

BUILDING HEALTHCARE DISCRETE-EVENT SIMULATION MODELS IN FREE AND OPEN SOURCE SOFTWARE: AN INTRODUCTORY TUTORIAL

Thomas Monks

University of Exeter Medical School
University of Exeter
t.m.w.monks@exeter.ac.uk

Alison Harper

University of Exeter Business School
University of Exeter
a.l.harper@exeter.ac.uk

Amy Heather

University of Exeter Medical School
University of Exeter
a.heather2@exeter.ac.uk

Andrew Mayne

Somerset NHS Foundation Trust
Taunton
andrew.mayne@somersetft.nhs.uk

Navonil Mustafee

University of Exeter Business School
University of Exeter
n.mustafee@exeter.ac.uk

ABSTRACT

This tutorial aims to support modellers working in healthcare research or practice to build Discrete-Event Simulation models using the Free and Open Source Software *SimPy* and Python. We provide a step-by-step guide to building a stylised urgent care telephone call centre model. Open materials that accompany the tutorial, including models and exercises, are available online and can be run without installing Python. As the materials are introductory, we also provide a “next steps” section that describes simple, intermediate and advanced extensions to the work.

1 INTRODUCTION

Computer simulation has long been used as a tool for decision support in healthcare process design and optimisation, particularly in supporting healthcare decision makers to ask ‘what-if’ we delivered care differently. Within the academic literature, Discrete-Event Simulation (DES) has become the predominant approach in healthcare modelling (Salleh et al. 2017). Recent reviews of the field demonstrate the wide-ranging applications of DES in health service delivery, for example, evaluating operational performance, improving patient flow, and optimizing service scheduling (Vázquez-Serrano et al. 2021, Forbus and Berleant 2022, Roy et al. 2021). DES models are valuable for decision making and are also valuable research artefacts in and of themselves: models are time-consuming to code and document in natural language; require significant clinical, methods, and informatics expertise; and depend on specialized software and logic. Given the cost that goes into a developing a coded model we argue that Free and Open Source Software (FOSS) and its value to Open Science (Monks et al. 2024) has a substantive role to play in increasing the ability for models to be shared and reused across health services.

This paper provides an introduction to building DES models in FOSS. By the end of the tutorial readers will have the skills to code and extend a simple queuing model in Python and *SimPy*. Before providing a step-by-step coding of a DES model in health, we introduce the concept of FOSS and what this means for computer simulation studies. We then briefly review the availability of FOSS

DES packages; the barriers associated with using them for research compared to modern Commercial Off-The-Shelf (COTS) simulation packages; and the other initiatives available that aim to increase FOSS uptake in research.

2 FREE AND OPEN SOURCE SOFTWARE

In data science, it is typical to use FOSS. For example, a data scientist working in Machine Learning is likely to use one of the popular Python libraries such as *Scikit-Learn*, *TensorFlow* or *PyTorch*. The benefit of FOSS tools like these libraries are that they are openly licensed, and grant users four key freedoms:

0. The freedom to run the program as you wish, for any purpose.
1. The freedom to study how the program works and modify it to suit your needs.
2. The freedom to redistribute copies to help your neighbours.
3. The freedom to distribute copies of your modified versions to others, allowing the whole community to benefit from your changes.

Importantly, FOSS involves more than just open source code; it grants users the right to freely adapt and distribute copies of code. FOSS repositories often include a *permissive* license like MIT or BSD-2 licenses. These grant users the four freedoms (within commercial or non-commercial work), waive liability, and require crediting (citation) of the authors and their software in subsequent work.

FOSS is increasingly used for computer simulation, such as DES (Monks and Harper 2023a). This tutorial will focus on DES in Python, as it is the language we use in our practical health care simulation studies (e.g. Harper et al. (2023)).

2.1 Python simulation tools

Dagkakis and Heavey (2016) reviewed open source software for DES. They identified three FOSS Python packages: *SimPy* (Team SimPy 2020), *PySimulator* (Pfeiffer et al. 2012) and *ScipySim* (McInnes and Thorne 2011). *PySimulator* and *ScipySim* were last updated in 2014 and 2010 respectively. *SimPy* continues to be maintained (last updated November 2023) and has been used in several publications relevant to Operations Research (Bovim et al. 2021, Allen et al. 2020).

An updated list of Python DES packages now includes *Salabim* (van der Ham 2018, MIT licensed, last updated September 2024), *Ciw* (Palmer et al. 2019, MIT licensed; last updated July 2024), and *DE-Sim* (Goldberg and Karr 2020, MIT licensed, last updated November 2020). *Salabim* is a fork of *SimPy* and includes many simulation tools including automatic results collection and animation. *Ciw* (the Welsh word for queue) was developed at Cardiff University and allows users to very quickly build complex multi-class queuing networks. An advanced feature of *Ciw* is network deadlock detection (when the DES model enters a state where there is circular blocking of activities). Palmer and Tian (2021b) provide two reproducible, open source implementations of *Ciw* for hybrid modelling (DES and System Dynamics), archived here (Palmer and Tian 2021a). *DE-Sim* is an object-orientated (OO) framework for developing complex interacting DES models. The *DE-Sim* authors argue that their OO framework is an advancement over *SimPy*'s process-based worldview; although we note that *SimPy* itself is highly flexible and can easily be used within an OO framework; for example see (Allen et al. 2020).

