> queuecomputer_output <- queue_step(arrivals = arrivals,
+    service = service, servers = 2)
R> head(sort(depart(queuecomputer_output)))

[1] 1.340151 2.288112 2.639976 2.796572 3.249794 5.714967
```

The DES packages **simmer** and **simpy** are not built to allow users to input $(\mathbf{a}, \mathbf{s})$ directly. Rather, the user supplies parameters for f_δ and $f_{\mathbf{s}}$ so that inter-arrival and service times can be sampled at each step when needed. To allow **simmer** and **simpy** to accept presampled input

$(\mathbf{a}, \mathbf{s})$ we use generator functions in place of the usual `rexp(rate)` or `random.expovariate(rate)` calls in R and Python respectively, details of this work can be found in the supplementary material. We create an interface to **simmer** so that it can be called in the same way as **queuecomputer**.

```
R> simmer_output <- simmer_step(arrivals = arrivals,
+    service = service, servers = 2)
R> head(simmer_output)

[1] 1.340151 2.288112 2.639976 2.796572 3.249794 5.714967
```

The same departure times are observed. Similarly in Python we create an interface to **simpy** so that it can be called in a similar way to **queuecomputer**.

```
python> simpy_step(interarrivals, service)[0:6]

array([ 1.34015149,  2.28811237,  2.63997568,  2.79657232,  3.24979406,
        5.7149671 ])
```

A check of all three sorted vectors of $\mathbf{d}$ from each package revealed that all were equal to within 5 significant figures for every $d_i, i = 1 : 1000$.

6.2. Replicate theoretical results for M/M/3

We use a $M/M/3/\infty/5 \times 10^6/FCFS$ simulation in **queuecomputer** to replicate theoretical equilibrium results for key performance indicators for a $M/M/2/\infty/\infty/FCFS$ queueing system. We set λ to 1 and set μ to 2.

Theoretical results

We first note that the traffic intensity is ρ of $2/3 = 0.\dot{6}$, which should correspond to the average number of busy servers. The probability of N customers in the system is given by Equation 2. We perform this computation up to $N = 20$ and display the results in Figure 1. The expected waiting time is $\mathsf{E}(\mathbf{w})$ is $0.\dot{4}$ and the expected number of customers in the system $\mathsf{E}(N)$ is $2.\dot{8}$.

Simulation results

The inputs $\mathbf{a}$ and $\mathbf{s}$ must first be generated.

```
R> set.seed(1)
R> n_customers <- 5e6
R> lambda <- 2
R> mu <- 1
R> interarrivals <- rexp(n_customers, lambda)
R> arrivals <- cumsum(interarrivals)
R> service <- rexp(n_customers, mu)
R> K = 3
```

We now use the `queue_step` function and the summary method for `queue_list` objects `summary.queue_list` to return observed key performance measures.

```
R> MM3 <- queue_step(arrivals = arrivals, service = service, servers = K)
R> summary(MM3)

