Journal of Simulation

ISSN: 1747-7778 (Print) 1747-7786 (Online) Journal homepage: https://orsociety.tandfonline.com/loi/tjsm20

Ciw: An open-source discrete event simulation library

Geraint I. Palmer, Vincent A. Knight, Paul R. Harper & Asyl L. Hawa

To cite this article: Geraint I. Palmer, Vincent A. Knight, Paul R. Harper & Asyl L. Hawa (2019) Ciw: An open-source discrete event simulation library, Journal of Simulation, 13:1, 68-82, DOI: 10.1080/17477778.2018.1473909

To link to this article: https://doi.org/10.1080/17477778.2018.1473909

JOURNAL OF SIMULATION
2019, VOL. 13, NO. 1, 68–82
https://doi.org/10.1080/17477778.2018.1473909

THE OPERATIONAL RESEARCH SOCIETY
Taylor & Francis
Taylor & Francis Group

ARTICLE

Ciw: An open-source discrete event simulation library

Geraint I. Palmer, Vincent A. Knight, Paul R. Harper and Asyl L. Hawa

School of Mathematics, Cardiff University, Cardiff, UK

ABSTRACT

This paper introduces Ciw, an open-source library for conducting discrete event simulations that has been developed in Python. The strengths of the library are illustrated in terms of best practice and reproducibility for computational research. An analysis of Ciw's performance and comparison to several alternative discrete event simulation frameworks is presented.

ARTICLE HISTORY
Received 12 September 2017 Revised 18 April 2018 Accepted 30 April 2018

KEYWORDS
Reproducibility; discrete event simulation; open source; python

1. Introduction

The analysis of queueing systems, especially those arranged into networks, is a standard approach to studying a variety of real-life operational systems. Discrete event simulation (DES) is an extremely popular and rapidly growing method of analysing networks of queues (Brailsford, Harper, Patel, & Pitt, 2009; Günal & Pidd, 2010; Robinson, 2005).

Reproducibility and replicability, described as "the cornerstone of cumulative science" (Sandve, Nekrutenko, Taylor & Hovig, 2013), is critical in order to assert correct results and build on the work of others (Hong, Crick, Gent, & Kotthoff, 2015; Sandve et al., 2013). In computational research, this can be achieved by following a number of best practices (Aberdour, 2007; Benureau & Rougier, 2017; Crick, Hall, Ishtiaq, & Takeda, 2014; Hong et al., 2015; Jiménez et al., 2017; Prlić & Procter, 2012; Sandve et al., 2013; Wilson et al., 2014). A popular software paradigm for research is open-source software which implies software source code that is freely usable and modifiable. In a recent review of open-source discrete event simulation software (Dagkakis & Heavey, 2016), 44 open-source discrete event simulation solutions were found and reviewed, however not all followed best practice: 14 were found to have no available documentation, and over half failed to use any version control. The paper did not consider automated testing of these packages.

This paper introduces the open-source Python library Ciw, which aims to enable best practices within the domain of discrete event simulation, and yield reproducible results. Ciw is a Python library for the simulation of open queueing networks. The core features of this library include the capability to simulate networks of queues (Jackson, 1957), multiple customer classes (Kelly, 1975) and restricted networks exhibiting blocking (Onvural & Perros, 1986). A number of other features are also implemented, including priorities (Cobham, 1954), baulking (Ancker & Gafarian, 1963a, 1963b), schedules (Doshi, 1986) and deadlock detection (Palmer, Harper, & Knight, 2018).

This paper is structured as follows: Section 2 will provide full motivation for the library's use and development, and Section 3 will outline the features currently implemented in the package. Then, in Section 4, we will briefly discuss the code's object-orientated structure and event-scheduling simulation algorithm, which will be followed by an example of Ciw's usage and syntax in Section 5. In Section 6, we will list how the library has been used in academic work to date, and finally, Section 7 will compare Ciw with five other simulation frameworks.

2. Motivation

Simulation options traditionally fall into discrete categories (Law, 2007; Robinson, 2014), consisting of programming languages, simulation packages and spreadsheet modelling. We consider simulation frameworks on a spectrum corresponding to the user interface. Such a spectrum is shown in Figure 1, with some suggested positions for a selection of simulation options, including Ciw, SimPy (Team SimPy, 2017), AnyLogic (The AnyLogic Company, 2017), SIMUL8 (SIMUL8 Corporation, 2017), as well as building a simulation in C++ and spreadsheet modelling. Note that spreadsheet modelling, despite usually being interfaced with a graphical user interface, is considered here a type of programming language due to its generic nature and syntax.

Advantages and disadvantages of these methods have

CONTACT Geraint I. Palmer palmergi1@cardiff.ac.uk

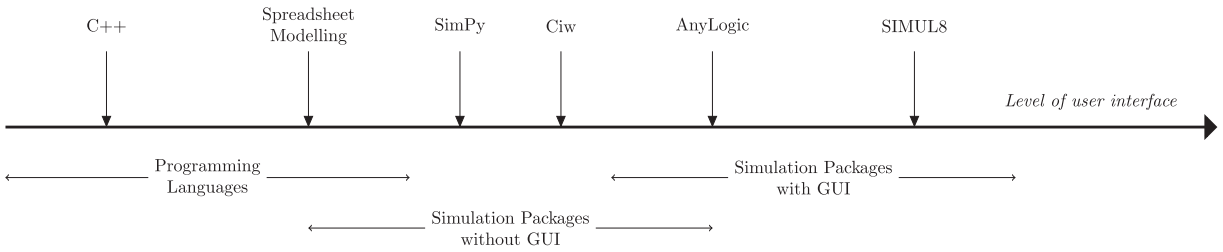

Figure 1. A suggested spectrum corresponding to user interface, with illustrative positioning for six simulation options.

been discussed extensively (Bell & O'Keefe, 1987; Dagkakis & Heavey, 2016; Law, 2007; Robinson, 2014). Programming from scratch is considered more flexible, is bespoke, may improve speed, and increases the variety of performance measures collected. However, a lack of user interface may hinder model communication. It is discussed in some literature that simulation packages, especially those with a graphical user interface (GUI), are more accessible, easily modifiable, and easier to communicate with non-specialists. GUIs can aid with conceptual modelling, model validation and verification (Bell & O'Keefe, 1987; Belton & Elder, 1994; Kirkpatrick & Bell, 1989).

However, some simulation packages come with several disadvantages for example high costs (licences, training, plug-ins and maintenance), they often lack modularity, low model reusability, and lack of access to the source code can impede understanding, customisation and flexibility. Furthermore, it is suggested (Bell & O'Keefe, 1987) that the addition of a GUI can lead to bad simulation practice. This includes: disregarding formal methodology such as statistical analysis in favour of watching animations; introducing bias in the model building process by building models that represent how a system should work instead of how they actually work and false model validation in which realistic graphics imply a realistic model.

Two themes arise when discussing research and software development: reproducibility and sustainability, and best practice (Aberdour, 2007; Benureau & Rougier, 2017; Crick et al., 2014; Jiménez et al., 2017; Hong et al., 2015; Prlić & Procter, 2012; Sandve et al., 2013; Wilson et al., 2014). The Ciw library aims to enable users to meet these standards in the domain of discrete event simulation.