2.2 FOSS: Barriers and initiatives in DES

For modellers used to working with COTS simulation tools, the switch to FOSS and code based DES models can be daunting, costly, and challenging (Monks and Harper 2023a). In other data science disciplines, such as machine learning, there is extensive community support, package documentation, applied examples, and dedicated code repositories available. A critical mass of equivalent materials that lowers the cost of entry to FOSS DES is not yet available; although there are positive signs that support is beginning to build in simulation. One example that supports FOSS for Agent Based Simulation is CoMSES.net¹. The initiative includes a model library linking to a large number of

¹<https://www.comses.net>

ABS models typically developed in NetLogo (and in October 2024, there were 1162 models: 61% NetLogo and 7% Python, etc). Taking a different approach, the Sharing Tools and Artefacts for Reusable Simulations (STARS) framework for healthcare aims to support simulation researchers to share a version of their FOSS computer simulation model that is accessible and reusable by others (Monks, Harper, and Mustafee 2024). STARS provides details of both architecture to share models, and also published applied examples of Python models for others to adapt and reuse. A software-specific initiative to highlight is the extensive and growing documentation to *Ciw*² that now contains over 40 topics of support (as of October 2024). Another software-specific initiative provides a tutorial to use *Streamlit* to build a browser based user-interface to Python models (Monks and Harper 2023b). Lastly, in the UK, the NIHR Applied Research Collaboration South West Peninsula funds the Health Service Modelling Associates Programme³ that has developed the “Little Book of DES” (<https://des.hsma.co.uk/>). The book teaches simulation methods to the NHS using Python and *Simpy*.

3 TUTORIAL OVERVIEW AND SETUP

3.1 Aims

Given the relatively low use of FOSS in healthcare DES relative to COTS (Monks and Harper 2023a) and the documented barriers to entry for new users, we provide introductory tutorial materials to using Python for DES. Our materials are a starting place for modellers looking to use Python in their DES studies. The materials are most suitable for individuals familiar with DES and coding. Although we use Python, the introductory nature of the material should make them accessible to individuals with experience in different coding languages such as R, Julia, Java, C etc. We also aim to make the example simulation code included in our tutorial simple for others to run and not require direct installation on their machine. Given the introductory level of the tutorial, we make several simplifications to the code that would need to be addressed in real studies. We discuss these limitations in the final section and provide more advanced supplementary material illustrating how code could be improved.

3.2 Case study model

This case study uses a simple model of an urgent care telephone call centre, similar to the NHS 111 service in the UK (NHS 2024). In this model, calls to the centre arrive randomly and operators handle them on a first-in-first-out basis. It does not include features like reneging, time dependency, shift work, or referrals and callbacks from medical or nursing practitioners. The simulation time units are in minutes. On average there are 100 new callers per hour (an inter-arrival time of 60.0 / 100 per minute). There are 13 caller operators available. Table 1 lists the statistical distributions and their parameters for the two activities in the model.

Table 1: Case study distributions.

Activity	Distribution
Inter-arrival time of callers	Exponential(mean = 60.0 / 100)
Call triage by operator	Triangular(low = 5.0, mode = 7.0, high = 10.0)

3.3 Simulation software

This tutorial uses *SimPy*: a popular FOSS DES package in Python with a process-based simulation worldview. To build a DES model in *SimPy*, users define Python generator functions (described in Section 5.3) and implement logic to request and release resources. Although *SimPy* provides a full DES engine, it does not contain some of the additional features offered by COTS packages - for example, the user interfaces available with *Arena* or *Simul8*. However, *SimPy* is a lightweight, flexible tool that can be integrated with the rest of the Python data science ecosystem.

It has been used to model a wide variety of health condition pathways and healthcare operations. For example, COVID-19 (Anagnostou et al. 2022, Bovim et al. 2023), renal services (Allen et al. 2020), stroke (Ren et al. 2021, Ren et al. 2020), heart failure (Wise et al. 2021), cancer care (Richardson

²<https://ciw.readthedocs.io>

³<https://sites.google.com/nihr.ac.uk/hsma/hsma-resources>

and Cohn 2018), end-of-life care (Chalk et al. 2021), and operating theatre management (Hassanzadeh et al. 2023, Harper et al. 2023).

3.4 Availability of code and materials

In following sections, we provide step-by-step code to build the case study model, which is explained in detail. Whilst code listings are included in this paper, we strongly recommend using the interactive online materials, if you are interested in building the model. These materials are shared as notebooks using JupyterLite (v0.4) with the Xeus-Python kernel⁴. The notebooks run in a pre-built Python environment in Jupyter-Lab, a popular notebook integrated development environment (IDE). JupyterLite is powered by WebAssembly, that allows readers to run and modify our Python code examples in their browser without having to install Python or *SimPy* locally. Our interactive materials are available at this URL: <https://pythonhealthdatascience.github.io/intro-open-sim/>. Alternatively the Jupyter notebooks can be downloaded directly from our GitHub repository <https://github.com/pythonhealthdatascience/intro-open-sim>. All code was developed in Python 3.11 and *SimPy* 4.1.1 (Team SimPy 2020).