Total customers:
 5000000
Missed customers:
 0
Mean waiting time:
 0.445
Mean response time:
 1.44
Utilization factor:
 0.666
Mean queue length:
 0.889
Mean number of customers in system:
 2.89
```

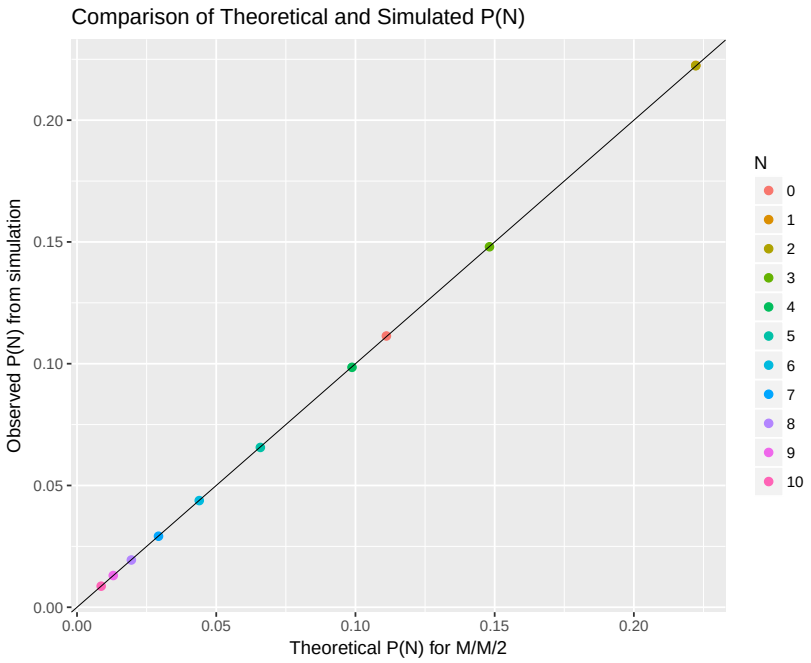

Figure 1: Comparison of theoretical equilibrium $\mathsf{P}(N)$ and observed proportions from simulation. Observation $N = 1$ is obscured by $N = 2$.

We see that the observed time average number of busy servers is 0.6661402 which is close to $0.\dot{6}$ the value for ρ. We can see that the observed mean waiting time is close to the expected mean waiting time. The expected number of customers in the system, from the distribution

$\mathsf{P}(N)$ is close to the observed number of customers in the system. The entire distribution of $\mathsf{P}(N)$ is replicated in Figure 1.

7. Benchmark

7.1. Method

The compare the computational efficiency of each package we compute the departure times from a $M/M/2/\infty/n/FCFS$ queueing system, with $\lambda = 1$ and $\mu = 1.\dot{1}$. To understand how n affects computation time we repeat the experiment 100 times for n $= 10^2, 10^3, 10^5$ and 10^6. We could only repeat the $n = 10^6$ experiment 10 times for **simmer** because of memory issues, we also repeat the experiment at $n = 10^7$ for **queuecomputer**. We compare the median time taken for each combination of package and n.

The simulation was conducted on a system with Intel (R) Core(TM) i7-6700 CPU @ 3.40GHz running Debian GNU/Linux. The version of R is 3.3.3 "Another Canoe" with **simmer** version 3.6.1 and **queuecomputer** version 0.8.1. The version of Python is 3.4.2 with **simpy** module version 3.6.1.

To assess the computation time for **queuecomputer** and **simmer** we use the `microbenchmark` function from the **microbenchmark** package (Mersmann 2015) with `time = 100` and compute the median. Full details can be found in the supplementary material.

7.2. Results and discussion

The median computation time for each package and for varying numbers of customers from 100 to 10^6 customers (up to 10^7 customers for **queuecomputer**) is shown in Figure 2. We observe phenomenal speedups for **queuecomputer** compared to both packages: compared to **simpy** speedups of 50 (at 100 customers) to 600 (at 10^6 customers) are observed, and for **simmer** speedups of 170 (at 100 customers) to 3100 (at 10^6 customers) are observed. The speedup is lower for smaller n since **queuecomputer** approaches a minimum computation time.

Simulating 10 million customers takes less than 1 second for **queuecomputer**. We see no reason why queues of different arrival and service distributions should not have similar speedups. This is because, as mentioned earlier, any non-negative $(\mathbf{a}, \mathbf{s})$ could come from two exponential distributions.

Clearly QDC and its implementation **queuecomputer** are a more computationally efficient way to simulate queueing systems of the form $G(t)/G(t)/K/\infty/M/FCFS$ than conventional DES algorithms implemented by **simpy** and **simmer**.

8. Examples

8.1. Call centre

We demonstrate **queuecomputer** by simulating a call centre. The arrival time for each customer is the time that they called, the service time is how long it takes for their problem to be resolved once they reach an available customer service representative. Let's assume that

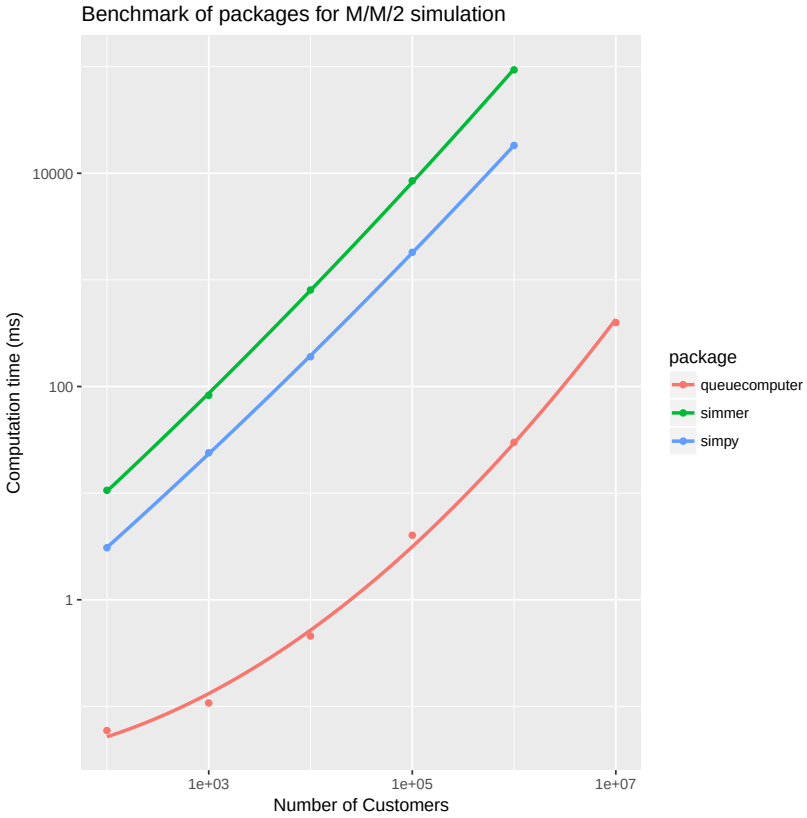

Figure 2: Computation time in milliseconds for varying numbers of passengers for each DES/queueing package. Each package returns exactly the same set of departure times, since the same arrival and service times are supplied. The computation time reported here is the median time of 100 runs for each number of customers and each package. Intel (R) Core(TM) i7-6700 CPU @ 3.40GHz running Debian GNU/Linux.

the customers arrive by a homogeneous Poisson process over the course of the day.

```
R> library("queuecomputer")
R> library("randomNames")
R> library("ggplot2")
R> set.seed(1)
R> interarrivals <- rexp(20, 1)
R> arrivals <- cumsum(interarrivals)
R> customers <- randomNames(20, name.order = "first.last")
```

We also need a vector of service times for every customer.

```
R> service <- rexp(20, 0.5)
R> head(service)
```

```
[1] 2.4245155 1.1304372 0.3815756 0.5626325 2.0107932 1.4076141
```

We put the arrival and service times into the `queue_step` function to compute the departure times. Here we have set the number of customer service representatives to two. The "servers" argument is used for this input.

```
R> queue_obj <- queue_step(arrivals, service, servers = 2, labels = customers)
R> head(queue_obj$departures_df)
```

```
# A tibble: 6 × 7
   labels  arrivals   service departures      waiting system_time server
    <chr>     <dbl>     <dbl>      <dbl>        <dbl>       <dbl>  <dbl>
1 Kwabena 0.7551818 2.6669670   3.422149 0.000000e+00    2.666967      1
2 Beatriz 1.9368246 1.2434810   3.180306 0.000000e+00    1.243481      2
3 Dashawn 2.0825313 0.4197332   3.600039 1.097774e+00    1.517507      2
4  Karina 2.2223266 0.6188957   4.041045 1.199822e+00    1.818718      1
5    Ilea 2.6583952 2.2118725   5.811911 9.416435e-01    3.153516      2
6 Brianna 5.5533638 1.5483755   7.101739 2.220446e-16    1.548376      1
```

We can see that Kwabena arrives first but leaves after Beatriz. This is possible because there are two servers. Kwabena's service took so long that the next two customers were served by the other server. It's easy to see how the departure times were computed in this simple example. Kwabena and Beatriz were the first customers for each server so we can compute their departure time by just adding their service times to their arrival times.