In Kilgore (2001), three properties are listed as minimum requirements to ensure reproducibility in simulation:

- Readability
- Modularity
- Extendibility

All three properties can apply to Python (The Python Software Foundation, 2015), the ecosystem in which modelling with Ciw takes place. Python is an open-source, object-orientated, high-level language. Python's advantages as a language for developing simulation models are listed in Dagkakis and Heavey (2016). These include its intuitive and readable syntax, and potential to form a community of users and developers. In addition, Python is attractive to researchers due to the extensive collection of other scientific libraries available to integrate work, for example combining simulation with data analysis, machine learning and optimisation. An example of this as a teaching exercise with Ciw is seen in Knight, Palmer, and Glynatsi (2017). There are a number of popular scientific Python libraries, including:

- NumPy (Walt, Colbert, & Varoquaux, 2011) and SciPy (Jones et al., 2001) for scientific computation
- scikit-learn (Pedregosa et al., 2011) for machine learning and optimisation algorithms
- Pandas (McKinney, 2010) for data analysis tools
- Matplotlib (Hunter, 2007) and Seaborn (Waskom et al., 2014) for data visualisation
- SymPy (Meurer et al., 2017) for symbolic mathematics

In Wilson et al. (2014), the authors discuss the advantages of using high-level programming languages such as Python for research software over low-level languages like C and Fortran. Advantages include an increase in productivity when writing in high-level languages, better readability and rapid design decisions and prototyping. A downside however, due to the fact that Python is an interpreted language, is that the computational speed will not be as fast as compiled languages.

Another strength of Ciw is that it is completely open source. It has one of the most flexible and permissive licences, the MIT licence, and is written in an open-source language. This offers the user immediate advantages over commercial-off-the-shelf simulation packages. All source code is available for inspection, testing and modification. This enables and encourages greater understanding of the underlying methodology, increases model confidence, and provides an extendible framework in which discrete event simulation may be carried out. Furthermore, the elimination of licence fees and maintenance costs facilitate model sharing, open science and reproducibility. This overcomes common problems with commercial software with more strin-

gent licences, where models may sometimes not be shared between two computers with the same software.

This ecosystem provides an opportune way in which reproducible scientific research can be conducted:

- All manual data manipulation can be avoided (Crick et al., 2014; Sandve et al., 2013; Wilson et al., 2014).
- All raw data can be saved (Sandve et al., 2013).
- All models can be version controlled (Benureau & Rougier, 2017; Sandve et al., 2013; Wilson et al., 2014).
- All models can be scrutinised by automated testing (Benureau & Rougier, 2017; Wilson et al., 2014).
- All models can be shared (Benureau & Rougier, 2017; Crick et al., 2014; Hong et al., 2015; Jiménez et al., 2017; Sandve et al., 2013).

Ciw is also developed in a sustainable manner, and strives to follow best practice in research software development. This includes extensive testing (it has 100% test coverage (Batchelder, 2017)), comprehensive documentation, readability, modularisation, transparency and use of version control (Prlić & Procter, 2012; Wilson et al., 2014).

Object orientation, an important feature of Python, lends itself well to simulation (Dagkakis & Heavey, 2016; Law, 2007). In Dagkakis and Heavey (2016), the authors state that "DES is a traditional paradigm where object orientation is intuitively adopted". The argument for linking object orientation to one particular method of discrete event simulation, the three-phase approach, is given in Pidd (1995). Breaking a simulation down into events, activities and entities, as is required for the three-phase approach, is a form of modularisation itself. This equally applies to the similar event scheduling approach used in Ciw. In addition, simulation modellers habitually think of entities as belonging to a class, or classes, of similar entities. It is intuitive to build systems like this in an environment where modularisation is key, such as in an object-orientated programming language. Further advantages of using object orientation are listed in Law (2007): its flexibility, its ability to deal with complexity through modularisation, and its high reusability.

As stated previously, using open-source software provides distinct advantages over traditional commercial-off-the-shelf simulation packages. Similarly, open-source development can provide many advantages over closed source development. However, Dagkakis and Heavey (2016) argues that apart from eliminating the licence fee, simply being open source does not offer immediate advantages for developers, but it is the ethos and culture that comes with open source that provide the advantages. It is argued in Von Krogh and Von Hippel (2006) that open-source culture provides incentives to innovate, as there is no need for a large demand or promise of recoupment of financial investment for certain features to be developed. That is, private needs create public goods. This has been evident in Ciw, where new research can be directly implemented into the software and tested and experimented quickly (see Section 6). New features have been implemented after discussions with users from around the world via the online issue tracker. Freedom of development is another crucial aspect of open source according to Von Krogh and Von Hippel (2006), where users can fork and develop their own versions of software for their bespoke needs. An argument for promoting best practice in open-source software is given in Aberdour (2007), as it achieves better quality software. Some of these best practices arise naturally in an open-source environment, for example rapid release cycles, code reviews and code modularity. Open-source development actually encourages these best practices due to its transparency and the opportunities for developers to showcase their work (Jiménez et al., 2017).

A review of open-source discrete event simulation software is given in Dagkakis and Heavey (2016), and as mentioned previously, a number of frameworks failed to follow best practice in their development. Three Python libraries were found, though only one was found to meet the quality requirements of the study, SimPy (2017). Further, Kilgore (2001) draws many parallels between open-source development and simulation modelling, while concluding that the "steady, long-term progress toward libraries of easily extendible and easily reusable simulation code" is an important direction for simulation modellers.

To summarise, Ciw is an object-orientated, open-source Python library with the following qualities:

- Open, accessible source code promotes understanding, development and modification. Online issue tracker and open development environment fosters discussion, idea generation and development. Permissive licence allows it to be extended, modified and shaped to the users' needs.
- Code development follows best practice guidelines for reproducibility, code quality and modification. Modularity allows modification and extension through inheritance.
- Python ecosystem allows it to be used flexibly within the programming language, allowing ease of experimentation and integration with other scientific tools. Models can be tested and version controlled.
- Models are readable and the package has extensive bilingual documentation, to enhance model communication.

3. Features

Ciw's main functionality is the simulation of open restricted queueing networks that exhibit blocking, and supports multiple classes of customer:

- A *queueing network* is a system consisting of a number of service centres where customers may wait in a queue for service; connected by a transition matrix of probabilities r_{ij}, the probability of joining node j after completing service at node i.
- A queueing network is described as *open* if customers can leave the system, and new customers can arrive from outside the system (Stewart, 2009).
- A queueing network is described as *restricted* if nodes have limited queueing capacity, that is, only room for a certain amount of customers to wait at any one time. If a node's queueing capacity is full, then external arrivals are rejected, and *Type I blocking* (Onvural & Perros, 1986) occurs for customers transitioning from other nodes. That is, after service they remain with their server until space becomes available at their destination node, while that server is unavailable to serve any other customer.
- *Multiple classes of customer* refers to the possibility of having more than one type of customer using and sharing the same resources, but using them in different ways. For example, each class of customer may have its own distinct inter-arrival time distributions, service time distributions and transition matrices.