3.5 Structure of tutorial

The tutorial is organised into the following sections:

- **Section 4:** Introduces the basic tools to model variability in Python via the *NumPy* library.
- **Section 5:** Creates a basic *SimPy* model to simulate the arrival process.
- **Section 6:** Adds queuing and service components to the model.
- **Section 7:** Adds basic result collection logic to the model.
- **Section 8:** Discusses where a reader should look next in their studies as well as highlighting some additional resources available in our online supplementary material.

4 MODELLING VARIABILITY IN PYTHON

In our call centre model, there is variation in (a) the time spent waiting for an operator, (b) the duration of the call, and (c) the time interval between successive callers. We model this using standard statistical distributions such as the uniform and exponential distributions. In Python, the *NumPy* packages provide an efficient and user-friendly implementation of many statistical distributions. For example, in Listing 1, we draw one million samples from a **uniform** distribution.

- **Line 1:** Imports the *NumPy* package and aliases it as *np*.
- **Line 3:** Creates a Pseudo-Random Number Generator (PRNG; using the seed 42). This is a standard approach in *NumPy* and makes use of the Permuted Congruential Generator 64-bit (PCG64; period = 2^{128} ; maximum number of streams = 2^{127}).
- **Line 4:** Uses the PRNG object to create one million samples and stores the results in a *NumPy* array called *samples*.

Here we used the **uniform** method of the PRNG object, but we could have **similarly** used the exponential method.

```
1 import numpy as np
2
3 rng = np.random.default_rng(42) # define arrivals PRNG
4 samples = rng.uniform(low=10, high=40, size=1_000_000)
```

Listing 1: Efficient sampling using the *NumPy* package

5 MODELLING AN ARRIVAL PROCESS

In our initial *SimPy* model, we have only included the arrivals of new callers (and excluded all queuing and service, which we add in Section 6). The model code is provided in Listing 2 and explained in Sections 5.1 to 5.4.

⁴<https://xeus-python.readthedocs.io/en/latest>

```

1 import numpy as np
2 import simpy
3
4
5 def arrivals_generator(env):
6     """Model caller arrival process."""
7     arrival_rng = np.random.default_rng()
8
9     while True:
10         inter_arrival_time = arrival_rng.exponential(60.0 / 100.0)
11         yield env.timeout(inter_arrival_time)
12         print(f"Call arrives at: {env.now}")
13
14
15 # model parameters
16 RUN_LENGTH = 100
17
18 # create the simpy environment object
19 env = simpy.Environment()
20
21 # tell simpy that the 'arrivals_generator' is a process to model
22 env.process(arrivals_generator(env))
23
24 # run the simulation model
25 env.run(until=RUN_LENGTH)
26 print(f"end of run. simulation clock time = {env.now}")

```

Listing 2: Modelling caller arrivals

5.1 The environment

SimPy processes execute in an **environment**. In *SimPy* models, the environment is the DES engine. Once an environment is created, users can access its attributes (such as the current simulation time) and methods (for example, creating a new process or running the model). In Listing 2, Line 2 imports the *SimPy* package. Using *SimPy*, Line 19 creates an instance of an environment that is assigned to a variable called `env`.

5.2 Delays

We can include **delays** or **activities** within a *SimPy* process model. For example, these might be the duration of a patient's stay on a ward or the duration of an operation. In this case, we introduce a time between arrival events (inter-arrival time). In Listing 2, an inter-arrival time is sampled on Line 10. This is used on Line 11 to create a delay via the `timeout` method of the environment object `env`. Model logic can access the simulation time before and after delays using `env.now` (e.g. line 12).

5.3 Processes

In *SimPy*, the event-process mechanism is implemented using Python **generators**. A generator is a Python object that can return a sequence of values; for example, a sequence of times between patient arrivals. A simple way to visualise the arrival processes in *SimPy* is as an infinite loop of delays, where each delay represents the time until the next arrival. Unlike a standard Python loop, a generator can be paused and resumed at later time, making it ideal for event-scheduling in *SimPy*.

For each arrival process in our model, we create a new generator process. In Listing 2, we have created a single arrival process (for callers) modelled by the generator function `arrivals_generator` (Lines 5 to 12). The generator function contains an infinite loop (here implemented as a **while** loop). On each iteration, a new inter-arrival time is sampled from the Exponential distribution and an environment timeout (delay) of the same duration is yielded. The keyword **yield** before the timeout is essential in Listing 2 Line 11: it is needed to tell the Python interpreter that the function is a Python generator. On line 11 `yield` is combined with `env.timeout()` to schedule the end of the event (the next caller arrival).

5.4 The setup script

In Listing 2, Lines 16 to 26 setup and run the DES model. Two new concepts are introduced here. Line 22 configures *SimPy* to execute `arrivals_generator` as a **process**. Line 25 run the model until the simulation clock reaches time 100 (i.e. generate arrivals until that time).

6 MODELLING A SERVICE PROCESS

We now expand the model to include a resource-bound service process: speaking with an operator. To do this we will:

1. Add a Python generator called `service` which contains the queuing and call activity logic (Listing 3).
2. Create a *SimPy* resource called `operator` (Listing 4)
3. Modify `arrivals_generator` to create and schedule the `service` process, which uses `operator`, as each caller arrives to the model (Listing 4).