```
R> firstcustomers <- arrivals[1:2] + service[1:2]
R> firstcustomers
```

```
[1] 3.422149 3.180306
```

Dashawn however had to wait for an available server, since he arrived after the first two customers arrived but before the first two customers departed. He must wait until one of these customers departs before he can be served. We add the departure time of the first customer (Beatriz) to his service time to compute his departure time.

```
R> firstcustomers[2] + service[3]
```

```
[1] 3.600039
```

So the first two customers had no waiting time but Dashawn had to wait for an available server. We can compute the waiting times for all three customers in this manner:

```
R> depart(queue_obj)[1:3] - arrivals[1:3] - service[1:3]
```

```
[1] 0.000000 0.000000 1.097774
```

The `depart` function is a convenience function for retrieving the departure times from a `queue_list` object. The `queue_step` function returns a `queue_list` object. There is a summary method for this object within the **queuecomputer** package, this can be accessed by calling `summary(departures)`.

```
R> summary(queue_obj)

Total customers:
 20
Missed customers:
 0
Mean waiting time:
 1.15
Mean response time:
 3.69
Utilization factor:
 0.834
Mean queue length:
 0.858
Mean number of customers in system:
 2.42
```

The plot method in **queuecomputer** for `queue_list` objects uses the plotting package **ggplot2** (Wickham 2009) to return a list of plots. We produce four plots: a histogram of the arrival and departure times (Figure 3); a plot of the queue length and number of customers in the system over time (Figure 4); a plot of the waiting and service times for each customer (Figure 5); and a plot of the empirical cumulative distribution function for arrival and departure times (Figure 6). These plots correspond to selections 2, 5 and 6 in the `which` argument, a similar API to the `plot.lm` method in the **stats** package (R Core Team 2016).

```
R> plot(queue_obj, which = c(2, 4, 5, 6)
```

Notice that in Figure 5, if we draw a horizontal line anywhere on the left-hand plot it will never pass through more than two red bars. This must be the case otherwise a server would be serving more than one customer at a time.

8.2. International airport terminal

The package integrates naturally with the popular data manipulation R package **dplyr** (Wickham and Francois 2016). We demonstrate how to integrate **queuecomputer** and **dplyr** with a more complex Airport Terminal than before (Figure 7). Passengers from a set of 120 flights disembark at the arrivals concourse, and proceed through immigration using either the "smart gate" or the "manual gate" route, we therefore have two queues in parallel. The route taken (smart gate or manual gate) by each passenger is predetermined, but the server used by the passenger within these separate queueing systems is not.

Their bags are unloaded from the flights and proceed to the baggage hall with a delay, the division of a passengers and bags is a fork/join network. The bags and passengers are forked at the arrival concourse and joined at the baggage hall. After immigration the passengers proceed on to the baggage hall where they pick up their bags.

We have a synthetic dataset of passengers `ID` from 120 flights `FlightNo`, with an average of 103.8 passengers per flight for a total of 20,758 passengers. The dataset includes (for each

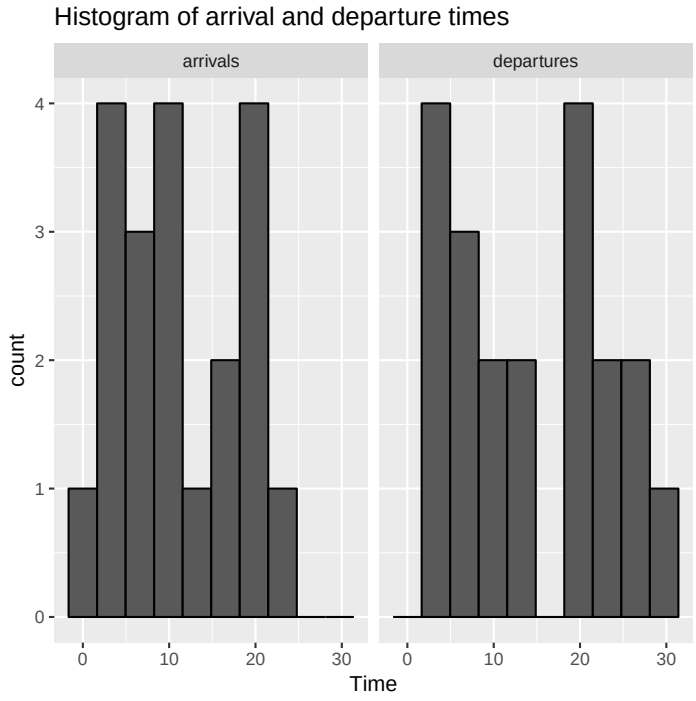

Figure 3: Histogram of arrival and departure times for all customers.

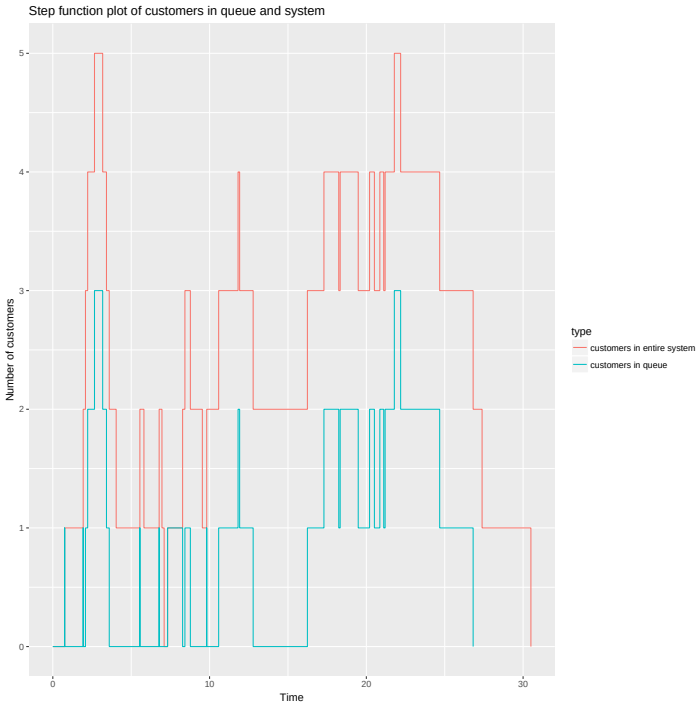

Figure 4: Plot of queue length and number of customers in system over time.

passenger `ID`): their flight number `FlightNo`, the arrival time of that flight `arrival`, the route

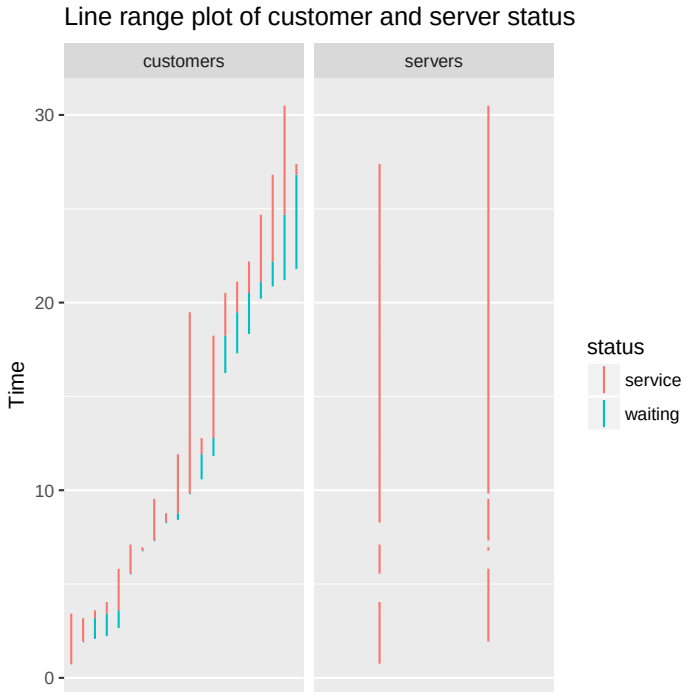

Figure 5: Waiting and service times for each customer.

take (smart/manual gate) by that passenger `route_imm`, the arrival times to immigration after they walk through the terminal `arrival_imm` and the service time needed by the passenger at their immigration queueing system `service_imm`.

