Chapter 4

Simulation Programming with Python

This chapter shows how simulations of some of the examples in Chap. 3 can be programmed using Python and the SimPy simulation library[1]. The goals of the chapter are to introduce SimPy, and to hint at the experiment design and analysis issues that will be covered in later chapters. While this chapter will generally follow the flow of Chap. 3, it will use the conventions and patterns enabled by the SimPy library.

4.1 SimPy Overview

SimPy is an object-oriented, process-based discrete-event simulation library for Python. It is open source and released under the M license. SimPy provides the modeler with components of a simulation model including processes, for active components like customers, messages, and vehicles, and resources, for passive components that form limited capacity congestion points like servers, checkout counters, and tunnels. It also provides monitor variables to aid in gathering statistics. Random variates are provided by the standard Python random module. Since SimPy itself is written in pure Python, it can also run on the Java Virtual Machine (Jython) and the .NET environment (IronPython). As of SimPy version 2.3, it works on all implementations of Python version 2.6 and up. Other documentation can be found at the SimPy website[1] and other SimPy tutorials[2].

SimPy and the Python language are each currently in flux, with users of Python in the midst of a long transition from the Python 2.x series to Python 3.x while SimPy is expected to transition to version 3 which will involve changes in the library interface. Scientific and technical computing users such as most simulation modelers and analysts are generally staying with the Python 2.x se-

[1] https://simpy.readthedocs.org/en/2.3.1/

ries as necessary software libraries are being ported and tested. SimPy itself supports the Python 3.x series as of version 2.3. In addition, SimPy is undergoing a major overhaul from SimPy 2.3 to version 3.0. This chapter and the code on the website will assume use of Python 2.7.x and SimPy 2.3. In the future, the book website will also include versions of the code based on SimPy 3.0[2] and Python 3.2 as they and their supporting libraries are developed.

SimPy comes with data collection capabilities. But for other data analysis tasks such as statistics and plotting it is intended to be used along with other libraries that make up the Python scientific computing ecosystem centered on Numpy and Scipy[3]. As such, it can easily be used with other Python packages for plotting (Matplotlib), GUI (WxPython, TKInter, PyQT, etc.), statistics (scipy.stats, statsmodels), and databases. This chapter will assume that you have Numpy, Scipy[3], Matplotlib[4], and Statsmodels[5] installed. In addition the network/graph library Networkx will be used in a network model, but it can be skipped with no loss of continuity. The easiest way to install Python along with its scientific libraries (including SimPy) is to install a scientifically oriented distribution, such as Enthought Canopy[6] for Windows, Mac OS X, or Linux; or Python (x,y)[7] for Windows or Linux. If you are installing using a standard Python distribution, you can install SimPy by using `easy_install` or `pip`. Note the capitalization of 'SimPy' throughout.

```
easy_install install SimPy
```

or

```
pip install SimPy
```

The other required libraries can be installed in a similar manner. See the specific library webpages for more information.

This chapter will assume that you have the Numpy, Scipy, Matplotlib, and Statsmodels libraries installed.

4.1.1 Random numbers

There are two groups of random-variate generations functions generally used, `random` from the Python Standard Library and the random variate generators in the `scipy.stats` model. A third source of random variate generators are those included in PyGSL, the Python interface to the GNU Scientific Library (`http://pygsl.sourceforge.net`)

The `random` module of the Python Standard Library is based on the Mersenne Twister as the core generator. This generator is extensively tested and produces

[2] https://simpy.readthedocs.org/en/latest/index.html
[3] http://www.scipy.org
[4] http://matplotlib.org/
[5] http://statsmodels.sourceforge.net/
[6] http://www.enthought.com/products/canopy/
[7] http://code.google.com/p/pythonxy/

53-bit precision floats and a period of $2^{19937} - 1$. Some distributions included in the the `random` module include uniform, triangular, Beta, Exponential, Gamma, Gaussian, Normal, Lognormal, and Weibull distributions.

The basic use of random variate generators in the `random` module is as follows:

1. Load the `random` module: `import random`
2. Instantiate a generator: `g = random.Random()`
3. Set the seed: `g.seed(1234)`
4. Draw a random variate:
   - A random value from 0 to 1: `g.random()`
   - A random value (float) from a to b: `g.uniform(a,b)`
   - A random integer from a to b (inclusive): `g.randint(a, b)`
   - A random sample of k items from list *population*: `g.sample(population, k)`

The `Scipy` module includes a larger list of random variate generators including over 80 continuous and 10 discrete random variable distributions. For each distribution, a number of functions are available:

- `rvs`: Random Variates generator,
- `pdf`: Probability Density Function,
- `cdf`: Cumulative Distribution Function,
- `sf`: Survival Function (1-CDF),
- `ppf`: Percent Point Function (Inverse of CDF),
- `isf`: Inverse Survival Function (Inverse of SF),
- `stats`: Return mean, variance, (Fisher's) skew, or (Fisher's) kurtosis,
- `moment`: non-central moments of the distribution.

As over 80 distributions are included, the parameters of the distributions are in a standardized form, with the parameters being the location, scale and shape of the distribution. The module documentation and a probability reference may be consulted to relate the parameters to those that you may be more familiar with.

The basic use of Scipy random number generators is as follows.

1. Load the Numpy and Scipy modules

```
import numpy as np
import scipy as sp
```

2. Set the random number seed. Scipy uses the Numpy random number generators so the Numpy seed function should be used: `np.random.seed(1234)`

3. Instantiate the generator[8]. Some examples:

   - Normal with mean 10 and standard deviation 4:
     `norm1 = sp.stats.norm(loc = 10, scale = 4)`
   - Uniform from 0 to 10:
     `unif1 = sp.stats.uniform(loc = 0, scale = 10)`
   - Exponential with mean 1:
     `expo1 = sp.stats.expon(scale = 1.0)`

4. Generate a random value: `norm1.rvs()`.

As you already know, the pseudorandom numbers we use to generate random variates in simulations are essentially a long list. Random-number streams are just different starting places in this list, spaced far apart. The need for streams is discussed in Chap. 7,but it is important to know that any stream is as good as any other stream in a well-tested generator. In Python, the random number stream used is set using the seed functions (`random.seed()` or `numpy.seed` as applicable).

4.1.2 SymPy components

SimPy is built upon a special type of Python function called **generators** [**?**]. When a generator is called, the body of the function does not execute, rather, it returns an iterator object that wraps the body, local variables, and current point of execution (initially the start of the function). This iterator then runs the function body up to the `yield` statement, then returns the result of the following expression.

Within SimPy, the `yield` statements are used to define the event scheduling. This is used for a process to either do something to itself (e.g. the next car arrives); to request a resource, such as requesting a server; to release a resource, such as a server that is no longer needed; or to wait for another event. [2].

- `yield` hold: used to indicate the passage of a certain amount of time within a process;

- `yield` request: used to cause a process to join a queue for a given resource (and start using it immediately if no other jobs are waiting for the resource);

[8] this method is known in the documentation as *freezing a distribution.*

- `yield` release: used to indicate that the process is done using the given resource, thus enabling the next thread in the queue, if any, to use the resource;

- `yield` passivate: used to have a process wait until "awakened" by some other process.

A **Process** is based on a sequence of these `yield` generators along with simulation logic.

In a SimPy simulation, the simulation is initialized, then resources are defined. Next, processes are defined to generate entities, which in turn can generate their own processes over the course of the simulation. The processes are then activated so that they generate other events and processes. Monitors collect statistics on entities, resources, or any other event which occurs in the model.

In SimPy, resources such as parking spaces are represented by **Resource** . There are three types of resources.

- A **Resource** is something whose units are required by a **Process**. When a **Process** is done with a unit of the **Resource** it releases the unit for another **Process** to use. A **Process** that requires a unit of Resource when all units are busy with other Processes can join a queue and wait for the next unit to be available.

- A **Level** is homogeneous undifferentiated material that can be produced or consumed by a **Process**. An example of a level is gasoline in a tank at a gas station.