6.1 A service generator

Listing 3 implements a simple *SimPy* process model of calls queuing for service with one of a set of call operator resources. We have implemented the following logic in the `service` process:

1. A caller service process requests a call operator resource. If none are available, the caller process waits in a queue.
2. Once an operator resource becomes available, the call process is assigned and undergoes a phone triage (represented as a delay). The duration of this delay is sampled from a triangular distribution.
3. After the call process is completed, the operator resource is released, and the caller process exits the model.

```

1 def service(identifier, operators, env, service_rng):
2     """Simulates the service process for a call operator"""
3
4     # record the time that call entered the queue
5     start_wait = env.now
6
7     # request an operator
8     with operators.request() as req:
9         yield req
10
11     # record the waiting time for call to be answered
12     waiting_time = env.now - start_wait
13     print(f"operator answered call {identifier} at " + f"{env.now:.3f}")
14
15     # sample call duration.
16     call_duration = service_rng.triangular(left=5.0, mode=7.0, right=10.0)
17
18     # schedule process to begin again after call_duration
19     yield env.timeout(call_duration)
20
21     # print out information for patient.
22     print(
23         f"call {identifier} ended {env.now:.3f}; "
24         + f"waiting time was {waiting_time:.3f}"
25     )

```

Listing 3: Modelling caller queuing and service time

6.1.1 Parameters

The `service` generator accepts four parameters (as in Line 1 of Listing 3):

- `identifier`: a unique caller ID.
- `operators`: the *SimPy* resource representing the call operators.
- `env`: the simulation environment object.
- `service_rng`: the PRNG for service processes. It provides the stream of random numbers used to sample service times.

6.1.2 Requesting, seizing, and releasing a resource

The pattern of seizing and releasing a resource in `service_process` in Listing 3 is typical of how simple queue and service mechanism is implemented in *SimPy* models.

Line 8 uses Python's **with** statement to open a context manager, which manages the request queuing, seizing and releasing of a *SimPy* resource. It calls the **request** method of the operator resource and assigns it to a variable called `req`. The subsequent lines are then indented by four spaces, which tells the Python interpreter that all of the code within the **with** block executes **while holding on to the seized resource**.

Line 9 **yields** the request - this simulates the queuing in the process. The process will pause at Line 9 until an operator resource becomes available (for example, when another call completes). When an operator becomes available Line 12 to 19 are executed. Line 19 simulates the delay to the caller while being triaged by an operator. The *SimPy* simulation engine will pause the process at Line 19. *SimPy* will check if there are any other processes due to complete before this time i.e. new caller arrivals or completion of other calls and resume those first. Once these are complete the service process for this caller will resume.

6.2 Starting a service process after an arrival

Before modifying the arrival process (in Listing 2), it is important to understand the relationship between the arrival and service process in the *SimPy* model. The arrival process spawns multiple instances of the service process - one for each new arrival - and so this can be thought of as **a one-to-many relationship**. To simulate queuing, the multiple service processes share a common resource (the operators). *SimPy* manages all of these processes for you. It pauses and resumes them depending on if operators are available or if a delay is scheduled.

Listing 4 illustrates a modified arrivals process. The following modifications have been made:

1. A *SimPy* resource called `operator` is passed to the `arrivals_generator` (Line 6).
2. A second pseudo-random number stream called `service_rng` is created (Line 12).
3. The arrivals loop is modified to count the number of callers that have arrived (`caller_count`) (Line 15).
4. After a call has arrived, a service process is created and scheduled (Line 21).

```

1 import numpy as np
2 import itertools
3 import simpy
4
5
6 def arrivals_generator(env, operators):
7
8     # create the arrival process rng
9     arrival_rng = np.random.default_rng()
10
11     # create the service rng that we pass to each service process created
12     service_rng = np.random.default_rng()
13
14     # with a counter variable that we can use for unique Ids
15     for caller_count in itertools.count(start=1):

```

```

16     inter_arrival_time = arrival_rng.exponential(60.0 / 100.0)
17     yield env.timeout(inter_arrival_time)
18     print(f"call arrives at: {env.now:.3f}")
19
20     # create a new simpy process for serving this caller.
21     env.process(service(caller_count, operators, env, service_rng))
22
23
24 # model parameters
25 RUN_LENGTH = 100
26 N_OPERATORS = 13
27
28 # create simpy environment and operator resources
29 env = simpy.Environment()
30 operators = simpy.Resource(env, capacity=N_OPERATORS)
31
32 env.process(arrivals_generator(env, operators))
33 env.run(until=RUN_LENGTH)
34 print(f"end of run. simulation clock time = {env.now}")

```

Listing 4: A modified arrival process and run script used to created service processes

6.2.1 A second pseudo-random number stream

For each sampling process in the model, it is good practice to include a unique PRNG. Although it is out of scope for this introductory tutorial, this practice is particularly useful if we later want to implement Common Random Numbers across experiments with the model (as described in Section 8.3.1).

In Listing 4, Line 12 creates a second PRNG `service_rng` to use with service processes. This is created *before* the main arrivals loop as we only need to create one PRNG that is shared across all service processes.