```
R> Passenger_df
```

```
## # A tibble: 25,012 × 7
##                   ID FlightNo arrival  route_imm arrive_imm service_imm
##                <chr>   <fctr>   <dbl>     <fctr>      <dbl>       <dbl>
## 1     Cordell, Megan   ABI481  564.85     manual   566.8549  0.29075606
## 2    Matheson, Dylan   ABI481  564.85     manual   566.8532  0.15927226
## 3      Avitia, Renee   ABI481  564.85     manual   567.2014  0.22450319
## 4       Woods, Tyrel   ABI481  564.85 smart gate   566.8377  0.18222445
## 5   Pope, Christiana   ABI481  564.85 smart gate   566.0994  0.09031344
## 6   Espinoza, Mariah   ABI481  564.85 smart gate   566.8928  0.43900281
## 7  Pacheco, Charleen   ABI481  564.85     manual   567.5558  0.12917143
## 8    Harmon, Brendan   ABI481  564.85 smart gate   566.3114  0.30565961
## 9   William, Gerardo   ABI481  564.85 smart gate   567.2563  0.31975687
## 10       Hood, Colen   ABI481  564.85 smart gate   567.2181  0.33944458
## # ... with 25,002 more rows, and 1 more variables: bag_time <dbl>
```

Immigration processing is split into two routes with the `route_imm` variable. The `"smart gate"` route has 5 servers, whereas the `"manual"` route has 10 servers before time 600, 12 servers between time 600 and time 780 and 8 servers from time 780 onwards. We store this information in a new `dataframe` called `server_df`.

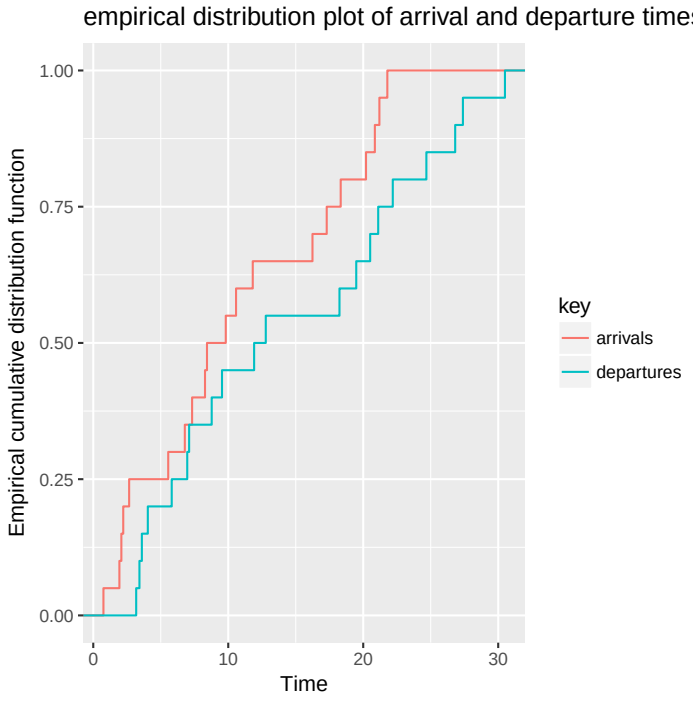

Figure 6: Empirical cumulative distribution functions for arrival and departure times. For each time the different of the functions is equal to the number of customers currently in the system (in queue and currently being served).