In addition to these main properties, Ciw can simulate a number of other features:

- *A choice of inter-arrival and service time distributions*: Including Uniform, Deterministic, Triangular, Exponential, Gamma, Truncated Normal, Lognormal, Weibull and the possibility of users defining their own Discrete, Continuous, Empirical, Sequential and Time-Dependant distributions.
- *Batch arrivals:* At each external arrival, a number of customers may arrive simultaneously. Ciw allows sampling from a discrete probability distribution to obtain batch sizes.
- *Priority classes:* A mapping from customer classes to priority levels. This allows customers with higher priority to jump ahead of customers with lower priority each time they enter a queue.
- *Baulking customers:* At each external arrival, customers have a probability $b(m)$ of baulking (choosing not to join the system), given that there are m customers already at that node. Ciw allows users to define their own baulking function $b(m)$ as a Python function.
- *Server schedules:* Cyclic server schedules may be defined for each node, that is the number of servers at a node may increase or decrease as servers go on and off duty at fixed times during the simulation run.
- *Dynamic customer classes:* After service at a node, customers may re-sample their customer class according the a class change matrix of probabilities p_{ij}, the probability of a customer of class i becoming a customer of class j after service at that node. This means that their behaviour (service distributions, transition matrices and priorities) will also change.
- *Deadlock detection:* Restricted queueing networks with cycles can cause the phenomenon of deadlock (Palmer et al., 2018). Traditionally deadlocks are difficult to detect, however Ciw has the ability to terminate a simulation run once deadlock has been reached, using the state digraph method.

Ciw also offers a number of termination conditions:

- Simulating until a maximum amount of time has passed.
- Simulating until a maximum number of customers have arrived/been accepted/finished.
- Simulating until deadlock has been reached.

Due to the modular nature of Ciw, a number of further features can be implemented through the use of inheritance of Ciw's classes, such as the ability to send customers to the shortest queue. Ciw is also a continuously developed library, and so the list of features is growing.

4. Architecture

Ciw makes full use of Python's object-orientated nature:

- A `Simulation` object is a one-use object used for one run of a simulation, which contains a network of `Node` objects.
- Each `Node` object contains `Server` objects.
- `Individual` objects are passed around the network of `Node` objects, where they wait to be served by `Server` objects.
- Each `Individual` object carries a number of named tuples that record the history of a single service.
- The `ArrivalNode` creates new `Individual` objects to enter the simulation.
- The `ExitNode` collects `Individuals` that leave the system.
- `Network` objects, that also consist of `ServiceCentre` and `CustomerClass` objects, define a queueing network in order to create a `Simulation` object.

Figure 2 summarises and categorises the interconnecting objects that make up the Ciw framework. Some optional objects, state trackers and deadlock detectors, are also used for some features.

Ciw uses the event scheduling approach (Robinson, 2014), similar to the popular three-phase approach. This deviates significantly from the other major Python alternative, SimPy, that uses the process-based approach.

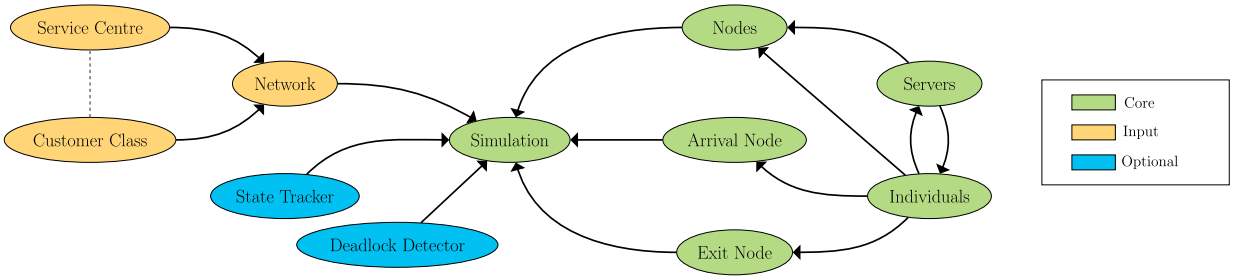

Figure 2. Ciw's architecture.

In the event scheduling approach, three types of event take place: A-Events move the clock forward, B-Events are pre-scheduled events and C-Events are events that arise because a B-Event has occurred.

Here A-Events correspond to moving the clock forward to the next B-Event. B-Events correspond to either an external arrival, a customer finishing service or a server shift change. C-Events correspond to a customer starting service, customer being released from a node and being blocked or unblocked.

In event-scheduling, the following process occurs:

1. Initialise the simulation.
2. A-Phase: move the clock to the next scheduled event.
3. Take a B-Event scheduled for now, carry out the event.
4. Carry out all C-Events that arose due to the event carried out in 3.
5. Repeat 3 - 4 until all B-Event scheduled for that date have been carried out.
6. Repeat 2 - 5 until a terminating criteria has been satisfied.

Each `Node` object, including the `ArrivalNode`, has a `have_event` method and a `next_event_date` attribute. The `next_event_date` attribute is updated each time an event occurs, and corresponds to the date that the next B-Event at that object is scheduled to happen. At the A-Phase, the simulation's clock is moved to the next `next_event_date` of all the `Node` objects.

The `ArrivalNode`'s `have_event` method spawns a new `Individual` and sends them to their appropriate `Node` (barring baulking or exceeding node capacity). All other `Nodes`' `have_event` method consist of one of two events: an `Individual` finishing service at that node, or a shift change for the `Servers`.

C-Events are not coded explicitly, but follow on naturally as consequences of the B-Event described above. For example when the `ArrivalNode` spawns a new `Individual` and is successfully sent to a `Node`, if there is not another individual waiting in the queue, that individual is attached to a `Server` and begins service. Similarly when an `Individual` finishes service, that individual is sent to another `Node`, and may join a queue or begin service if a server is free, while a waiting customer now begins service with the freed `Server` at the current `Node`.

5. Illustrative use

The library is installable from the Python package index, which means it is readily available to anyone with Python (The Python Software Foundation, 2015) (versions 2.7, 3.4 and above) and an internet connection. The source code is available on GitHub: https://github.com/CiwPython/Ciw, under the MIT licence. Full documentation is available (both in English and Welsh, in pdf and html form), and hosted on Read The Docs: http://ciw.readthedocs.io/.

In order to demonstrate usage in more detail, consider the following system:

- Two classes of jobs enter a computer repair clinic: scheduled jobs (S) and unscheduled jobs (U).
- Scheduled jobs arrive in batches of two, once per hour.
- Unscheduled jobs arrive randomly according to a Poisson distribution, at rate one per hour.
- Unscheduled jobs take priority over scheduled jobs, and thus join the queue ahead of scheduled jobs.
- The repair clinic consists of two nodes: an inspection desk with two servers where jobs arrive, and a repair room with one server.
- There is infinite queueing capacity at the inspection desk, but no queueing capacity between the inspection desk and the repair room, thus Type I blocking occurs here.
- All service times follow Exponential distributions: scheduled jobs take an average of 20 minutes to be inspected and 60 minutes to repair, unscheduled jobs take an average of 40 minutes to be inspected and 90 minutes to repair.
- 5% of all scheduled jobs require repair and 40% of all unscheduled jobs require repair.