6.2.2 Counting callers

We use the built-in Python library *itertools* (imported on Line 2) to count the number of callers. This is implemented inside the **for** loop on Line 15, with the variable `caller_count` referring to arrival number of the next caller.

6.2.3 Scheduling a service process for each arrival

Line 21 schedules a new service process for each arrival. This is done using the `env.process` method, and its argument is a new instance of a `service` process. The call to create the process includes the current unique caller number (`caller_count`), the shared *SimPy* resource for call operators (`operators`), the simulation environment (`env`), and the PRNG for sampling (`service_rng`). The line is indented within the arrivals loop, ensuring that a new service process is created for every arrival.

6.3 Creating a *SimPy* resource

In Listing 4, Lines 25 to 34 represent a modified script to run the model. In Line 30, we create an instance of *SimPy*'s Resource class called `operators` and set its capacity to 13. This object represents the call operators used across all service processes in the model. In Line 32, we schedule the arrival process and pass the `operators` resource as a parameter.

7 BASIC RESULTS COLLECTION

As *SimPy* models are just Python code, there are many ways to implement results collection. Three possible ways include:

1. **Code an auditor/observer process.** This process will periodically observe the state of the system. We can use this to collect information on the current state at time t . For example,

how many callers are queuing and how many have a call in progress, at certain timepoints throughout the day.

2. **Store process metrics during a run and perform calculations at the end of a run.** For example, if you want to calculate mean patient waiting time then store each caller waiting time in a list and calculate the mean at the end of the run.
3. **Monitor resources to calculate time weighted statistics.** Resource utilisation and mean queue length can be tracked as resources are requested and released. This is more a advanced approach to results collection that requires both knowledge of theory and Object Orientated Programming to implement. The method is beyond the scope of our tutorial, but we provide an example implementation in our online supplementary material.

In this tutorial, we will calculate the mean waiting time for a single replication of the model using method 2. To do this we will make the following modifications to the code:

- Replace all calls to Python's print function with a custom `trace` function that can be used to toggle the display of model output on or off.
- Create a Python **dictionary** called `results` that will store all waiting times for service. For simplicity, we will give this global variable scope.
- Modify `service` to calculate caller waiting times and store these in `results`.
- Create a function called `single_run` that will run the model once and return the mean waiting time as a result.

7.1 Modifying the service process

Listing 5 contains the modifications to the `service` generator for recording wait times:

- **Line 17:** Computes the waiting time (current simulation time minus the arrival time, which we already recorded on Line 10).
- **Line 18:** Stores this time in `results` (the global dictionary) by appending to a list stored under the key `waiting_times`.

Lines 20 and 25 are also modified to call the function `trace`, defined in Lines 1 to 4. This simple function prints output messages depending on whether a constant called `TRACE` is set to true or false.

```
1 def trace(msg):
2     """Wrap print function to toggle simulation printed output on/off"""
3     if TRACE:
4         print(msg)
5
6
7 def service(identifier, operators, env, service_rng):
8     """Simulates the service process for a call operator"""
9     # record the time that call entered the queue
10    start_wait = env.now
11
12    # request an operator
13    with operators.request() as req:
14        yield req
15
16    # record the waiting time for call to be answered
17    waiting_time = env.now - start_wait
18    results["waiting_times"].append(waiting_time)
19
20    trace(f"operator answered call {identifier} at " + f"{env.now:.3f}")
21
22    call_duration = service_rng.triangular(left=5.0, mode=7.0, right=10.0)
23    yield env.timeout(call_duration)
24
25    trace(
26        f"call {identifier} ended {env.now:.3f}; "
```

```

27         + f"waiting time was {waiting_time:.3f}"
28     )

```

Listing 5: Setup of results collection mode

7.2 A run function and global variables

In Listing 6, our original code to create the simulation environment and run the model has been encapsulated into the function `single_run` (Lines 1 to 13). It also has some additional end-of-replication result processing; in this case, finding the mean of all the recorded waiting times.

The advantage of encapsulating a single replication of the model in a function is that we can now easily run multiple independent replications: by creating a loop repeatedly calling `single_run`, and storing the results of each replication as it iterates. In this instance we only run a single replication of the model. We do so by modifying the run script (Lines 17 to 26). We provide example code to run and analyse multiple replications in our supplementary material (see `07_experiments.ipynb`).

We also create the `results` dictionary on Lines 17 and 18, with the key `waiting_times` pointing to an empty Python List. This is the results global variable that is referenced in `service`.

```

1 def single_run(run_length, n_operators):
2     """Perform a single replication of the simulation model and
3     return the mean waiting time as a result"""
4     env = simpy.Environment()
5     operators = simpy.Resource(env, capacity=n_operators)
6
7     env.process(arrivals_generator(env, operators))
8     env.run(until=run_length)
9     print(f"end of run. simulation clock time = {env.now}")
10
11     # Calculate results on notebook level variables.
12     mean_waiting_time = np.mean(results["waiting_times"])
13     return mean_waiting_time
14
15
16 # script to run the model
17 results = {}
18 results["waiting_times"] = []
19
20 # toggle event logging to "off"
21 TRACE = False
22
23 # model parameters
24 RUN_LENGTH = 1000
25 N_OPERATORS = 13
26
27 mean_waiting_time = single_run(RUN_LENGTH, N_OPERATORS)
28 print("Simulation Complete")
29 print(f"Waiting time for call operators: {mean_waiting_time:.2f} minutes")

```

Listing 6: Modifications to collect basic waiting time results

8 DISCUSSION AND NEXT STEPS

This paper provides an introductory tutorial for building healthcare DES models in Python and *SimPy*. It aims to provide modellers familiar with DES and coding the basic materials to begin building their own *SimPy* FOSS models.