- A **Store** models the production or consumption of specific items of any type. An example would be a storeroom that holds a variety of surgical supplies,

The simulation must also collect data for use in later calculating statistics on the performance of the system. In SimPy, this is done through creating a **Monitor**.

Collecting data within a simulation is done through a **Monitor** or a **Tally**. As the **Monitor** is the more general version, this chapter will use that. You can go to the SimPy website for more information. The **Monitor** makes observations and records the value and the time of the observation, allowing for statistics to be saved for analysis after the end of the simulation. The **Monitor** can also be included in the definition of any **Resource** (to be discussed in the main simulation program) so statistics on queues and other resources can be collected as well. In the `Car` object; the **Monitor** `parking` tracks the value of the variable `self.sim.parkedcars` at every point where the value of `self.sim.parkedcars` changes, as cars enter and leave the parking lot.

In the main part of the simulation program, the simulation object is defined, then resources, processes, and monitors are defined and activated as needed to start the simulation. In addition, the following function calls can be made:

```
import random, math
import numpy as np
import scipy as sp
import scipy.stats as stats
import matplotlib.pyplot as plt
import SimPy.Simulation as Sim
```

Figure 4.1: Library imports.

- initialize(): set the simulation time and the event list
- activate(): used to mark a thread (process) as runnable when it is first created and add its first event into the event list.
- simulate(): starts the simulation
- reactivate(): reactive a previously passivated process
- cancel(): cancels all events associated with a previously passivated process.

The functions `activate()`, `reactivate()`, and `cancel()` can also be called within processes as needed.

4.2 Simulating the $M(t)/M/\infty$ Queue

Here we consider the parking lot example of Sect. 3.1a queueing system with time-varying car arrival rate, exponentially distributed parking time and an infinite number of parking spaces.

A SimPy simulation generally consists of import of modules, defining processes and resources, defining parameters, a main program declaring, and finally reporting.

At the top of the program are imports of modules used. (Fig. 4.1) By convention, modules that are part of the Python Standard Library such as `math` and `random` are listed first. Next are libraries considered as foundational to numerical program such as `numpy`, `scipy`, and `matplotlib`. Note that we use the `import libraryname as lib` convention. This short form allows us to make our code more readable as we can use `lib` to refer to the module in the code instead of having to spell out `libraryname`. Last, we import in other modules and modules we may have written ourselves. While you could also import directly into the global namespace using the construct `from SimPy.Simulation import *`, instead of `import SimPy.Simulation as Sim`, this is discouraged as there is a potential of naming conflicts if two modules happened to include functions or classes with the same name.

Next are two components, the cars and the source of the cars. (Fig. 4.2) Both of these are classes that subclass from **Sim.Process**. The class `Arrival` generates the arriving cars using the `generate` method. The mean arrival rate is a function based on the time during the day. Then the actual interarrival time is drawn from an exponential distribution.

```
class Arrival(Sim.Process):
    """ Source generates cars at random

        Arrivals are at a time-dependent rate
    """

    def generate(self):
        i=0
        while (self.sim.now() < G.maxTime):
            tnow = self.sim.now()
            arrivalrate =  100 + 10 * math.sin(math.pi * tnow/12.0)
            t = random.expovariate(arrivalrate)
            yield Sim.hold, self, t
            c = Car(name="Car%02d" % (i), sim=self.sim)
            timeParking = random.expovariate(1.0/G.parkingtime)
            self.sim.activate(c, c.visit(timeParking))
            i += 1

class Car(Sim.Process):
    """ Cars arrives, parks for a while, and leaves

        Maintain a count of the number of parked cars as cars arrive and leave
    """

    def visit(self, timeParking=0):
        self.sim.parkedcars += 1
        self.sim.parking.observe(self.sim.parkedcars)
        yield Sim.hold, self, timeParking
        self.sim.parkedcars -= 1
        self.sim.parking.observe(self.sim.parkedcars)
```

Figure 4.2: Model components.

Within the `Car` object, the `yield Sim.hold, self, timeParking` line is used to represent the car remaining in the parking lot. To look at this line more closely

1. `yield`: a Python keyword that returns a generator. In this case, it calls the function that follows in a computationally efficient manner.

2. `Sim.hold`: The `hold` function within `Simpy.Simulation`. This causes a process to wait.

3. `self`: A reference to the current object (Car). The first parameter passed to the `Sim.hold` command. In this case it means the currently created Car should wait for a period of time. If the `yield hold` line was preceded by a `yield request`, the resource required by `yield request` would be included in the `yield hold`, i.e. both the current process and any resources the process is using would wait for the specified period of time.

4. `timeParking`: The time that the Car should wait. This is the second parameter passed to the `Sim.hold` command.

Typically, `yield, request, hold, release` are used in sequence. For example, if the number of parking spaces were limited, the act of a car taking a parking space for a period of time would be represented as follows within the Car.visit() function.

```
yield Sim.request, self, parkingspace
yield Sim.hold, self, timeParking
yield Sim.release, self, parkingspace
```

In this case, since the purpose of the simulation is to determine demand for parking spaces, we assume there are unlimited spaces and we count the number of cars in the parking lot at any point in time.

In SimPy, resources such as parking spaces are represented by Resources. There are three types of resources.

- A **Resource** is something whose units are required by a **Process**. When a **Process** is done with a unit of the **Resource** it releases the unit for another **Process** to use. A **Process** that requires a unit of Resource when all units are busy with other Processes can join a queue and wait for the next unit to be available.

- A **Level** is homogeneous undifferentiated material that can be produced or consumed by a **Process**. An example of a level is gasoline in a tank at a gas station.

- A **Store** models the production or consumption of specific items of any type. An example would be a storeroom that holds a variety of surgical supplies,

```
class G:
    maxTime = 24.0 # hours
    arrivalrate = 100 # per hour
    parkingtime = 2.0 # hours
    parkedcars = 0
    seedVal = 9999
```

Figure 4.3: Global declarations.

Global declarations are in Fig. 4.3. While Python allows these to be in the global namespace, it is preferred that these be put into a class created for the purpose as shown here. In particular, this is generally used as a central location for parameters of the model.

The main simulation class is shown in Fig. 4.4. The simulation class `Parkingsim` is a subclass of **Sim.Simulation**. While it can have other functions, the central function is the `run(self)` function. Note that the first argument of a member function of a class is `self`, indicating that the function has access to the other functions and variables of the class. The `run` function takes one argument, the random number `seed`. `Sim.initialize()` sets all Monitors and the simulation clock. Then processes and resources are created. In this simulation there is one process, the generation of arriving cars in `Arrival`. Note that at this point, there are no `Car`'s, as the `Arrival` process will create the cars. This simulation does not have any resources; but if, for example, the parking lot had a limited capacity of 20 spaces, it could have been created here by:

```
self.parkinglot = Sim.Resource(capacity = 20, name='Parking lot',
unitName='Space', monitored=True, sim=self)
```

This creates the parking lot with 20 spaces. By default, resources are a FIFO, non-priority queue with no preemption. In addition to the capacity of the Resource (number of servers), there are two queues (lists) associated with the resource. First is the `activeQ`, which is the list of process objects currently using the resource. Second is the `waitQ`, which is the number of processes that have requested but have not yet received a unit of the resource. If when creating the resource the option `monitored=True` is set, then a **Monitor** is created for each queue, and statistics can be collected on resource use and the wait queue for further analysis. The last option is `sim=self`, which declares that the Process or Resource is part of the simulation defined in the current class derived from **Sim.Simulation**. If the simulation was defined in the global namespace (not inside of a **class**), then this option would not be needed.