```
R> server_df <- data.frame(immigration_route = c("smart gate", "manual"))
R> server_df$servers <-
+    list(5, as.server.stepfun(x = c(600,780), y = c(10,12,8)))
```

To compute the departure times from the parallel servers we use the **dplyr** function `group_by`. The dataset is then processed as if it has been split in two.

```
R> Passenger_df <- left_join(Passenger_df, server_df)

## Joining, by = "route_imm"

R> Passenger_df <- Passenger_df %>%
+    group_by(route_imm) %>%
+    mutate(
+      departures_imm =
+        queue(arrive_imm, service_imm, servers = servers[[1]])
+      ) %>%
+    ungroup() %>%
+    mutate(departures_bc = pmax.int(departures_imm, bag_time))
R> Passenger_df %>%
+    select(FlightNo, arrive_imm, departures_imm, departures_bc)
```

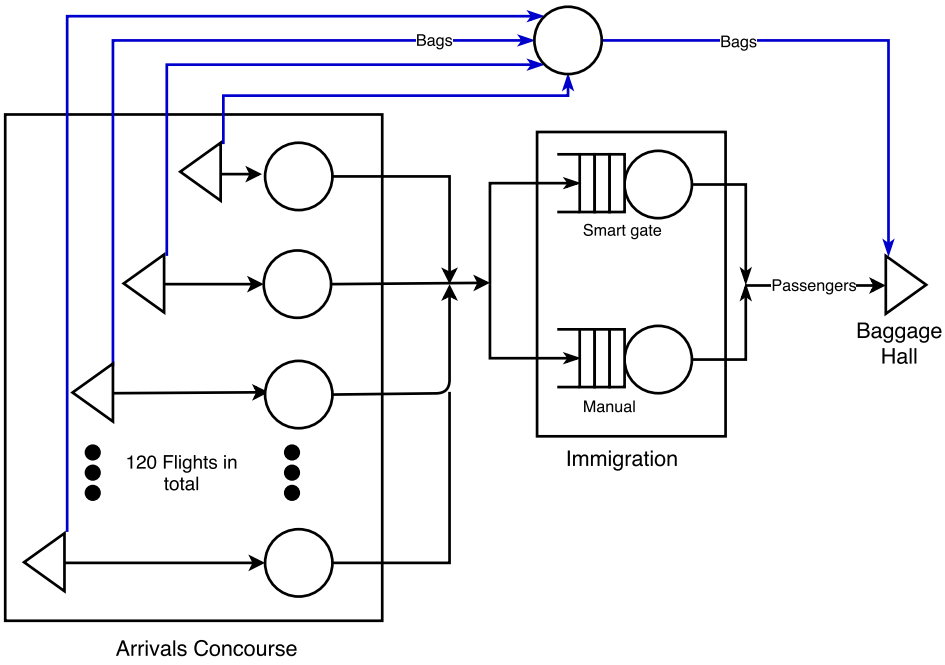

Figure 7: Diagram of larger airport scenario, there are 120 flights in total and two multi-server queueing systems operate in parallel. Passengers are preassigned to travel through either the "manual" or "smart gate" route through immigration. The passengers and bags are "forked" when each aircraft arrives are are "joined" at the baggage hall.

```
## # A tibble: 25,012 × 4
##    FlightNo arrive_imm departures_imm departures_bc
##      <fctr>      <dbl>          <dbl>         <dbl>
## 1    ABI481   566.8549       578.8443      578.8443
## 2    ABI481   566.8532       578.6479      578.6479
## 3    ABI481   567.2014       579.6133      579.6133
## 4    ABI481   566.8377       571.9148      574.6535
## 5    ABI481   566.0994       570.2073      574.7161
## 6    ABI481   566.8928       572.2135      574.7912
## 7    ABI481   567.5558       579.9953      579.9953
## 8    ABI481   566.3114       570.6290      574.8944
## 9    ABI481   567.2563       573.3438      574.8996
## 10   ABI481   567.2181       573.1459      574.9295
## # ... with 25,002 more rows
```

The column `departures_imm` represents the times at which passengers depart immigration after having been served either through the manual counter of smart gate. The column `departures_bc` represents the times that customers leave with their bags from the baggage hall. Waiting times can be summarised with the `summarise` function from **dplyr**, here we compute summaries of waiting times for each `FlightNo` and immigration route `route_imm` and a summary of waiting times only by `route_imm`.

```
R> Passenger_df %>%
```

```
+    group_by(FlightNo, route_imm) %>%
+    summarise(
+      waiting_imm = mean(departures_imm - service_imm - arrive_imm),
+      waiting_bc = mean(departures_bc - departures_imm)
+    )

## Source: local data frame [240 x 4]
## Groups: FlightNo [?]
##
##    FlightNo  route_imm waiting_imm waiting_bc
##      <fctr>     <fctr>       <dbl>      <dbl>
## 1    ABI481     manual  11.2718493   6.292506
## 2    ABI481 smart gate   4.9586928  12.276112
## 3    AEB843     manual   0.8497289  16.594387
## 4    AEB843 smart gate   1.0133830  16.412237
## 5    ARH364     manual  12.5063642   3.795084
## 6    ARH364 smart gate   7.3569441   8.169201
## 7    BCH445     manual   1.8033024  13.700561
## 8    BCH445 smart gate   1.4368622  15.087956
## 9    BJN726     manual  19.5150170   2.064476
## 10   BJN726 smart gate   7.2110772   9.751714
## # ... with 230 more rows