The system is shown in Figure 3. The repair clinic runs for 24 hours a day. The example below will run a simulation of this system using Ciw and will obtain estimates for the values of:

- The average waiting time of unscheduled jobs at the inspection desk.
- The average time a job is spent blocked at the inspection desk (regardless of job class).

The code shown in Figure 4 gives the code needed to create the `Network` object that defines the system above. The code in Figure 5 runs the simulation over 20 trials, for 7 days, with a warm-up time of 1 day.

6. Use cases

To date, Ciw has been used for various theoretical, practical and pedagogic applications, including:

- Theoretical work investigating deadlock in open restricted queueing networks in Palmer et al. (2018). A graph theoretical technique to detect deadlock during a run of a discrete event simulation was developed, and incorporated into the Ciw framework: http://ciw.readthedocs.io/en/latest/Guides/deadlock.html. Experiments on the time to reach deadlock were undertaken.
- The modelling of an ophthalmology clinic at Cardiff and Value University Health Board was undertaken by Morgan, J. and Hölscher, L., in order to investigate the best patient scheduling strategy. This project was essential to the development of Ciw, as many features were added to the library due to the requirements of the project: server schedules, dynamic customer classes and exact arithmetic.
- Models of cancer patient diagnoses were built by Harper, P.R. and Palmer, G.I. for the Wales Cancer Network and Cwm Taf University Health Board. These models and what-if scenarios were used to advise national policy on capacity increases for diagnostic tests in Wales in order to reach potential Welsh government set cancer diagnosis time targets.
- A Nuffield research placement (The Nuffield Foundation, 2017) student Huang, C. undertook research with Knight V.A. and Palmer, G.I. studying deadlock in queueing networks, extending previous experiments to include baulking customers and servers taking vacations.
- The library has been used as part of a 2 day 'hackathon' as part of the MSc in Operational Research and Applied Statistics at Cardiff University. The hackathon aims to increase familiarity with object-orientated programming by working on a Python project. In 2017, the project was to write a genetic algorithm to best configure three queues in series, using Ciw to obtain the cost function. An example solution is given in Knight et al. (2017).

7. Comparison with other simulation frameworks

Five other popular simulation frameworks from across the spectrum corresponding to user interface (see Figure 1) were chosen for comparison. They are compared on their appropriateness for conducting reproducible research in the domain of discrete event simulation. The frameworks chosen were

- C++ (version 11, compiled using g++ 4.2.1)
- Spreadsheet modelling (implemented in Microsoft Excel 2013)
- SimPy (version 3.0.10, using Python 3.5.1)
- Ciw (version 1.1.3, using Python 3.5.1)
- AnyLogic (AnyLogic 8 University 8.1.0)
- SIMUL8 (SIMUL8 2014 [Exclusive EDUCATIONAL SITE Edition])

A model of an $M/M/3$ queue (Stewart, 2009), with arrival rate $\lambda = 10$, and service rate $\mu = 4$ was built in each framework. The models were run for 800 time units, with a warmup time of 100 time units. The aim was to find the average waiting time in the queue.

The C++, SimPy and Ciw models can be found in Appendices 1, 2 and 3. For the purpose of this paper, Microsoft Excel was chosen as the spreadsheet software. Screenshots of the spreadsheet, SIMUL8 and AnyLogic models are shown in Figures 6–8, respectively. All models are archived and can be found at Palmer, Hawa, Knight, and Harper (2017). Some initial observations are summarised in Table 1.

The capabilities of a spreadsheet model were not found to align with the expectations of research best practice. Seeds cannot be set, thus reproducibility is impossible. The resulting model has very low interpretability, unless set out as a "black box" model where parameters are input and results are output. However, this style of model would hinder model understanding and communication. In fact the model, including input, output and mechanics, are all shown in Figure 6, and yet the model is still very difficult to interpret. This in turn leads to low confidence in results, a well reported phenomenon (Ziemann, Eren, & El-Osta, 2016) with the use of spreadsheets. In addition, most data had to be handled manually, further impeding reproducibility.

Building a model with a GUI, such as with SIMUL8 and AnyLogic, may ease the model development process although care should be taken to follow best practice. The model may be more accessible given its visual nature which can aid with communication, although knowledge of the software is required to read much of the model parameters as many of these are hidden behind objects and menus (for example, Figures 7 and 8 do not in themselves show basic model parameters such as number of servers, arrival and service rates).

Table 1. Summary of the comparisons between six simulation frameworks.

	Version controllable	Licence	Modifiable	GUI	Animation	Support
C++	✓	GNU GPL free licence	Bespoke models	N/A	N/A	N/A
Spreadsheet modelling	✗	Depends on software	Limitations to what can be modelled	N/A	✗	N/A
SimPy	✓	MIT	Extensible & modifiable source code	Limited GUI available for running models	✗	Online documentation
Ciw	✓	MIT	Extensible & modifiable source code	✗	✗	Online documentation
AnyLogic	With professional licence only	Limited PLE version available, otherwise commercial	Extend with Java	✓	✓	Online documentation & paid training
SIMUL8	✗	Commercial	Extend with visual logic	✓	✓	Online documentation & paid training

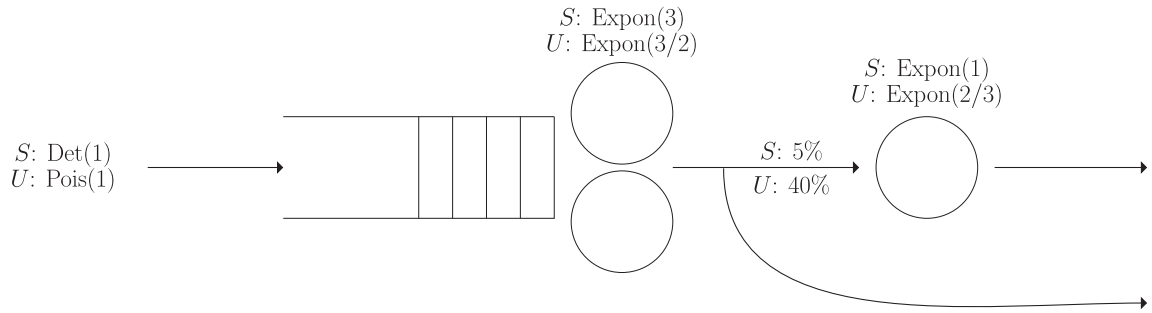

Figure 3. Diagrammatic representation of the repair clinic example. *S* denotes scheduled jobs, and *U* denotes unscheduled jobs.

```
>>> import ciw
>>> assert ciw.__version__ == '1.1.3'

>>> N = ciw.create_network(
...     Arrival_distributions={
...         'Class 0': [['Deterministic', 1.0],
...                     'NoArrivals'],
...         'Class 1': [['Exponential', 1.0],
...                      'NoArrivals']},
...     Service_distributions={
...         'Class 0': [['Exponential', 3.0],
...                     ['Exponential', 1.0]],
...         'Class 1': [['Exponential', 3.0/2.0],
...                     ['Exponential', 2.0/3.0]]},
...     Transition_matrices={
...         'Class 0': [[0.0, 0.05],
...                     [0.0, 0.0]],
...         'Class 1': [[0.0, 0.4],
...                     [0.0, 0.0]]},
...     Batching_distributions={
...         'Class 0': [['Deterministic', 2],
...                     ['Deterministic', 1]],
...         'Class 1': [['Deterministic', 1],
...                     ['Deterministic', 1]]},
...     Priority_classes={
...         'Class 0': 1,
...         'Class 1': 0},
...     Queue_capacities=['Inf', 0],
...     Number_of_servers=[2, 1]
... )
```

Figure 4. Ciw code required to create the `Network` object for the example system.