After Processes and Resources are created, any additional Monitors are created here. Note that Processes, Resources, and Monitors are created using `self.` constructs. This classifies the object as part of the class, and not only local to the `run()` function. Other classes and methods that are part of this simulation (declared using `sim = self`) can refer to any object created here using `self.sim.objectname`. This includes Monitors as well as variables that need to be updated from anywhere in the simulation. For example, the variable `self.parkedcars` can be updated from the `Car` class in Fig. 4.2 using

```
class Parkingsim(Sim.Simulation):
    def run(self, aseed):
        random.seed(seed)
        Sim.initialize()
        s = Arrival(name='Arrivals', sim=self)
        self.parking = Sim.Monitor(name='Parking', ylab='cars',
                                   tlab='time', sim=self)
        self.activate(s, s.generate(), at=0.0)
        self.simulate(until=G.maxTime)

parkinglot = Parkingsim()
parkinglot.run(1234)
```

Figure 4.4: Main Simulation.

```
plt.figure(figsize=(5.5,4))
plt.plot(parkinglot.parking.tseries(),parkinglot.parking.yseries())
plt.xlabel('Time')
plt.ylabel('Number of cars')
plt.xlim(0, 24)
```

Figure 4.5: Plotting results.

`self.sim.parkedcars`.

Following the main simulation class, the class is instantiated using `parkinglot = Parkingsim()`, then the simulation is run using `parkinglot.run(seedvalue)`. One advantage of using a simulation class and object is that the monitors created in the simulation are available to the object instance `parkinglot` after the simulation is run. If the simulation is run from within the Python command line or the IPython notebook[9], this allows for interactive exploration of the simulation results. For example, the monitor `parking` monitors the number of cars that are in the parking lot by time. So a graph can be plotted that tracks the number of cars over the course of the 24 hour day. (Fig. 4.5) The steps are:

1. Load matplot lib (Fig. 4.1) `import matplotlib.pyplot as plt`
2. Set figure size `plt.figure(figsize=(5.5,4))`;
3. Define the plot. Note the use of the attributes of the monitor `parking`.
   `plt.plot(parkinglot.parking.tseries(), parkinglot.parking.yseries())`
4. Set labels and other options

This results in the plot in Fig. 4.6.

To make inferences or conclusions based on the simulation, it is necessary to run the simulation many replications. So to find the daily average of the number of cars, we can run the simulation over 1000 days and look at the average and maximum parking over that period. Fig. 4.7 shows how to do this using a `for`

[9] http://ipython.org

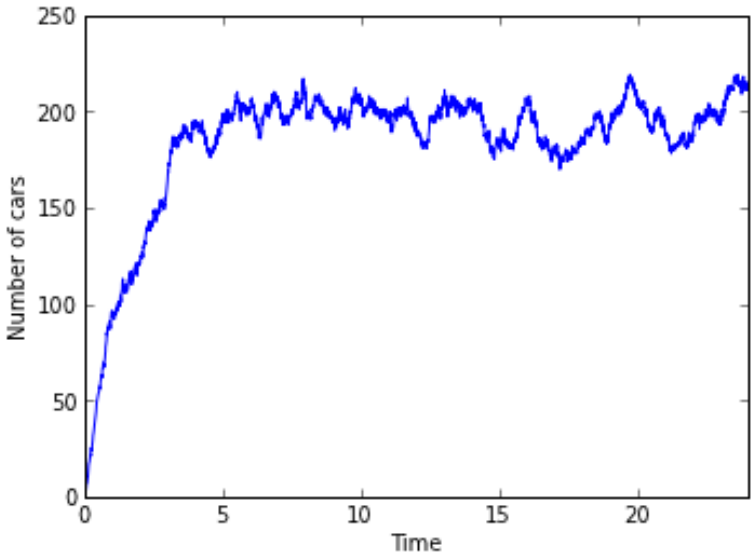

Figure 4.6: Number of parked cars over time.

```
initialseed = 4321
parkingdemand = []
daysrep = 1000
for i in range(daysrep):
    parkinglot.initialize()
    parkinglot.run(initialseed + i)
    parkingdemand.append(parkinglot.parking)
```

Figure 4.7: 1000 replications

loop.[10] For each replication, initialize the `parkinglot` instance of the simulation object, then `run` using a new random number seed. Then, `append` the simulation results to a list object that was initialized using `parkingdemand = []`. Note that any object can be appended to a list. In this case the monitor `parking` for each simulation run was saved to the list `parkingdemand`.

From this list of simulation monitor results, one can then extract a particular data element, then calculate statistics from it. So, for each simulation monitor saved to `parkingdemand`, the time average number of cars in the lot is given by `parkingdemand[i].timeAverage()`. So, to have a list of the time average number of cars for each replication, we can use the list comprehension

`[parkingdemand[i].timeAverage() for i in range(daysrep)]`.

The construct `[function(i) for i in range(datasetlength)]` is known as a list comprehension. It is a compact method for stating:

[10]The Python scientific libraries include provisions for working with multi-core processors that this chapter will not utilize.

```
averagedailyparking = [parkingdemand[i].timeAverage() for
                i in range(daysrep)]
plt.hist(averagedailyparking, bins=25, cumulative = True,
                label = 'Average number of cars during day')
plt.xlabel('Average number of cars in day')
plt.ylabel('Days (cumulative)')
```

Figure 4.8: 1000 replications

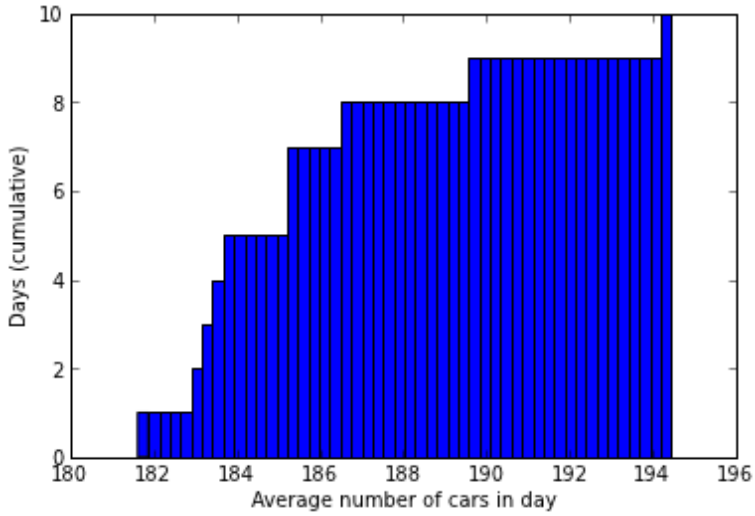

Figure 4.9: Cumulative histogram of average number of parked cars.

```
for i in range(datasetlength):
function(i)
```

In addition to being more compact, this construct is used often in data analysis because it can be relatively easily parallelized if the computing facilities are available. The resulting list can then be analyzed through visualization or statistics. In Fig. 4.9 there is the resulting cumulative histogram.

Fig. 4.10 shows a standard histogram of the maximum number of cars parked in a given simulation day, indicated by the option `cumulative=False`. The corresponding plot is in Fig. 4.11.

```
maxparkingdailyparking = [max(parkingdemand[i].yseries())
                for i in range(daysrep)]
plt.hist(maxparkingdailyparking, bins=25, cumulative = False)
plt.xlabel('Maximum number of cars')
plt.ylabel('Days (cumulative)')
```

Figure 4.10: Maximum number of cars parked in a day.

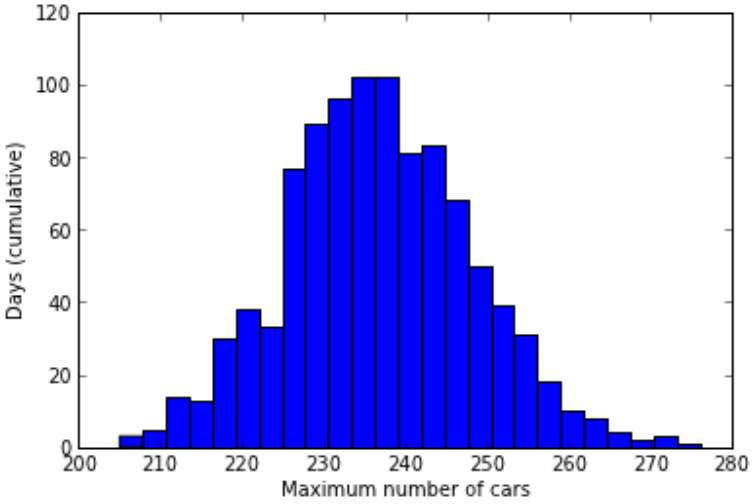

Figure 4.11: Histogram of maximum number of parked cars.

```
testvalue, pvalue = sp.stats.normaltest(averagedailyparking)
print(pvalue)
```

```
0.761955484599
```

Figure 4.12: Test for normality for average cars parked.