R> Passenger_df %>%
+    group_by(route_imm) %>%
+    summarise(
+      waiting_imm = mean(departures_imm - service_imm - arrive_imm),
+      waiting_bc = mean(departures_bc - departures_imm)
+    )

## # A tibble: 2 × 3
##    route_imm waiting_imm waiting_bc
##       <fctr>       <dbl>      <dbl>
## 1     manual    8.252137   9.559913
## 2 smart gate    4.488549  12.786145
```

We can quickly build a complex dynamic queueing model involving tandem, parallel and fork/join topologies. The model is efficient to compute, modular and easily extended. This was achieved by combining the **queuecomputer** and **dplyr** packages.

9. Conclusion

The R package **queuecomputer** implements QDC. It can be used to simulate any queueing systems or tandem network of queueing systems of general form $G(t)/G(t)/K(t)/\infty/n/FCFS$. Fast algorithms for multi-server queueing systems have been proposed in the past (Krivulin 1994; Sutton and Jordan 2010; Kin and Chan 2010). These algorithms have generated little

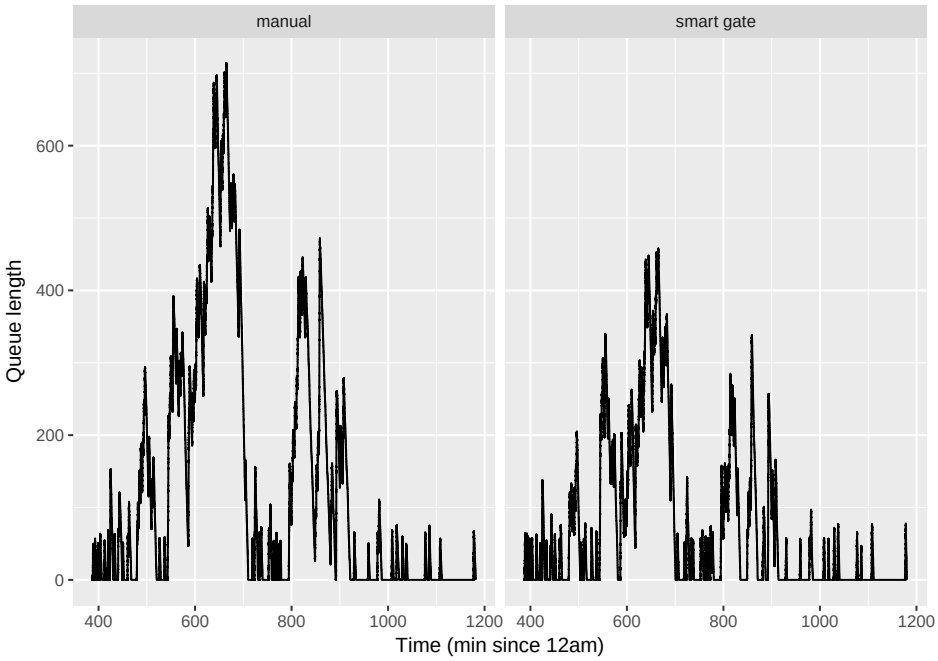

Figure 8: Queue lengths over the course of the day for "manual" and "smart gate" immigration routes.

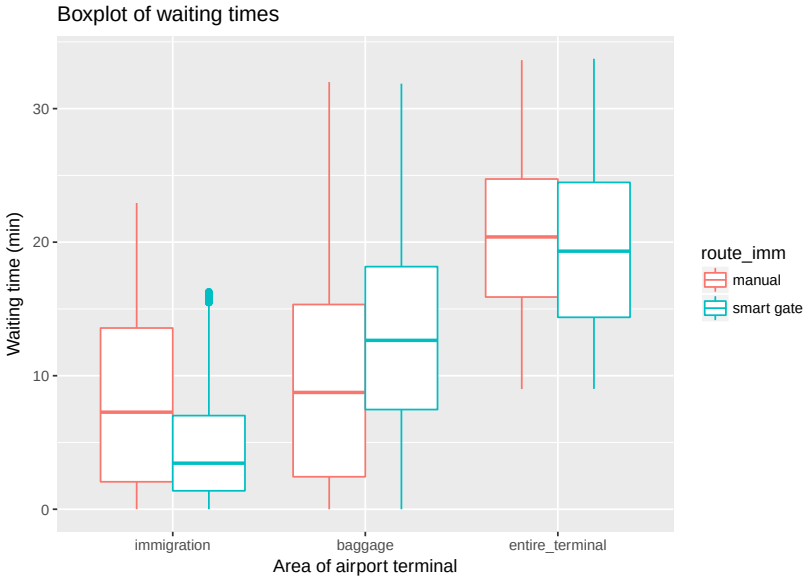

Figure 9: Boxplot of waiting times for each stage of passenger processing within the international airport terminal.

notice, even in the cases where their computational efficiency is demonstrated (Kin and Chan 2010). QDC is conceptually simpler, more efficient memory-wise and modular.

We validated QDC with analytic results and by replicating output generated by existing DES packages **simpy** and **simmer**. We observe speedups of up to 3 orders of magnitude. The speed

of the package will allow queue simulations to be embedded within ABC algorithms, which will be addressed in future work. Unlike existing DES packages, sampling and departure time computation are clearly 'decoupled' and therefore allow the user to simulate queueing systems with arrival and service time distributions of arbitrary complexity. The package integrates well with the data manipulation package **dplyr** and these two packages together allow the user to quickly and easily simulate queueing networks with parallel, tandem and fork/join topologies.

Acknowledgements

This work is supported by the Australian Research Council Centre of Excellence for Mathematical and Statistical Frontiers (ACEMS). This work was funded through the Australian Research Council (ARC) linkage grant "Improving the Productivity and Efficiency of Australian Airports" (LP140100282).

References

Armero C (1994). "Bayesian Inference in Markovian Queues." *Queueing Systems*, **15**(1-4), 419–426.

Armero C, Bayarri M (1999). "Dealing with Uncertainties in Queues and Networks of Queues: A Bayesian Approach."

Bertoli M, Casale G, Serazzi G (2009). "**JMT**: Performance Engineering Tools for System Modeling." *SIGMETRICS Perform. Eval. Rev.*, **36**(4), 10–15. ISSN 0163-5999. doi:http://doi.acm.org/10.1145/1530873.1530877.

Bhat UN (2015). *An Introduction to Queueing Theory: Modeling and Analysis in Applications*. Birkhäuser.

Brahimi M, Worthington DJ (1991). "Queueing Models for Out-Patient Appointment Systems—A Case Study." *Journal of the Operational Research Society*, **42**(9), 733–746. URL http://link.springer.com/article/10.1057/jors.1991.144.

Brown L, Gans N, Mandelbaum A, Sakov A, Shen H, Zeltyn S, Zhao L (2005). "Statistical Analysis of a Telephone Call Center: A Queueing-Science Perspective." *Journal of the American statistical association*, **100**(469), 36–50.

Cao S (2013). *Queueing Network Modeling of Human Performance in Complex Cognitive Multi-task Scenarios*. Ph.D. thesis, University of Michigan. URL https://deepblue.lib.umich.edu/bitstream/handle/2027.42/102477/shicao_1.pdf?sequence=1.

Cassandras CG, Lafortune S (2009). *Introduction to Discrete Event systems*. Springer-Verlag.

Clarke AB (1957). "Maximum Likelihood Estimates in a Simple Queue." *The Annals of Mathematical Statistics*, **28**(4), 1036–1040.

Dahl OJ, Nygaard K (1966). "SIMULA: An ALGOL-based Simulation Language." *Communications of the ACM*, **9**(9), 671–678.

Dallery Y, Gershwin SB (1992). "Manufacturing Flow Line Systems: A Review of Models and Analytical Results." *Queueing systems*, **12**(1-2), 3–94. URL http://link.springer.com/article/10.1007/BF01158636.