The binary files which represent the SIMUL8 models and the restrictive commercial licences on both SIMUL8 and AnyLogic inhibit accessibility and model sharing, and thus reproducibility.

The bespoke model developed in C++, given its compiled nature and that the model was not held back by unused features, used the least (by far) computation time. The model (a script) is shareable and can be put under version control. Readability is hindered here as all details, including internal simulation mechanisms, are shown which may make communications with non-specialists challenging.

The models developed with SimPy and Ciw are version controllable and shareable, and thus ideal for replicable results. Compared to the model developed in C++, much of the simulation mechanics are hidden from the user and pre-tested. This aids with model communication, validation and reduces error. The authors suggest that the Ciw model is more readable than the SimPy version. Much of the simulation mechanics are on show with SimPy, which increases its flexibility but reduces readability, whereas Ciw prioritises readability.

Runtime analysis was carried out on the software for which it was possible: C++, SimPy, Ciw and AnyLogic. All analyses were performed on an Apple MacBook Air with 1.4GHz Intel Core i5 processor, OS X 10.9.5 (13F34), with 4 GB 1600 MHz DDR3 memory. SIMUL8 could not be included in the analysis; due to licensing restrictions, the model could not be run on the same machine as the others (for consistency). Importantly,

```
>>> def run_trial(seed, N):
...     ciw.seed(seed)
...     Q = ciw.Simulation(N)
...     Q.simulate_until_max_time(24*8)
...     recs = Q.get_all_records()
...     waits_unscheduled = [r.waiting_time for r in recs
...                          if r.customer_class==1
...                          if r.node==1
...                          if r.arrival_date > 24]
...     servers_blocked = [r.time_blocked for r in recs
...                        if r.node==1
...                        if r.arrival_date > 24]
...     average_wait = sum(waits_unscheduled) / len(waits_unscheduled)
...     average_blocked = sum(servers_blocked) / len(servers_blocked)
...     return average_wait, average_blocked

>>> waits = []
>>> blocked = []
>>> for seed in range(20):
...     average_wait, average_blocked = run_trial(seed, N)
...     waits.append(average_wait)
...     blocked.append(average_blocked)

>>> print(sum(waits) / len(waits))
2.08127848232403

>>> print(sum(blocked) / len(blocked))
0.23280830359597315
```

Figure 5. Ciw code used to run the example system over 20 trials, and obtain the average waiting time for unscheduled jobs, and the average time blocked for all jobs.

	A	B	C	D	E	F	G
1	Parameters					Results	
2							
3	Arrival rate	10				Trial	Mean Wait
4	Service rate	4				1	0.303224
5	Number of servers	3				2	0.170857
6	Max Simulation Time	800				3	0.3949
7	Warmup	100				4	0.494869
8						5	1.261468
9						6	0.290787
10						7	0.145116
11						8	0.200018
12						9	0.324214
13						10	0.341543
14						11	0.268243
15						12	0.290829
16						13	0.205088
17						14	0.596678
18						15	0.12989
19						16	0.151971
20						17	0.272161
21						18	0.318548
22						19	0.30651
23						20	0.843754

(a)

	A	B	C	D	E	F	G
1	Arrival Date	Service Time	Service Start Date	End Date	Wait	Include	Wait*Include
2				0			
3				0			
4				0			
5	0	0.3448766	0	0.34488	0	0	0
6	0.0137547	0.2572122	0.01375466	0.27097	0	0	0
7	0.1596319	0.1017449	0.159631899	0.26138	0	0	0
8	0.1996071	0.2825094	0.261376776	0.54389	0.06177	0	0
9	0.2874161	0.0913433	0.287416095	0.37876	0	0	0
10	0.3691407	0.453877	0.369140658	0.82302	0	0	0
11	0.6465689	0.1575984	0.646568948	0.80417	0	0	0
12	0.7559878	0.5457062	0.755987759	1.30169	0	0	0
13	1.0385525	0.0620693	1.038552508	1.10062	0	0	0
14	1.0749212	0.3488628	1.074921206	1.42378	0	0	0
15	1.1397818	0.103078	1.139781823	1.24286	0	0	0
16	1.2203849	0.0843532	1.242859819	1.32721	0.02247	0	0
17	1.2691663	0.2803188	1.301693996	1.58201	0.03253	0	0
18	1.3074656	0.2850337	1.32721301	1.61225	0.01975	0	0
19	1.3336224	0.3055875	1.423783988	1.72937	0.09016	0	0
20	1.6726619	0.000902	1.672661886	1.67356	0	0	0
21	1.805258	0.0145591	1.805258023	1.81982	0	0	0
22	1.9285575	0.0467061	1.928557505	1.97526	0	0	0
23	1.9532835	0.1331858	1.953283474	2.08647	0	0	0
24	2.0832097	0.0570247	2.083209691	2.14023	0	0	0
25	2.1146871	0.5448817	2.114687113	2.65957	0	0	0
26	2.122529	0.0796813	2.122528956	2.20221	0	0	0

(b)

Figure 6. Screenshots of the spreadsheet model developed in Microsoft Excel: (a) shows parameters and results; (b) shows the mechanics of one trial.

the version used does not include tools to record computational runtime. Figure 9 shows the average runtimes from five runs, as a ratio of the fastest running model C++, for maximum simulation times from 200 to 5000 time units. Also included in the analysis is a bespoke Python model (an equivalent model to the C++ model but coded in Python). The three Python models were run through two interpreters: the standard CPython interpreter (version 3.5.1) and PyPy (2017) (version 5.1.1), which is an alternative implementation

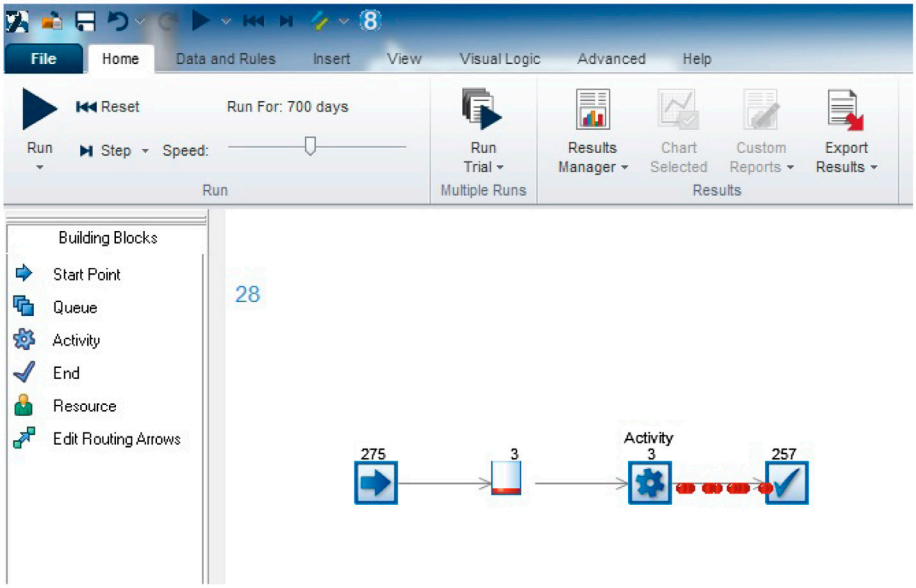

Figure 7. Screenshot of the SIMUL8 model.