The model introduced is only elementary and does not fully reflect the complexities seen in real health systems, or the requirements of rigorous reproducible simulation research. For some aspects of DES modelling, the extensions are trivial and our materials can be adapted. We provide several simple, intermediate and advanced extensions below.

8.1 Simple Extensions

8.1.1 Multiple arrival processes

In some healthcare problems, there will be multiple patient types with differing arrival rates. For example, Griffiths et al. (2010) develop a DES model of a Critical Care Unit. In their model, patients arrive from six sources at differing rates: emergency department, x-ray, emergency surgery, inter-hospital transfers, and elective operations. To model this in *SimPy*, we can create six arrival processes each with their own PRNG and sampling distribution. We would also modify `single_run` to schedule these six generators to run in the *SimPy* DES engine.

8.1.2 Service processes dependent on patient class

Some health systems may share a common patient arrival process, but then have multiple patient classes, each with different treatment processes. A common way to model this problem is to include probabilities that each type of arrival is a different class of patient that follows a specific process. For example, in an urgent care centre, there might be a 10% chance a new patient has trauma injuries and a 90% chance that a patient has a medical emergency. The processes that each class of urgent care patient follows will be very different (for example, trauma patients may need urgent stabilisation/resuscitation and medical patients may just need to follow a less urgent triage and medical consultation process). Our example can be extended by creating two service generator functions encapsulating each process. We would then modify the arrivals generator to (a) sample the class of patient arriving and (b) include conditional logic (a Python `if` statement) to schedule the appropriate service process. We provide an example of an extended version of the call centre model in our online supplementary material (see the `08_full_model.ipynb` notebook).

8.1.3 Introducing a warm-up period

To remove initialisation bias from the output of a non-terminating system, a model should include a warm-up period. At the end of a warm-up period, all result collection processes should be reset. Implementing a warm-up in *SimPy* requires a modeller to i.) partition the run length parameter of the model into durations for warm-up and data collection; and ii.) create a new *SimPy* process that is scheduled to execute when the simulation time equals the warm-up time. We provide an example of implementing a warm-up period in our online supplementary material (see the `08_full_model.ipynb` notebook).

8.2 Intermediate Extensions

8.2.1 Separating parameters from model logic

In our simple tutorial, we passed parameters and simulation objects (e.g. number of operators, simulation environment, PRNGs) as individual arguments to the functions and generators we code. We also limited the number of parameters we needed to pass by hard coding parameters for distributions (e.g. the parameters for the triangular distribution in Listing 3). In a real simulation study, there may be tens or hundreds of parameters (including data files) that a modeller wishes to vary for experimentation, sensitivity analysis or optimisation. It is impractical to pass these individually to each function and generator used in the modelling.

Several options in Python exist for separating parameters from model logic including dictionaries, dataclasses and classes. Our recommendation is to create an `Experiment` class that we pass to the `single_run` (or multiple replications) function. This class should contain default values and a way to quickly set all of the parameters. By using a class, the number of parameters passed to each function within a model is minimal (as low as one: the `Experiment` object). When using this approach, we would also recommend eliminating all global variables (which have only been included in our tutorial for simplicity). Monks and Harper (2023b) provide an advanced tutorial describing this object-oriented approach using classes, and apply it to both *SimPy* and *Ciw* models. We provide an applied example using an `Experiment` class in our supplementary online material (see the `07_experiments.ipynb` notebook).

8.2.2 Time dependent arrival rates

It is common to encounter time dependency in the arrival of patients to healthcare systems; for example, following a Non-Stationary Poisson Process (NSPP). Unlike COTS packages, tools like *SimPy* do not include built-in algorithms for sampling from a NSPP such as thinning or from a piece-wise linear function. A modeller is therefore required to implement these algorithms themselves. To help, we direct users to applied example 2 in Monks, Harper, and Mustafee (2024), which includes an implementation of NSPP via thinning.

8.3 Advanced Extensions

8.3.1 Common Random Numbers

Common Random Numbers (CRN) - also known as streaming - is a variance reduction approach: a method to reduce the variation (sampling noise) between experiments conducted with a simulation model. We note that CRN are not essential for simulation research but, on a practical level, CRN reduce both the number of replications that need to be run to compare experiments, and help support repeatable results for each replication of a model. In this tutorial, we have taken some steps to implement CRN by introducing a PRNG for each sampling process included (arrivals and service). To fully implement CRN, a method is needed to create non-overlapping streams of PRNs for each replication. In Python, *NumPy*'s PRNG tools can be used to spawn non-overlapping streams for each replication with only a small amount of code. For an example implementation of this approach, we again refer readers to Applied Example 2 in Monks, Harper, and Mustafee (2024). We also provide an applied example using CRN in our supplementary online material (see the `07_experiments.ipynb` notebook).