Ebert A (2016). ***queuecomputer**: Computationally Efficient Queue Simulation*. R package version 0.6.1, URL https://CRAN.R-project.org/package=queuecomputer.

Eddelbuettel D, François R, Allaire J, Chambers J, Bates D, Ushey K (2011). "**Rcpp**: Seamless R and C++ Integration." *Journal of Statistical Software*, **40**(8), 1–18.

Eddelbuettel D, Sanderson C (2014). "**RcppArmadillo**: Accelerating R with High-Performance C++ Linear Algebra." *Computational Statistics & Data Analysis*, **71**, 1054–1063.

Gans N, Koole G, Mandelbaum A (2003). "Telephone Call Centers: Tutorial, Review, and Research Prospects." *Manufacturing & Service Operations Management*, **5**(2), 79–141. URL http://pubsonline.informs.org/doi/abs/10.1287/msom.5.2.79.16071.

Glynn PW, Whitt W (1991). "Departures from many Queues in Series." *The Annals of Applied Probability*, pp. 546–572. URL http://www.jstor.org/stable/2959706.

Gosling J (2000). *The Java language specification*. Addison-Wesley Professional.

Green L, Kolesar P, Svoronos A (1991). "Some Effects of Nonstationarity on Multiserver Markovian Queueing Systems." *Operations Research*, **39**(3), 502–511.

Hunt CS, Foote BL (1995). "Fast Simulation of Open Queueing Systems." *Simulation*, **65**(3), 183–190. URL http://sim.sagepub.com/content/65/3/183.short.

Jackson JR (1957). "Networks of Waiting Lines." *Operations research*, **5**(4), 518–521.

Kendall DG (1953). "Stochastic Processes Occurring in the Theory of Queues and their Analysis by the Method of the Imbedded Markov Chain." *The Annals of Mathematical Statistics*, pp. 338–354.

Kiefer J, Wolfowitz J (1955). "On the Theory of Queues with many Servers." *Transactions of the American Mathematical Society*, **78**(1), 1–18. URL http://www.jstor.org/stable/1992945.

Kim C, Agrawala AK (1989). "Analysis of the Fork-Join Queue." *IEEE Transactions on computers*, **38**(2), 250–255. URL http://ieeexplore.ieee.org/abstract/document/16501/.

Kin W, Chan V (2010). "Generalized Lindley-Type Recursive Representations for Multiserver Tandem Queues with Blocking." *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, **20**(4), 21. URL http://dl.acm.org/citation.cfm?id=1842726.

Koopman BO (1972). "Air-Terminal Queues under Time-Dependent Conditions." *Operations Research*, **20**(6), 1089–1114.

Kozan E (1997). "Comparison of Analytical and Simulation Planning Models of Seaport Container Terminals." *Transportation Planning and Technology*, **20**(3), 235–248. URL http://www.tandfonline.com/doi/abs/10.1080/03081069708717591.

Krivulin NK (1994). "A Recursive Equations Based Representation for the G/G/m Queue." *Applied Mathematics Letters*, **7**(3), 73–77. URL http://www.sciencedirect.com/science/article/pii/0893965994901163.

Lin M, Zhang L, Wierman A, Tan J (2013). "Joint Optimization of Overlapping Phases in MapReduce." *Performance Evaluation*, **70**(10), 720–735. URL http://www.sciencedirect.com/science/article/pii/S0166531613000916.

Lindley DV (1952). "The Theory of Queues with a Single Server." In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 48, pp. 277–289. Cambridge Univ Press. URL http://journals.cambridge.org/article_S0305004100027638.

Lipsky L (2008). *Queueing Theory: A Linear Algebraic Approach*. Springer-Verlag.

Lünsdorf O, Scherfke S (2013). ***simpy**: Discrete Event Simulation for **Python***. python package version 3.0.10, URL https://simpy.readthedocs.io/en/latest/index.html.

Malone KM (1995). *Dynamic Queueing Systems: Behavior and Approximations for Individual Queues and for Networks*. Ph.D. thesis, Massachusetts Institute of Technology.

Mersmann O (2015). ***microbenchmark**: Accurate Timing Functions*. R package version 1.4-2.1, URL https://CRAN.R-project.org/package=microbenchmark.

Muddapur M (1972). "Bayesian Estimates of Parameters in some Queueing Models." *Annals of the Institute of Statistical Mathematics*, **24**(1), 327–331.

Nance RE (1981). "The Time and State Relationships in Simulation Modeling." *Communications of the ACM*, **24**(4), 173–179.

R Core Team (2016). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Rios Insua D, Ruggeri F, Wiper M (2012). *Bayesian Analysis of Stochastic Process Models*, volume 978. John Wiley & Sons.

Sanderson C, Curtin R (2016). "**Armadillo**: A Template-Based C++ Library for Linear Algebra." *Journal of Open Source Software*, **1**(2), 26–32.