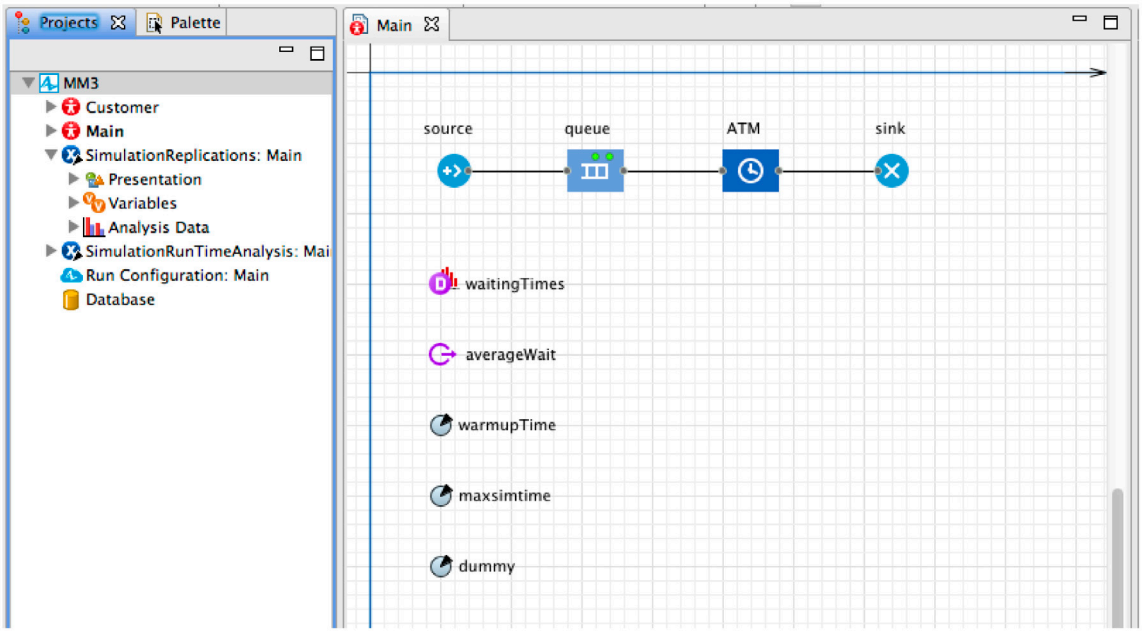

Figure 8. Screenshot of the AnyLogic model.

Table 2. Pros and cons of using Ciw.

Pros	Cons
• Free (no licence fee, maintenance, or training costs) • Source code open source, thus available for understanding and modification • Full documentation, including tutorials, available online • Fully and openly tested, giving confidence in use • Scripting environment offers flexibility in experimentation and results analysis • Can be used in conjunction with other scientific Python tools • Readable models • Documented version control enables sustainability • Continuous development • Enables testable and version controlled models	• Some features not available yet • No GUI or animation currently implemented • Execution speed compromised for readability

of Python with a Just In Time compiler that improves runtimes.

It is worth noting that the runtime recorded for the AnyLogic model does not take into consideration

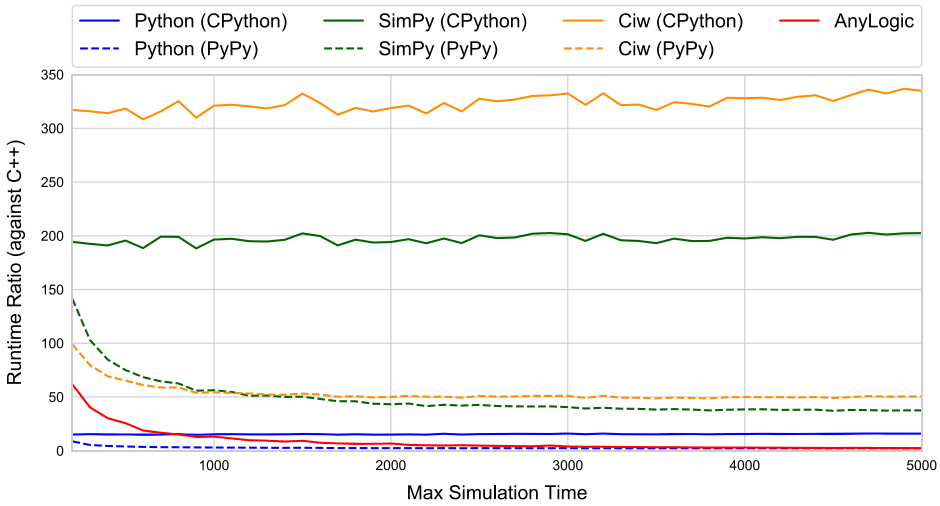

Figure 9. Comparison between the runtimes of the C++ model, bespoke Python model, the SimPy model, the Ciw model and the AnyLogic model.

experiment initialisation time, nor does it include application launch and model loading. Therefore in practice, running AnyLogic models would take longer than what is shown in the plot.

Figure 9 reflects what has been discussed in much of the literature (Law, 2007; Robinson, 2014) that bespoke models coded using programming languages yield faster running models than simulation packages. The simulation packages here, SimPy and Ciw, carry around many unused features that may slow down the model. Running Ciw with PyPy greatly improves performance and we can see that PyPy seems to be mainly affected by the simulation initialisation time, as running the simulation for longer sees the performance approaching that of the C++ model. Furthermore, using Python's inbuilt parallel processing library, it is straightforward to parallelise trials of Ciw. Speed, however, is not a priority of Ciw, favouring instead code readability, and the library's ability to enable reproducible scientific research. Future development of Ciw may involve performance improvement, as long as this does not impact readability. In practice, the computational runtime for Ciw models does not hinder practical use in research, though may be disadvantageous in complex systems or other models where speed is a priority.

8. Summary

This paper has introduced the Python library Ciw for discrete event simulations of open queueing networks. The library strives to allow best practices in research software and enable reproducibility of simulation models. Ciw has a growing number of features intended to be able to model more complex systems and ensure that models reflect reality. It offers advantages over both commercial off the shelf solutions and programming from scratch.

Section 2 discussed a number of properties of open-source software that can enable reproducible simulation research. Ciw not only offers these properties, but is built with these at the forefront of its design. These include modular and extendible components, readable syntax and a permissive licence. Its scripted nature and its place in the Python ecosystem allows a number of best practices for reproducible research, including testing of simulation models, version controllable models and integration with other open scientific tools. Table 2 summarises the pros and cons of using Ciw.