We think that the distribution of the average number of cars in the lot on any given day follows a normal distribution. We can test this using the the `sp.stats.normaltest` function. (Figure 4.12) In addition, we can plot a normal probability plot using the `sp.stats.statsplot` function. Figure 4.13 shows a normal probability plot that is consistant with a hypothesis that the average number of cars in a day follows a normal distribution.

4.2.1 Issues and Extensions

1. The $M(t)/M/\infty$ simulation presented here simulates 24 hours of parking lot operation, and treats each 24-hour period as as independent replication starting with an empty garage. This only makes sense if the garage is emptied each day, for instance if the mall closes at night. Is the assumed arrival rate $\lambda(t)$ appropriate for a mall that closes at night?

2. Suppose that the parking lot serves a facility that is actually in operation 24 hours a day, seven days per week (that is, all the time). How should the simulation be initialized, and how long should the run length be in this case?

3. How could the simulation be initialized so that there are 100 cars in the

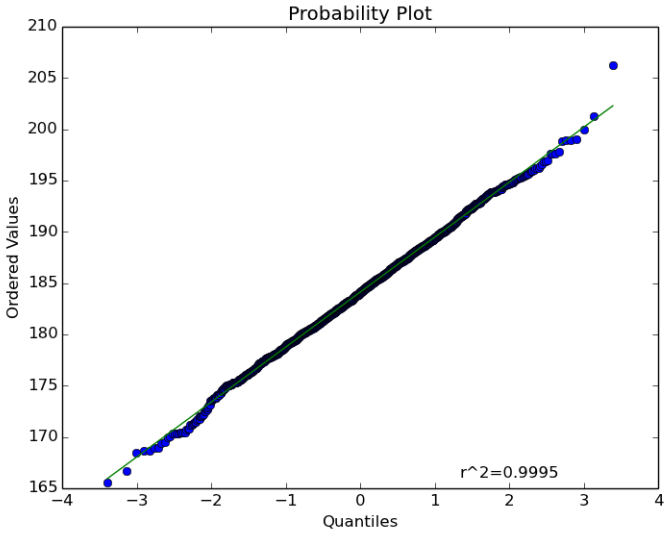

Figure 4.13: Normal probability plot of average cars parked in a day generated by `scipy.stats.statsplot()`.

parking lot at time 0?

4. When this example was introduced in Sect. 3.1 , it was suggested that we size the garage based on the (Poisson) distribution of the number of cars in the garage at the point in time when the mean number in the garage was maximum. Is that what we did, empirically, here? If not, how is the quantity we estimated by simulation related to the suggestion in Sect. 3.1(for instance, will the simulation tend to suggest a bigger or smaller garage than the analytical solution in Sect. 3.1

5. One reason that this simulation executes quite slowly when $\lambda(t) = 1000 + 100\sin(\pi t/12)$ is that the thinning method we used is very inefficient (lots of possible arrivals are rejected). Speculate about ways to make it faster.

6. For stochastic processes experts: Another reason that the simulation is slow when $\lambda(t) = 1000 + 100\sin(\pi t/12)$ is that there can be 1000 or more pending departure events in the event list at any time, which means that scheduling a new event in chronological order involves a slow search. However, it is possible to exploit the memoryless property of the exponential distribution of parking time to create an equivalent simulation that has only two pending events (the next car arrival and next car departure) at any point in time. Describe how to do this.

4.3 Simulating the $M/G/1$ Queue

Here we consider the hospital example of Sect. 3.2,a queueing system with Poisson arrival process, some (as yet unspecified) service-time distribution, and a single server (either a receptionist or an electronic kiosk); in other words, an $M/G/1$ queue. Patient waiting time is the key system performance measure, and the long-run average waiting time in particular.

Recall that Lindley's Equation (3.3) provides a shortcut way to simulate successive customer waiting times:

$$\begin{aligned} Y_0 &= 0 \qquad X_0 = 0 \\ Y_i &= \max\{0, Y_{i-1} + X_{i-1} - A_i\},\ i = 1, 2, \ldots \end{aligned}$$

where Y_i is the ith customer's waiting time, X_i is that customer's service time, and A_i is the interarrival time between customers $i-1$ and i. Lindley's equation avoids the need for an event-based simulation, but is limited in what it produces (how would you track the time-average number of customers in the queue?). In this section we will describe both recursion-based and event-based simulations of this queue, starting with the recursion.

4.3.1 Lindley Simulation of the $M/G/1$ Queue

To be specific, suppose that the mean time between arrivals is 1 minute, with the distribution being exponential, and the mean time to use the kiosk is 0.8 minutes (48 seconds), with the distribution being an Erlang-3. An Erlang-p is the sum of p i.i.d. exponentially distributed random variables, so an Erlang-3 with mean 0.8 is the sum of 3 exponentially distributed random variables each with mean 0.8/3.

In Sect. 3.2 we noted that the waiting-time random variables $Y_1, Y_2, \ldots$ converge in distribution to a random-variable Y, and it is $\mu = \mathbb{E}(Y)$ that we will use to summarize the performance of the queueing system. We also noted that $\bar{Y}(m) = m^{-1} \sum_{i=1}^{m} Y_i$ converges with probability 1 to μ as the number of customers simulated m goes to infinity.

All of this suggests that we make a very long simulation run (large m) and estimate μ by the average of the observed waiting times $Y_1, Y_2, \ldots, Y_m$. But this is not what we will do, and here is why: Any m we pick is not ∞, so the waiting times early in the run—which will tend to be smaller than μ because the queue starts empty—will likely pull $\bar{Y}(m)$ down. To reduce this effect, we will let the simulation generate waiting times for a while (say d of them) before starting to actually include them in our average. We will still make m large, but our average will only include the last $m - d$ waiting times. That is, we will use as our estimator the truncated average

$$\bar{Y}(m, d) = \frac{1}{m - d} \sum_{i=d+1}^{m} Y_i. \tag{4.1}$$

Table 4.1: Ten replications of the $M/G/1$ queue using Lindley's equation.

replication	$Y(55000, 5000)$
1	2.057990460
2	2.153527560
3	2.172301541
4	2.339207242
5	2.094318451
6	2.137171478
7	2.168302534
8	2.100971509
9	2.117776760
10	2.220187818
average	2.156175535
std dev	0.075074639

In addition, we will not make a single run of m customers, but instead will make n replications. This yields n i.i.d. averages $\bar{Y}_1(m,d), \bar{Y}_2(m,d), \ldots, \bar{Y}_n(m,d)$ to which we can apply standard statistical analysis. This avoids the need to directly estimate the asymptotic variance γ^2, a topic we defer to later chapters.

Figure 4.14 shows a Python simulation of the $M/G/1$ queue using Lindley's equation. In this simulation $m = 55{,}000$ customers, we discard the first $d = 5000$ of them, and make $n = 10$ replications.

The ten replication averages can be then individually written to an comma separated value (csv) file named "lindley.csv" (Fig. 4.15) and also printed out to the screen with the mean and standard deviation as in Table 4.1.

Notice that the average waiting time is a bit over 2 minutes, and that Python, like all programming languages, could display a very large number of output digits. How many are really meaningful? A confidence interval is one way to provide an answer.