Sunnåker M, Busetto AG, Numminen E, Corander J, Foll M, Dessimoz C (2013). "Approximate Bayesian Computation." *PLoS Comput Biol*, **9**(1), e1002803.

Sutton C, Jordan MI (2011). "Bayesian Inference for Queueing Networks and Modeling of Internet Services." *The Annals of Applied Statistics*, pp. 254–282.

Sutton CA, Jordan MI (2010). "Inference and Learning in Networks of Queues." In *AISTATS*, pp. 796–803. URL http://www.jmlr.org/proceedings/papers/v9/sutton10a/sutton10a.pdf.

Takagi H, Kanai Y, Misue K (2016). "Queueing Network Model for Obstetric Patient Flow in a Hospital." *Health care management science*, pp. 1–19. URL http://link.springer.com/article/10.1007/s10729-016-9363-5.

Thomopoulos NT (2012). *Fundamentals of Queuing Systems: Statistical Methods for Analyzing Queuing Models*. Springer-Verlag.

Ucar I, Smeets B (2016). ***simmer**: Discrete-Event Simulation for R*. R package version 3.5.1, URL https://CRAN.R-project.org/package=simmer.

Van Rossum G, Drake FL (2014). "The Python Language Reference." https://docs.python.org/release/3.4.2/reference/index.html#reference-index.

Van Woensel T, Vandaele N (2007). "Modeling Traffic Flows with Queueing Models: A Review." *Asia-Pacific Journal of Operational Research*, **24**(04), 435–461. URL http://www.worldscientific.com/doi/abs/10.1142/S0217595907001383.

Weinberg J, Brown LD, Stroud JR (2007). "Bayesian Forecasting of an Inhomogeneous Poisson Process with Applications to Call Center Data." *Journal of the American Statistical Association*, **102**(480), 1185–1198.

Wickham H (2009). *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York. ISBN 978-0-387-98140-6. URL http://ggplot2.org.

Wickham H, Francois R (2016). ***dplyr**: A Grammar of Data Manipulation*. R package version 0.5.0, URL https://CRAN.R-project.org/package=dplyr.

Worthington D (2009). "Reflections on Queue Modelling from the last 50 Years." *Journal of the Operational Research Society*, **60**(1), S83–S92.

Wu PPY, Mengersen K (2013). "A Review of Models and Model Usage Scenarios for an Airport Complex System." *Transportation Research Part A: Policy and Practice*, **47**, 124–140.

Affiliation:

Anthony Ebert
School of Mathematical Sciences
Science and Engineering Faculty
Queensland University of Technology
Brisbane Queensland 4000, Australia
E-mail: ac.ebert@qut.edu.au
URL: https://bragqut.wordpress.com/people/anthony-ebert/

Type	Notation	Definition
Queue Specification	λ	Rate parameter of exponential inter-arrival distribution $f_\delta = \text{Exp}(\lambda)$ for $M/M/K$ queue.
	μ	Rate parameter of exponential service distribution $f_{\mathbf{s}} = \text{Exp}(\mu)$ for $M/M/K$ queue.
	$\rho := \frac{\lambda}{K\mu}$	Traffic intensity, defined only for $M/M/K$ queues.
	θ_I	Parameters of arrival and service joint distribution for QDC, $(\mathbf{a}, \mathbf{s}) \sim f_{\mathbf{a},\mathbf{s}}(\cdot\|\theta_I)$.
	K	Number of servers.
	C	Capacity of system.
	n	Total number of customers.
	R	Service discipline of queue.
	FCFS	First come first serve (Type of service discipline)
	L	Number of knot locations for server changes
Input/Output	$\mathbf{a} = (a_1, \cdots, a_n)$	Arrival process, where a_i is the time at which the ith customer arrives at the queue.
	$\delta = (\delta_1, \cdots, \delta_n)$	Inter-arrival process, where δ_i is $a_i - a_{i-1} \quad \forall i \in 1 : n$ and $\delta_1 = a_1$.
	$\mathbf{s} = (s_1, \cdots, s_n)$	Service process, where s_i is the service time of the ith customer.
	$\mathbf{d} = (d_1, \cdots, d_n)$	Departure process, where d_i is the time at which the ith customer leaves the queue after being served.
	$\mathbf{p} = (p_1, \cdots, p_n)$	Server process, where p_i is the server who served the ith customer.
	$\mathbf{b} = (b_1, \cdots, b_K)$	This vector represents the time at which each server $1 : K$ will next be free. We consider this vector to be the state of the system.
	$\mathbf{x} = (x_1, \cdots, x_L)$	Change times for number of open servers.
	$\mathbf{y} = (y_1, \cdots, y_{L+1})$	Number of open servers in each epoch.
Performance Measures	$N(t)$	Number of customers in system at time t.
	$\bar{B}$	Time average number of busy servers, referred to as "Resource utilization".
	$\bar{w}$	Average waiting time per customer.

Table 1: Notation and definitions.