In some aspects, the lack of a GUI could be considered to make models less interpretable to non-specialists; however, Ciw's readable model framework offers many advantages for collaboration and reusability. Given that Ciw is built in Python, it is malleable in that it can be combined with other bespoke functionality with ease. For example, a user can use inheritance to change the behaviour of a particular aspect of the system. This implies that Ciw allows for multi method simulation: for example, it can be combined with agent-based models (inherent to the object-orientated nature of Python) or combined with some of the ordinary differential equation solvers in the Python ecosystem (SciPy) to allow for a combination with system dynamics (Robinson, 2014).

Section 6 listed a number of uses of the library. In each of these cases Ciw enabled best practices in reproducibility, allowing other researchers to promptly re-run, scrutinise and build upon the research. It is hoped others, both academic researchers and practitioners, will make use of Ciw to conduct reproducible simulations, and also contribute to the library's development and help add to its growing functionality.

Acknowledgements

The authors would like to express gratitude to the anonymous referees whose comments have improved this work. The authors would like to thank Syaribah Brice and Mark Tuson for their discussions and advice regarding the AnyLogic model. The authors would also like to thank all contributors to the Ciw project, those who have written code, and colleagues and users of the library who have generated ideas and contributed to valuable discussions on the library's development. These include: Geraint Palmer, Vincent Knight, Lieke Hölscher, Sam Luen-English, Nikoleta Glynatsi, Adam Johnson, Alex Carney, Paul Harper and Jennifer Morgan, and a number of users who the authors have only interacted with online. A variety of software libraries have been used in this work: (i) The networkx library (graph theory) (Schult & Swart, 2008); (ii) The hypothesis library (property-based testing) (MacIver, 2017); (iii) The tqdm library (progress bars) (da Costa-Luis et al., 2017); (iv) The PyYAML library (yaml parsing) (The PyYAML library developers, 2017); (v) The Pandas library (data analysis) (McKinney, 2010); (vi) The matplotlib library (data visualisation) (Hunter, 2007).

Disclosure statement

No potential conflict of interest was reported by the authors.

References

Aberdour, M. (2007). Achieving quality in open-source software. *IEEE Software, 24*(1).

Ancker, Jr., C., & Gafarian, A. (1963a). Some queueing problems with balking and reneging i. *Operations Research, 11*(1), 88–100.

Ancker, Jr., C., & Gafarian, A. (1963b). Some queueing problems with balking and reneging ii. *Operations Research, 11*(6), 928–937.

Batchelder, N. (2017). *Coverage*. Retrieved from https://coverage.readthedocs.org/

Bell, P., & O'Keefe, R. (1987). Visual interactive simulation history, recent developments, and major issues. *Simulation, 49*(3), 109–116.

Belton, V., & Elder, M. (1994). Decision support systems: Learning from visual interactive modelling. *Decision support systems, 12*(4–5), 355–364.

Benureau, F., & Rougier, N. (2017). *Re-run, repeat, reproduce, reuse, replicate: Transforming code into scientific contributions*. arXiv preprint arXiv:1708.08205.

Brailsford, S., Harper, P. R., Patel, B., & Pitt, M. (2009). An analysis of the academic literature on simulation and modelling in health care. *Journal of Simulation, 3*(3), 130–140.

Cobham, A. (1954). Priority assignment in waiting line problems. *Operations Research, 2*(1), 70–76.

Crick, T., Hall, B.A., Ishtiaq, S., & Takeda, K. (2014). Share and enjoy: Publishing useful and usable scientific models. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing* (pp. 957–961). IEEE Computer Society.

da Costa-Luis, C., Stephen, l., Mary, H., noamraph, Korobov, M., Ivanov, I., ..., Umer, A. (2017). *tqdm/tqdm: tqdm v4.19.5 stable*.

Dagkakis, G., & Heavey, C. (2016). A review of open source discrete event simulation software for operations research. *Journal of Simulation, 10*(3), 193–206.

Doshi, B. T. (1986). Queueing systems with vacationsa survey. *Queueing Systems, 1*(1), 29–66.

Günal, M. M., & Pidd, M. (2010). Discrete event simulation for performance modelling in health care: A review of the literature. *Journal of Simulation, 4*(1), 42–51.

Hong, N., Crick, T., Gent, I., & Kotthoff, L. (2015). *Top tips to make your research irreproducible*. arXiv preprint arXiv:1504.00062.

Hunter, J. D. (2007). Matplotlib: A 2d graphics environment. *Computing in Science & Engineering, 9*(3), 90–95.

Jackson, J. (1957). Networks of waiting lines. *Operations Research, 5*(4), 518–521.

Jiménez, R. C., Kuzak, M., Alhamdoosh, M., Barker, M., Batut, B., Borg, M., ... Watson-Haigh, N. S.(2017). Four simple recommendations to encourage best practices in research software. *F1000Research*, 6.

Jones, E., Oliphant, T., Peterson, P., & The SciPy library developers, (2001). *SciPy: Open source scientific tools for Python*. Retrieved from https://scipy.org/citing.html

Kelly, F. (1975). Networks of queues with customers of different types. *Journal of Applied Probability, 12*(3), 542–554.

Kilgore, R. A. (2001). Open source simulation modeling language (sml). In *Proceedings of the 33nd conference on Winter simulation* (pp. 607–613). IEEE Computer Society.

Kirkpatrick, P., & Bell, P. (1989). Simulation modelling: A comparison of visual interactive and traditional approaches. *European Journal of Operational Research, 39*(2), 138–149.

Knight, V. A., Palmer, G. I., & Glynatsi, N. E. (2017). *Genetic algorithm exercise for an MSc hackathon using Ciw*, doi:10.5281/zenodo.836243.

Law, A. M. (2007). *Simulation modeling and analysis*. McGraw-Hill.

MacIver, D. (2017). *Hypothesis*. Retrieved from https://hypothesis.readthedocs.org/

McKinney, W. (2010). Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference* (Vol. 445, pp. 51–56). Austin, TX: SciPy.

Meurer, A., Smith, C., Paprocki, M., Čertík, O., Kirpichev, S., Rocklin, M., ... Scopatz, A. (2017). Sympy: Symbolic computing in python. *PeerJ Computer Science, 3*, e103.

Onvural, R., & Perros, H. (1986). On equivalencies of blocking mechanisms in queueing networks with blocking. *Operations Research Letters, 5*(6), 293–297.

Palmer, G. I., Harper, P. R., & Knight, V. A. (2018). Modelling deadlock in open restricted queueing networks. *European Journal of Operational Research, 266*(2), 609–621.

Palmer, G. I., Hawa, A. L., Knight, V. A., & Harper, P. R. (2017). *M/M/3 simulation models for the paper "Ciw: An open source discrete event simulation library"*. doi:10.5281/zenodo.848644.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research, 12*, 2825–2830.

Pidd, M. (1995). Object-orientation, discrete simulation and the three-phase approach. *The Journal of the Operational Research Society, 46*(3), 362–374.

Prlić, A., & Procter, J. (2012). Ten simple rules for the open development of scientific software. *PLoS Computational Biology, 8*(12).

Robinson, S. (2005). Discrete-event simulation: From the pioneers to the present, what next? *Journal of the Operational Research Society, 56*(6), 619–629.

Robinson, S. (2014). *Simulation: The practice of model development and use*. Palgrave Macmillan.