Since the across-replication averages are i.i.d., and since each across-replication average is itself the within-replication average of a large number of individual waiting times (50,000 to be exact), the assumption of independent, normally distributed output data is reasonable. This justifies a t-distribution confidence interval on μ. The key ingredient is $t_{1-\alpha/2,n-1}$, the $1-\alpha/2$ quantile of the t distribution with $n-1$ degrees of freedom. If we want a 95% confidence interval, then $1-\alpha/2 = 0.975$, and our degrees of freedom are $10-1 = 9$. Since $t_{0.975,9} = 2.26$, we get $2.156175535 \pm (2.26)(0.075074639)/\sqrt{10}$ or $2.156175535 \pm 0.053653949$. This implies that we can claim with high confidence that μ is around 2.1 minutes, or we could give a little more complete information as 2.15 ± 0.05 minutes. Any additional digits are statistically meaningless.

Is an average of 2 minutes too long to wait? To actually answer that question would require some estimate of the corresponding wait to see the receptionist,

```
import numpy as np
#  Use scipy.stats because it includes the Erlang distribution
from scipy.stats import expon, erlang
import matplotlib.pyplot as plt
def lindley(m=55000, d = 5000):
    ''' Estimates waiting time with m customers, discarding the first d

        Lindley approximation for waiting time in a M/G/1 queue
    '''
    replications = 10
    lindley = []
    for rep in range(replications):
        y = 0
        SumY = 0
        for i in range(1, d):
            # Random number variable generation from scipy.stats
            # shape = 0, rate =1, 1 value
            a = expon.rvs(0, 1)
            # rate = .8/3, shape = 3
            x = erlang.rvs(3, scale = 0.8/3, size=1)
            y = max(0, y + x - a)
        for i in range(d, m):
            a = expon.rvs(0, 1)
            # rate = .8/3, shape = 3
            x = erlang.rvs(3, scale = 0.8/3, size=1)
            y = max(0, y + x - a)
            SumY += y
        result = SumY / (m - d)
        lindley.append(result)
    return lindley
```

Figure 4.14: Simulation of the $M/G/1$ queue using Lindley's equation.

```
import csv
with open("lindley.csv"), "rb") as myFile:
    lindleyout = csv.writer(myFile)
    lindleyout.writerow("Waitingtime")
    for row in result:
        print row

for i in range(len(result)):
    print ("%1d & %11.9f " % (i+1, result[i]))
print("average & %11.9f" % (mean(result)))
print("std dev & %11.9f" % (std(result)))
```

Figure 4.15: Simulation of the $M/G/1$ queue using Lindley's equation.

```
import SimPy.Simulation as Sim
import numpy as np
from scipy.stats import expon, erlang
from random import seed

class G:
    maxTime = 55000.0    # minutes
    warmuptime = 5000.0
    timeReceptionist = 0.8  # mean, minutes
    phases = 3
    ARRint = 1.0      # mean, minutes
    theseed = 99999
\textbf
```

Figure 4.16: Declarations for the hospital simulation.

either from observational data or a simulation of the current system. Statistical comparison of alternatives is a topic of Chap. 8.

4.3.2 Event-based Simulation of the $M/G/1$ Queue

The simulation program consists of some global declarations (Fig. 4.16), declarations (Fig. 4.17), the main program (Fig. 4.18), some event routines (Fig. 4.19), running the simulation and reporting provided by SimPy.

This model will illustrate the **Process**, **Resource**, and **Monitor** classes. At a high level, here is what they do:

- **Process** objects are used to generate entities or to govern the behavior of entities in the system.
- **Resource** objects including **Level** and **Store** are used to designate re-

```
class Hospitalsim(Sim.Simulation):
    def run(self, theseed):
        np.random.seed(theseed)
        self.receptionist = Sim.Resource(name="Reception", capacity=1,
                unitName="Receptionist", monitored=True, sim=self)
        s = Arrivals('Source', sim=self)
        self.initialize()
        self.activate(s, s.generate(meanTBA=G.ARRint,
                resource=self.receptionist), at=0.0)
        self.simulate(until=G.maxTime)
        avgutilization = self.receptionist.actMon.timeAverage()[0]
        avgwait = self.receptionist.waitMon.mean()
        avgqueue = self.receptionist.waitMon.timeAverage()[0]
        leftinqueue= mg1.receptionist.waitMon.yseries()[-1:][0]
        return [avgwait, avgqueue, leftinqueue,avgutilization]
```

Figure 4.17: Main program for the hospital simulation.

sources that are required by a **Process** over the course of the simulation. Any of these can have a **Process** utilizing a unit of resource, or there could be **Process** in a queue waiting for a resource to be available.

- A **Resource** has units that are required by a **Process**.
- A **Level** is an undifferentiated item that can be taken or produced by a **Process**.
- A **Store** is an inventory of heterogeneous items where a **Process** will require a specific type of item from the **Store**.

- **Monitor** objects are used to record observations so that they may be analyzed later. In particular, a **Resource** can also have a **Monitor** to observe **Process** that are utilizing or waiting in a queue to utilize a unit of a **Resource**.

Figure 4.17 shows the declarations of the **Process** and **Resource** objects, specifically:

```
self.receptionist = Sim.Resource(name="Reception", capacity=1,
                unitName="Receptionist", monitored=True, sim=self)
s = Arrivals('Source', sim=self)
```

These are both declared within the Simulation object `Hospitalsim`, and both specify that they are a part of the current simulation (`sim=self`). While these could be declared as a local variable, the **Resource** `self.receptionist` is declared as part of the simulation object `self`. When it is declared, it can be given a `capacity`, a `qtype` (queue type, FIFO, LIFO, or priority), if it

is `preemptable`, and if it is `monitored` and if so, what `monitorType` (**Tally** or **Monitor**). Default values are assumed if these are not specified. If it is monitored, the observations of the units and processes (`monitored=True`) utilizing the receptionist can be exposed outside of the simulation program itself. The arrival **Process** `s` generates the arrivals to the process.

The simulation itself is initialized, then each process is activated through the `self.activate` function. So here, the arrival process `s` begins generating arrivals according to the class `s` (defined in Figure 4.18). Then the simulation is run using `self.simulate`. The simulation runs until the time designated in the argument `until=G.maxTime`. This can also be modified by using

The statements after `self.simulate()` are for data collection. The **Resource** `self.receptionist` includes a **Monitor** for both the resource itself `actMon` and the queue of processes waiting for the resource. Among the statistics that can be reported are the `mean`, `timeAverage`, and the data recorded for all processes that used that resource (such as the actual waiting time indicated by `yseries`).

Finally, at the end of the simulation, if the `HospitalSim` simulation was called by another function, the `return` function specifies the return values. These could be summary statistics or lists of recorded data which can be used by the calling function in any way.

The arriving patients are represented by two classes. First, the `Arrivals` class represents the source of customers. It `generate` `Patient` through `activate`, which creates instances of the `Patient`, which then requests a **Resource**. Next, the customers, represented by the `Patients` class. It's actions are governed by the `visit` function, which is called by the generating class `Arrivals` when a patient visit is `activate`. The `Patient` class is what actually claims a unit of the resource or is placed in the queue.

In this simulation we are interested in long-run performance, so the length of a replication is determined by whatever we decide is long enough (which is not an easy decision, actually). When we used Lindley's equation it was natural to specify the replication length in terms of the number of customers simulated. However, in more complex simulations with many different outputs, it is far more common to specify the replication length by a stopping time T chosen so that it will be long enough for all of the outputs. Similarly, if we plan to discard data, then it is easier to specify a time T_d at which all statistics will be cleared.

Let $\bar{Y}(T, T_d)$ be a replication average of all waiting times recorded between times T_d and T. Clearly $\bar{Y}(m, d)$ based on count, and $\bar{Y}(T, T_d)$ based on time, will have different statistical properties, but it is intuitively clear that both will be good estimators of μ if their arguments are fixed (not a function of the data) and large enough. Notice also that a run time and deletion time are ideal for continuous-time statistics like the time-average number in queue.