8.3.2 Reproducible pipelines and experiments

Reproducible analytical pipelines (RAPs), in the context of model experimentation, are automated processes for the parameterisation, execution, output analysis, and reporting of simulation models. For example, imagine a journal article that reports the results of two experiments with our urgent care call centre. Each experiment produces a formatted table of performance measures. A RAP here would reproduce the exact figures reported in the table (including potentially LaTeX formatting). The RAP would parameterise the model, run it in an identical software environment, use the same random number streams, analyse the results, and report tables identical to those found in the journal article. Code-based models such as those built in *SimPy* lend themselves to a RAP approach (which is good science), but require more time to develop as code will need to be structured appropriately.

9 SUMMARY

Our aim in this tutorial is to support modellers in their journey to build and share FOSS DES models. When compared to DES models built using COTS packages, FOSS models are still relatively rare. We hope this tutorial will be part of the solution to increase the FOSS DES prevalence in the literature. This is because we feel that - in order for the modelling and simulation community to learn and adopt the skills needed to build FOSS models - we need to build a critical mass of open models. As discussed, the tutorial we provide is only introductory, but it is relatively simple to extend our work to health systems with more detailed processes and patient pathways. We hope that the intermediate and advanced next steps we outline also open readers to new ideas, such as RAPs, and an understanding of how their DES models work and could be organised.

ACKNOWLEDGMENTS

TM is supported by the NIHR Applied Research Collaboration South West Peninsula. The views expressed in this publication are those of the author(s) and not necessarily those of the NIHR or the Department of Health and Social Care.

FUNDING


This work was supported by the Medical Research Council [MR/Z503915/1].


REFERENCES


- Allen, M., A. Bhanji, J. Willemsen, S. Dudfield, S. Logan, and T. Monks. 2020. "A simulation modelling toolkit for organising outpatient dialysis services during the COVID-19 pandemic". *PLoS One* 15 (8): e0237628.
- Anagnostou, A., D. Groen, S. J. Taylor, D. Suleimenova, N. Abubakar, A. Saha, K. Mintram, M. Ghorbani, H. Daroge, T. Islam, Y. Xue, E. Okine, and N. Anokye. 2022. "FACS-CHARM: A Hybrid Agent-Based and Discrete-Event Simulation Approach for Covid-19 Management at Regional Level". In *2022 Winter Simulation Conference (WSC)*, 1223–1234.
- Bovim, T. R., A. N. Gullhav, H. Andersson, J. Dale, and K. Karlsen. 2021, December. "Simulating emergency patient flow during the COVID-19 pandemic". *Journal of Simulation* 0 (0): 1–15. Publisher: Taylor & Francis _eprint: <https://doi.org/10.1080/17477778.2021.2015259>.
- Bovim, T. R., A. N. Gullhav, H. Andersson, J. Dale, and K. Karlsen. 2023. "Simulating emergency patient flow during the COVID-19 pandemic". *Journal of Simulation* 17 (4): 407–421.
- Chalk, D., S. Robbins, R. Kandasamy, K. Rush, A. Aggarwal, R. Sullivan, and C. Chamberlain. 2021. "Modelling palliative and end-of-life resource requirements during COVID-19: implications for quality care". *BMJ open* 11 (5): e043795.
- Dagkakis, G., and C. Heavey. 2016. "A review of open source discrete event simulation software for operations research". *Journal of Simulation* 10 (3): 193–206.
- Forbus, J. J., and D. Berleant. 2022. "Discrete-Event Simulation in Healthcare Settings: A Review". *Modelling* 3 (4): 417–433.
- Goldberg, A. P., and J. R. Karr. 2020. "DE-Sim: an object-oriented, discrete-event simulation tool for data-intensive modeling of complex systems in Python". *Journal of Open Source Software* 5 (55): 2685.
- Griffiths, J. D., M. Jones, M. Read, and J. E. Williams. 2010. "A simulation model of bed-occupancy in a critical care unit". *Journal of Simulation* 4 (1): 52–59.
- Harper, A., T. Monks, R. Wilson, M. T. Redaniel, E. Eyles, T. Jones, C. Penfold, A. Elliott, T. Keen, M. Pitt et al. 2023. "Development and application of simulation modelling for orthopaedic elective resource planning in England". *BMJ open* 13 (12): e076221.
- Hassanzadeh, H., J. Boyle, S. Khanna, B. Biki, F. Syed, L. Sweeney, and E. Borkwood. 2023. "A discrete event simulation for improving operating theatre efficiency". *The International Journal of Health Planning and Management* 38 (2): 360–379.
- McInnes, A. I., and B. R. Thorne. 2011. "ScipySim: Towards Distributed Heterogeneous System Simulation for the SciPy Platform".
- Monks, T., and A. Harper. 2023a. "Computer model and code sharing practices in healthcare discrete-event simulation: a systematic scoping review". *Journal of Simulation* 0 (0): 1–16. Publisher: Taylor & Francis _eprint: <https://doi.org/10.1080/17477778.2023.2260772>.
- Monks, T., and A. Harper. 2023b. "Improving the usability of open health service delivery simulation models using Python and web apps". *NIHR Open Research* 3:48.
- Monks, T., A. Harper, and N. Mustafee. 2024. "Towards sharing tools and artefacts for reusable simulations in healthcare". *Journal of Simulation* 0 (0): 1–20.
- NHS 2024. "NHS 111 Online Service". <https://111.nhs.uk/>. Accessed: 2024-10-10.
- Palmer, Geraint and Tian, Yawen 2021a, March. "Source code for Ciw hybrid simulations."
- Palmer, G. I. et al. 2019. "Ciw: An open-source discrete event simulation library". *Journal of Simulation* 13 (1): 68–82.
- Palmer, G. I., and Y. Tian. 2021b. "Implementing hybrid simulations that integrate DES+ SD in Python". *Journal of Simulation*:1–17.
- Pfeiffer, A. et al. 2012. "PySimulator—A simulation and analysis environment in Python with plugin infrastructure".
- Ren, Y., M. Phan, P. Luong, J. Wu, D. Shell, C. D. Barras, S. Chrysosidis, H. K. Kok, M. Burney, B. Tahayori, J. Maingard, A. Jhamb, V. Thijs, D. M. Brooks, and H. Asadi. 2021. "Application of a computational model in simulating an endovascular clot retrieval service system within regional Australia". *Journal of Medical Imaging and Radiation Oncology* 65 (7): 850–857.
- Ren, Y., M. Phan, P. Luong, J. Wu, D. Shell, C. D. Barras, H. K. Kok, M. Burney, B. Tahayori, H. M. Seah, J. Maingard, K. Zhou, A. Lamanna, A. Jhamb, V. Thijs, D. M. Brooks, and H. Asadi. 2020.