Sandve, G., Nekrutenko, A., Taylor, J., & Hovig, E. (2013). Ten simple rules for reproducible computational research. *PLoS Computational Biology, 9*(10), 1–4.

Schult, D.A., & Swart, P. (2008). Exploring network structure, dynamics, and function using networkx. In *Proceedings of the 7th Python in Science Conferences (SciPy 2008)*. (Vol. 2008pp. 11–16).

SIMUL8 Corporation. (2017). *Simul8*. Retrieved from https://www.simul8.com/

Stewart, W. (2009). *Probability, markov chains, queues, and simulation*. Princeton University Press.

Team SimPy. (2017). *Simpy*. Retrieved from https://simpy.readthedocs.io/

The AnyLogic Company. (2017). *Anylogic*. Retrieved from https://www.anylogic.com/

The Nuffield Foundation. (2017). *Nuffield research placements*. Retrieved from http://www.nuffieldfoundation.org/nuffield-research-placements

The PyPy developers. (2017). *Pypy*. Retrieved from https://pypy.org/

The Python Software Foundation. (2015). *Python 3.5.1*. Retrieved from www.python.org

The PyYAML Library Developers. (2017). *Pyyaml*. Retrieved from http://pyyaml.org/

Von Krogh, G., & Von Hippel, E. (2006). The promise of research on open source software. *Management Science, 52*(7), 975–983.

Walt, S., Colbert, V. D. S. C., & Varoquaux, G. (2011). The numpy array: A structure for efficient numerical computation. *Computing in Science & Engineering*, *13*(2), 22–30.

Waskom, M., Botvinnik, O., Hobson, P., Cole, J.B., Halchenko, Y., Hoyer, S., ... Ziegler, E. (2014). *Seaborn: v0.5.0*.

Wilson, G., Aruliah, D., Brown, C. T., Hong, N., Davis, M., Guy, R., ... Wilson, P. (2014). Best practices for scientific computing. *PLoS biology, 12*(1), e1001745.

Ziemann, M., Eren, Y., & El-Osta, A. (2016). Gene name errors are widespread in the scientific literature. *Genome Biology, 17*(1), 177.

Appendix 1. C++ model

```
#include <iostream>
#include <vector>
#include <cmath>
#include <algorithm>
using namespace std;

double runTrial(int seed, double arrivalRate, double serviceRate, int numberOfServers,
↪  double maxSimTime, double warmup){
  int i;
  double outcome, r1, r2, serviceTime, serviceStartDate, serviceEndDate, wait;
  double sumWaits = 0.0, arrivalDate = 0.0;
  vector<double> serversEnd;
  vector<double> temp;
  vector<double> waits;
  vector<vector<double> > records;
  srand(seed);

  for(i = 0; i < numberOfServers; ++i){
    serversEnd.push_back(0);
  }

  while(arrivalDate < maxSimTime){
    r1 = ((double)rand() / (double)RAND_MAX);
    r2 = ((double)rand() / (double)RAND_MAX);
    while (r1 == 0.0 || r2 == 0.0 || r1 == 1.0 || r2 == 1.0){
      r1 = ((double)rand() / (double)RAND_MAX);
      r2 = ((double)rand() / (double)RAND_MAX);
    }
    arrivalDate += (-log(r1))/arrivalRate;
    serviceTime = (-log(r2))/serviceRate;
    serviceStartDate = max(arrivalDate, (*min_element(serversEnd.begin(),
    ↪  serversEnd.end())));
    serviceEndDate = serviceStartDate + serviceTime;
    wait = serviceStartDate - arrivalDate;
    serversEnd.push_back(serviceEndDate);
    serversEnd.erase(min_element(serversEnd.begin(), serversEnd.end()));
    temp.push_back(arrivalDate);
    temp.push_back(wait);
    records.push_back(temp);
    temp.clear();
  }

  for(i = 0; i < records.size(); ++i){
    if(records[i][0] > warmup){
      waits.push_back(records[i][1]);
    }
  }


  for(i = 0; i < waits.size(); ++i){
    sumWaits += waits[i];
  }

  outcome = (sumWaits) / (waits.size());
  return outcome;
}

int main(int argc, char **argv){
  int i, seed;
  double solution;
  int numberOfServers = 3;
  int numberOfTrials = 20;
  double arrivalRate = 10.0;
  double serviceRate = 4.0;
  double maxSimTime = 800.0;
  double warmup = 100.0;
  double sumMeanWaits = 0.0;
  vector<double> meanWaits;

  for(seed = 0; seed < numberOfTrials; ++seed ){
    meanWaits.push_back(runTrial(seed, arrivalRate, serviceRate, numberOfServers,
    ↪  maxSimTime, warmup));
  }

  for(i = 0; i < meanWaits.size(); ++i){
    sumMeanWaits += meanWaits[i];
  }
  solution = (sumMeanWaits) / (meanWaits.size());
}
```

Appendix 2. SimPy model

```
import simpy
import random

arrival_rate = 10.0
number_of_servers = 3
service_rate = 4.0
max_simulation_time = 800
warmup = 100
num_trials = 20

def source(env, arrival_rate, service_rate, server, records):
    """Source generates customers randomly"""
    while True:
        c = customer(env, server, service_rate, records)
        env.process(c)
        t = random.expovariate(arrival_rate)
        yield env.timeout(t)

def customer(env, server, service_rate, records):
    """Customer arrives, is served and leaves."""
    arrive = env.now
    with server.request() as req:
        results = yield req
        wait = env.now - arrive
        records.append((env.now, wait))
        tib = random.expovariate(service_rate)
        yield env.timeout(tib)

def run_trial(seed, arrival_rate, service_rate, number_of_servers,
              max_simulation_time, warmup):
    """Run one trial of the simulation, returning the average waiting time"""
    random.seed(seed)
    records = []
    env = simpy.Environment()
    server = simpy.Resource(env, capacity=number_of_servers)
    env.process(source(env, arrival_rate, service_rate, server, records))
    env.run(until=max_simulation_time)
    waiting_times = [r[1] for r in records if r[0] > warmup]
    return sum(waiting_times) / len(waiting_times)

mean_waits = [run_trial(
  s, arrival_rate, service_rate, number_of_servers,
  max_simulation_time, warmup) for s in range(num_trials)]

average_waits = sum(mean_waits) / len(mean_waits)
```

Appendix 3. Ciw model

```
import ciw

max_simulation_time = 800
warmup = 100
num_trials = 20

N = ciw.create_network(
    Arrival_distributions=[['Exponential', 10.0]],
    Service_distributions=[['Exponential', 4.0]],
    Number_of_servers=[3]
)

def run_trial(s, max_simulation_time, warmup):
    """Run one trial of the simulation, returning the average waiting time"""
    ciw.seed(s)
    Q = ciw.Simulation(N)
    Q.simulate_until_max_time(max_simulation_time)
    recs = Q.get_all_records()
    waits = [r.waiting_time for r in recs if r.arrival_date > warmup]
    return sum(waits) / len(waits)

mean_waits = [run_trial(s, max_simulation_time, warmup) for s in range(num_trials)]

average_waits = sum(mean_waits) / len(mean_waits)
```