For this event-based simulation it is easy to record and report a number of statistics. **Monitor** objects and **Resource** objects that are `monitored` have methods which collect data on observations and can calculate statistics. Table 4.2 shows the results from 10 replications, along with the overall averages and 95% confidence interval halfwidths.

```
class Arrivals(Sim.Process):
    """ Source generates customers randomly """

    def generate(self, meanTBA, resource):
        i=0
        while self.sim.now() < G.maxTime:
            c = Patient(name="Patient%02d" % (i), sim=self.sim)
            self.sim.activate(c, c.visit(b=resource))
            t = expon.rvs(0, 1.0 / meanTBA, size = 1)
            yield Sim.hold, self, t
            i+=1

class Patient(Sim.Process):
    """ Patient arrives, is served and leaves """

    def visit(self, b):
        arrive = self.sim.now()
        yield Sim.request, self, b
        wait = self.sim.now() - arrive
        tib = erlang.rvs(G.phases,
               scale = float(G.timeReceptionist)/G.phases,
               size = 1)
        yield Sim.hold, self, tib
        yield Sim.release, self, b
```

Figure 4.18: Event routines for the hospital simulation.

```
hospitalreception = []
for i in range(10):
    mg1 = Hospitalsim()
    mg1.startCollection(when=G.warmuptime,
        monitors=mg1.allMonitors)
    result = mg1.run(4321 + i)
    hospitalreception.append(result)

hospitalaverage = mean(hospitalreception, axis=0)
hospitalstd = std(hospitalreception, axis=0)
hospital95 = [hstd * t.ppf(0.975, reps-1) for hstd in hospitalstd]
```

Figure 4.19: Initializing and running the hospital simulation.

Table 4.2: Ten replications of the $M/G/1$ queue using the event-based simulation.

Rep	Total Wait	Queue	Remaining	Utilization
1	3.213462878	2.160833599	0	0.798567300
2	3.280708784	2.246598372	1	0.808480012
3	3.165536548	2.108143837	12	0.793258928
4	2.879284462	1.919272701	6	0.794702843
5	3.222073305	2.176042086	0	0.800707100
6	3.179741189	2.145124000	4	0.800669924
7	3.201467600	2.170802331	1	0.801600605
8	3.095751521	2.062482343	3	0.795603232
9	3.445755412	2.331405016	5	0.802041612
10	3.039486198	2.032295429	2	0.796786730
average	3.172326790	2.135299972	3.400000000	0.799241829
std dev	0.141778429	0.108549951	3.469870315	0.004231459
95 C.I.	0.320725089	0.245557048	7.849391986	0.009572226

Again there are meaningless digits, but the confidence intervals can be used to prune them. For instance, for the mean total wait we could report 3.17 ± 0.32 minutes. How does this statistic relate to the 2.16 ± 0.08 minutes reported for the simulation via Lindley's equation? Mean total time in the kiosk system (which is what the event-based simulation estimates) consists of mean time waiting to be served (which is what the Lindley simulation estimates) plus the mean service time (which we know to be 0.8 minutes). So it is not surprising that these two estimates differ by about 0.8 minutes.

4.3.3 Issues and Extensions

1. In what situations does it make more sense to compare the simulated kiosk system to simulated data from the current receptionist system rather than real data from the current receptionist system?

2. It is clear that if all we are interested in is mean waiting time, defined either as time until service begins or total time including service, the Lindley approach is superior (since it is clearly faster, and we can always add in the mean service time to the Lindley estimate). However, if we are interested in the distribution of total waiting time, then adding in the mean service time does not work. How can the Lindley recursion be modified to simulate total waiting times?

3. How can the event-based simulation be modified so that it also records waiting time until service begins?

4. How can the event-based simulation be modified to clear statistics after exactly 5000 patients, and to stop at exactly 55,000 patients?

5. The experiment design method illustrated in the event-based simulation is often called the "replication-deletion" method. If we only had time to generate 500,000 waiting times, what issues should be considered in deciding the values of n (replications), m (run length) and d (deletion amount)? Notice that we must have $nm = 500{,}000$, and only $n(m - d)$ observations will be used for estimating μ.

6. An argument against summarizing system performance by long-run measures is that no system stays unchanged forever (55,000 patients is approximately 38 24-hour days during which time there could be staff changes, construction or emergencies), so a measure like μ is not a reflection of reality. The counter argument is that it is difficult, if not impossible, to model all of the detailed changes that occur over any time horizon (even the time-dependent arrival process in the $M(t)/M/\infty$ simulation is difficult to estimate in practice), so long-run performance at least provides an understandable summary measure ("If our process stayed the same, then over the long run...."). Also, it is often mathematically easier to obtain long-run measures than it is to estimate them by simulation (since simulations have to stop). Considering these issues, what sort of analysis makes the most sense for the hospital problem?

4.4 Simulating the Stochastic Activity Network

Here we consider the construction example of Sect. 3.4which is represented as a stochastic activity network (SAN). Recall that the time to complete the project, Y, can be represented as

$$Y = \max\{X_1 + X_4, X_1 + X_3 + X_5, X_2 + X_5\}$$

where X_i is the duration of the ith activity. This simple representation requires that we enumerate all paths through the SAN, so that the project completion time is the longest of these paths. Path enumeration itself can be time consuming, and this approach does not easily generalize to projects that have resources shared between activities, for instance. Therefore, we also present a discrete-event representation which is more complicated, but also more general.

4.4.1 Maximum Path Simulation of the SAN

Figure 4.20 shows a Python implementation of the algorithm displayed in Sect. 3.4 and repeated here:

1. set $s = 0$
2. repeat n times:
   (a) generate $X_1, X_2, \ldots, X_5$
   (b) set $Y = \max\{X_1 + X_4, X_1 + X_3 + X_5, X_2 + X_5\}$
   (c) if $Y > t_p$ then set $s = s + 1$
3. estimate θ by $\widehat{\theta} = s/n$

Since $\Pr\{Y \leq t_p\}$ is known for this example (see Eq. 3.12), the true $\theta = \Pr\{Y > t_p\} = 0.16533$ when $t_p = 5$ is also computed by the program so that we can compare it to the simulation estimate. Of course, in a practical problem we would not know the answer, and we would be wasting our time simulating it if we did. Notice that even if all of the digits in this probability estimate are correct, they certainly are not practically useful.

The simulation estimate turns out to be $\widehat{\theta} = 0.15400$. A nice feature of a probability estimate that is based on i.i.d. outputs is that an estimate of its standard error is easily computed:

$$\widehat{\text{se}} = \sqrt{\frac{\widehat{\theta}(1-\widehat{\theta})}{n}}.$$

Thus, $\widehat{\text{se}}$ is approximately 0.011, and the simulation has done its job since the true value θ is well within $\pm 1.96\,\widehat{\text{se}}$ of $\widehat{\theta}$. This is a reminder that simulations do not deliver *the answer*, like Eq. 3.12, but do provide the capability to estimate the simulation error, and to reduce that error to an acceptable level by increasing the simulation effort (number of replications).

4.4.2 Discrete-event Simulation of the SAN

This section uses the NetworkX graph library and may be skipped without loss of continuity.