- “Geographic Service Delivery for Endovascular Clot Retrieval: Using Discrete Event Simulation to Optimize Resources”. *World Neurosurgery* 141:e400–e413.
- Richardson, D. B., and A. E. M. Cohn. 2018. “MODELING THE IMPACT OF MAKE-AHEAD CHEMOTHERAPY DRUG POLICIES THROUGH DISCRETE-EVENT SIMULATION”. In *2018 Winter Simulation Conference (WSC)*, 2690–2700.
- Roy, S., S. P. Venkatesan, and M. Goh. 2021. “Healthcare services: A systematic review of patient-centric logistics issues using simulation”. *Journal of the Operational Research Society* 72 (10): 2342–2364.
- Salleh, S., P. Thokala, A. Brennan, R. Hughes, and A. Booth. 2017. “Simulation Modelling in Healthcare: An Umbrella Review of Systematic Literature Reviews”. *PharmacoEconomics* 35 (9): 937–949.
- Team SimPy 2020. “SimPy 3.0.11”. <https://simpy.readthedocs.io/en/latest/index.html>.
- van der Ham, R. 2018. “Salabim: open source discrete event simulation and animation in python”. In *Proceedings of the 2018 Winter Simulation Conference*, 4186–4187.
- Vázquez-Serrano, J. I., R. E. Peimbert-García, and L. E. Cárdenas-Barrón. 2021. “Discrete-Event Simulation Modeling in Healthcare: A Comprehensive Review”. *International Journal of Environmental Research and Public Health* 18 (22): 12262.
- Wise, A. F., L. E. Morgan, A. Heib, C. S. Currie, A. Champneys, R. Nadarajah, C. Gale, and M. Mamas. 2021. “Modeling Of Waiting Lists For Chronic Heart Failure In The Wake Of The COVID-19 Pandemic”. In *2021 Winter Simulation Conference (WSC)*, 1–11.

AUTHOR BIOGRAPHIES

THOMAS MONKS is an Associate Professor of Health Data Science at University of Exeter Medical School. His research interests include open science for computer simulation, urgent and emergency care, and real-time discrete-event simulation. His email address is t.m.w.monks@exeter.ac.uk 

ALISON HARPER is a lecturer in operations and analytics at the Centre for Simulation, Analytics and Modelling, University of Exeter. Her research interests include applied health and social care modelling and simulation, real-time simulation, and reusable modelling in healthcare. Her email address is a.l.harper@exeter.ac.uk 

AMY HEATHER is a Postdoctoral Research Associate the Peninsula Collaboration for Health Operational Research and Data Science (PenCHORD) at the University of Exeter. Her research interests include health data science and open science. Her email address is a.heather2@exeter.ac.uk 

ANDREW MAYNE is Chief Scientist for Data, Operational Research and Artificial Intelligence at Somerset NHS Foundation Trust. He is also Chief Technology Officer for the South West Secure Data Environment. His interests are in admission prediction and simulation modelling of patient pathways in the NHS. His email address is andrew.mayne@somersetft.nhs.uk 

NAVONIL MUSTAFEE is Professor of Analytics and Operations Management at the University of Exeter Business School, UK. His research focuses on modelling and simulation methodologies, including hybrid modelling and real-time simulation, and their application in healthcare, supply chain management, circular economy and climate change adaptation and resilience. He is a Joint Editor-in-Chief of the *Journal of Simulation* (UK OR Society journal). His email address is n.mustafee@exeter.ac.uk 