As noted in Sect. 3.4,we can think of the completion of a project activity as an event, and when all of the inbound activities $\mathcal{I}(j)$ to a milestone j are completed then the outbound activities $i \in \mathcal{O}(j)$ can be scheduled, where the destination milestone of activity i is $\mathcal{D}(i)$. Thus, the following generic milestone event is the only one needed:

event milestone (activity ℓ inbound to node j)
$\mathcal{I}(j) = \mathcal{I}(j) - \ell$
if $\mathcal{I}(j) = \emptyset$ then
    for each activity $i \in \mathcal{O}(j)$
        schedule milestone(activity i inbound to node $\mathcal{D}(i)$ to occur X_i time units later)
end if

```
import math, random

class SAN:
    N =  1000
    tp = 5.0

def constructionsan(seed):
    random.seed(seed)
    X = [random.expovariate(1.0) for i in range(5)]
    Y = max(X[0]+X[3], X[0]+X[2] + X[4], X[1] + X[4])
    return Y

initialseed = 2124
Y = [constructionsan(initialseed + i) for i in range(SAN.N)]
Ytp = [1.0 if Y[i]>SAN.tp else 0 for i in range(SAN.N)]
thetahat = sum(Ytp)/SAN.N
sethetahat = math.sqrt((thetahat*(1-thetahat))/SAN.N)

Theta = 1 - ( (math.pow(SAN.tp,2) / 2 - 3.0 * SAN.tp - 3) * \
    math.exp(-2 * SAN.tp) + (-1.0/2 *math.pow(SAN.tp, 2) - 3 * SAN.tp + 3) * \
    math.exp(-SAN.tp) + 1.0 - math.exp(-3 * SAN.tp))

Theta = 1 - ( (math.pow(SAN.tp,2) / 2 - 3.0 * SAN.tp - 3) * \
    math.exp(-2 * SAN.tp)  + (-1.0/2 *math.pow(SAN.tp, 2) - 3 * SAN.tp + 3) * \
    math.exp(-SAN.tp) + 1.0 - math.exp(-3 * SAN.tp))
```

Figure 4.20: Simulation of the SAN as the maximum path through the network.

```
import random
import SimPy.Simulation as Sim
import networkx as nx

class SANglobal:
    F = nx.DiGraph()
    a = 0
    b = 1
    c = 2
    d = 3
    inTo = 0
    outOf = 1
    F.add_nodes_from([a, b, c, d])
    F.add_edges_from([(a,b), (a,c), (b,c), (b,d), (c,d)])
```

Figure 4.21: Network description for the discrete-event SAN simulation.

Of course, this approach shifts the effort from enumerating all of the paths through the SAN to creating the sets $\mathcal{I}, \mathcal{O}, \mathcal{D}$, but these sets have to be either explicitly or implicitly defined to define the project itself. The key lesson from this example, which applies to many simulations, is that it is possible to program a single event routine to handle many simulation events that are conceptually distinct, and this is done by passing event-specific information to the event routine.

In this case we need to develop the configuration of that activity network and use that description to direct the simulation. To do so we will use the NetworkX[11] graph library. NetworkX is a Python language software library for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. While it includes a wide range of graph algorithms, we will use it as a standard representation of graphs such as the stochastic activity network.

Using NetworkX, we create a directed graph (`nx.DiGraph()`) with four nodes with five directed edges. (Figure 4.21) Implicitely, it also creates predecessor and successor lists for each node that can be accessed using `F.predecessors(i)` or `F.successors(i)`.

We then define events as the completion of activities that go into a given node. Events trigger a signal that can be used to trigger other activities.

In the first block of code in Figure 4.22 an event is defined for each node in the network and added to a list of events (`nodecomplete`) which corresponds to the list of nodes. Then, for each node, the list of predecessor events is created (`preevents`) and the node is created as an `ActivityProcess`. (Figure)

For each `ActivityProcess`, the `waitup()` function is focused on the `waitevent` event. This takes as an arguement `myEvent`, which is the list of predecessor

[11] http://networkx.github.io/

```
SANglobal.finishtime = 0
Sim.initialize()
SANglobal.F.nodecomplete= []
for i in range(len(SANglobal.F.nodes())):
    eventname = 'Complete%1d' % i
    SANglobal.F.nodecomplete.append(Sim.SimEvent(eventname))
SANglobal.F.nodecomplete

activitynode = []
for i in range(len(SANglobal.F.nodes())):
    activityname = 'Activity%1d' % i
    activitynode.append(ActivityProcess(activityname))
for i in range(len(SANglobal.F.nodes())):
    if i <> SANglobal.inTo:
        prenodes = SANglobal.F.predecessors(i)
        preevents = [SANglobal.F.nodecomplete[j] for j in prenodes]
        Sim.activate(activitynode[i], activitynode[i].waitup(i,preevents))
startevent = Sim.SimEvent('Start')
Sim.activate(activitynode[SANglobal.inTo],
        activitynode[SANglobal.inTo].waitup(SANglobal.inTo, startevent))
sstart = StartSignaller('Signal')
Sim.activate(sstart, sstart.startSignals())
Sim.simulate(until=50)
```

Figure 4.22: Main SAN DES simulation.

```
class ActivityProcess(Sim.Process):
    def waitup(self,node, myEvent):      # PEM illustrating "waitevent"
                                    # wait for "myEvent" to occur
        yield Sim.waitevent, self, myEvent
        tis = random.expovariate(1.0)
        print ('   The activating event(s) were %s' %
             ([x.name for x in self.eventsFired]))
        yield Sim.hold, self, tis
        finishtime = Sim.now()
        if finishtime >SANglobal.finishtime:
            SANglobal.finishtime = finishtime
        SANglobal.F.nodecomplete[node].signal()
```

Figure 4.23: Activity process waits for events to be cast.

```
class StartSignaller(Sim.Process):
    # here we just schedule some events to fire
    def startSignals(self):
        yield Sim.hold, self, 0
        startevent.signal()
```

Figure 4.24: StartSignaller class for initiating the SAN simulation.

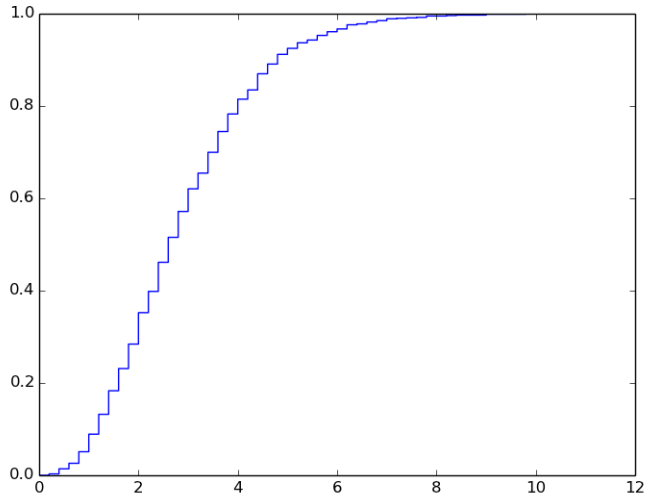

Figure 4.25: Empirical cdf of the project completion times.

events that were identified in the main simulation. As each in the `myEvent` list occurs, it broadcasts its associated signal using the `signal()` function. When all the events in a `ActivityProcess waitevent` list (`myEvent` in Figure **??** have occurred, the `yield` condition is met and the next line in `ActivityProcess` begins.

To start the simulation, we create a **Process** that will provide the initiating event of the simulation. (Figure 4.24) Similarly, we treat the initial node of the simulation differently by having it wait for the start signal to begin instead of waiting for predecessor events like the other nodes.

Notice (see Fig. 4.22) that the simulation ends when there are no additional activities remaining to be completed.

A difference between this implementation of the SAN simulation and the one in Sect. 4.4.1 is that here we write out the actual time the project completes on each replication. By doing so, we can estimate $\Pr\{Y > t_p\}$ for any value of t_p by sorting the data and counting how many out of 1000 replications were greater

than t_p. Figure 4.25 shows the empirical cdf of the 1000 project completion times, which is the simulation estimate of Eq. (3.12).

4.4.3 Issues and Extensions

1. In real projects there are not only activities, but also limited and often shared resources that are needed to complete the activities. Further, there may be specific resource allocation rules when multiple activities contend for the same resource. How might this be modeled in SimPy?

2. Time to complete the project is an important overall measure, but at the planning stage it may be more important to discover which activities or resources are the most critical to on-time completion of the project. What additional output measures might be useful for deciding which activities are "critical?"

4.5 Simulating the Asian Option

Here we consider estimating the value of an Asian option

$$\nu = \mathbb{E}\left[\mathrm{e}^{-rT}(\bar{X}(T) - K)^+\right]$$

as described in Sect. (3.5),where the maturity is $T = 1$ year, the risk-free interest rate is $r = 0.05$ and the strike price is $K = \$55$. The underlying asset has an initial value of $X(0) = \$50$ and the volatility is $\sigma^2 = (0.3)^2$. Recall that the key quantity is

$$\bar{X}(T) = \frac{1}{T}\int_0^T X(t)\,\mathrm{d}t$$

the time average of a continuous-time, continuous-state geometric Brownian motion process which we cannot truly simulate on a digital computer. Thus, we approximate it by dividing the interval $[0, T]$ into m steps of size $\Delta t = T/m$ and using the discrete approximation

$$\widehat{\bar{X}(T)} = \frac{1}{m}\sum_{i=1}^{m} X(i\Delta t).$$

This makes simulation possible, since

$$X(t_{i+1}) = X(t_i)\exp\left\{\left(r - \frac{1}{2}\sigma^2\right)(t_{i+1} - t_i) + \sigma\sqrt{t_{i+1} - t_i}\,Z_{i+1}\right\}$$

for any increasing sequence of times $\{t_0, t_1, \ldots, t_m\}$, where $Z_1, Z_2, \ldots, Z_m$ are i.i.d. $\mathrm{N}(0, 1)$.

Figure 4.26 is Python code that uses $m = 32$ steps in the approximation, and makes 10,000 replications to estimate ν. Discrete-event structure would slow execution without any obvious benefit, so a simple loop is used to advance

Table 4.3: Arrival rate of faxes by hour.

Time	Rate (faxes/minute)
8 AM–9 AM	4.37
9 AM–10 AM	6.24
10 AM–11 AM	5.29
11 AM–12 PM	2.97
12 PM–1 PM	2.03
1 PM–2 PM	2.79
2 PM–3 PM	2.36
3 PM–4 PM	1.04

time. The value of the option from each replication is saved to a list for post-simulation analysis.

The estimated value of ν is \$2.20 with a relative error of just over 2% (recall that the relative error is the standard error divided by the mean). As the histogram in Fig. 4.27 shows, the option is frequently worthless (approximately 68% of the time), but the average payoff, conditional on the payoff being positive, is approximately \$6.95.

4.6 Case Study: Service Center Simulation

This section presents a simulation case based on a project provided by a former student. While still relatively simple, it is more complex than the previous stylized examples, and the answer is not known without simulating. The purpose of this section is to illustrate how one might attack simulation modeling and programming for a realistic problem.

Example 4.1 (Fax Center Staffing) *A service center receives faxed orders throughout the day, with the rate of arrival varying hour by hour. The arrivals are modeled by a nonstationary Poisson process with the rates shown in Table 4.3.*

A team of Entry Agents select faxes on a first-come-first-served basis from the fax queue. Their time to process a fax is modeled as normally distributed with mean 2.5 *minutes and standard deviation* 1 *minute. There are two possible outcomes after the Entry Agent finishes processing a fax: either it was a simple fax and the work on it is complete, or it was not simple and it needs to go to a Specialist for further processing. Over the course of a day, approximately 20% of the faxes require a Specialist. The time for a Specialist to process a fax is modeled as normally distributed with mean* 4.0 *minutes and standard deviation* 1 *minute.*

Minimizing the number of staff minimizes cost, but certain service-level requirements much be achieved. In particular, 96% of all simple faxes should be

completed within 10 *minutes of their arrival, while 80% of faxes requiring a Specialist should also be completed (by both the Entry Agent and the Specialist) within* 10 *minutes of their arrival.*

The service center is open from 8 AM to 4 PM daily, and it is possible to change the staffing level at 12 PM. Thus, a staffing policy consists of four numbers: the number of Entry Agents and Specialists before noon, and the number of Entry Agents and Specialists after noon. Any fax that starts its processing before noon completes processing by that same agent before the agent goes off duty; and faxes in the queues at the end of the day are processed before the agents leave work and therefore are not carried over to the next day.

The first step in building any simulation model is deciding what question or questions that the model should answer. Knowing the questions helps identify the system performance measures that the simulation needs to estimate, which in turn drives the scope and level of detail in the simulation model.

The grand question for the service center is, what is the minimum number of Entry Agents and Specialists needed for both time periods to meet the service-level requirements? Therefore, the simulation must at least provide an estimate of the percentage of faxes of each type entered within 10 minutes, given a specific staff assignment.

Even when there seems to be a clear overall objective (minimize the staff required to achieve the service-level requirement), we often want to consider trade offs around that objective. For instance, if meeting the requirement requires a staff that is so large that they are frequently underutilized, or if employing the minimal staff means that the Entry Agents or Specialists frequently have to work well past the end of the day, then we might be willing to alter the service requirement a bit. Statistics on the number and the time spent by faxes in queue, and when the last fax of each day is actually completed, provide this information. Including additional measures of system performance, beyond the most critical ones, makes the simulation more useful.

Many discrete-event, stochastic simulations involve entities that dynamically flow through some sort of queueing network where they compete for resources. In such simulations, identifying the entities and resources is a good place to start the model. For this service center the faxes are clearly the dynamic entities, while the Entry Agents and Specialists are resources. The fax machines themselves might also be considered a resource, especially if they are heavily utilized or if outgoing as well as incoming faxes use the same machines. It turns out that for this service center there is a bank of fax machines dedicated to incoming faxes, so it is reasonable to treat the arrival of faxes as an unconstrained external arrival process. This fact was not stated in the original description of the problem; follow-up questions are often needed to fully understand the system of interest.

Whenever there are scarce resources, queues may form. Queues are often first-in-first-out, with one queue for each resource, as they are in this service center. However, queues may have priorities, and multiple queues may be served by the same resource, or a single queue may feed multiple resources. Queueing

behavior is often a critical part of the model.

When the simulation involves entities flowing through a network of queues, then there can be two types of arrivals: arrivals from outside of the network and arrivals internal to the network. Outside arrivals are like those we have already seen in the $M(t)/M/\infty$ and $M/G/1$ examples. Internal arrivals are departures from one queue that become arrivals to others. How these are modeled depends largely on whether the departure from one queue is an immediate arrival to the next—in which case the departure and arrival events are effectively the same thing—or whether there is some sort of transportation delay—in which case the arrival to the next queue should be scheduled as a distinct event. For the service center the arrival of faxes to the Entry Agents is an outside arrival process, while the 20% of faxes that require a Specialist are internal arrivals from the Entry Agents to the Specialists.

Critical to experiment design is defining what constitutes a replication. Replications should be independent and identically distributed. Since the service center does not carry faxes over from one day to the next, a "day" defines a replication. If faxes did carry over, but all faxes are cleared weekly, then a replication might be defined by a work week. However, if there is always significant carry over from one day to the next, then a replication might have to be defined arbitrarily.

The work day at the service center is eight hours; however the staff does not leave until all faxes that arrive before 4 PM are processed. If we defined a replication to be exactly eight hours then we could be fooled by a staffing policy that allows a large queue of faxes to build up toward the end of the day, since the entry of those faxes would not be included in our statistics. To model a replication that ends when there is no additional work remaining, we will cut off the fax arrivals at 4 PM and then end the simulation when the event calendar is empty. This works because idle Entry Agents and Specialists will always take a fax from their queue if one is available.

Rather than walk through the SimPy code line by line, we will point out some highlights to facilitate the reader's understanding of the code.

Figure 4.28 shows the global declarations for the service center simulation.

```
import math, random

def Asianoption(interestrate, sigma, steps, initialValue, strikePrice, maturity, seed):
    """  Asian Options simulation


    """
    sumx = 0.0
    # Create instance of a random number object
    generator = random.Random()
    generator.seed(seed)
    x = initialValue
    sigma2 = (sigma*sigma)/2.0
    for j in range(steps):
       z = generator.normalvariate(0, 1)
       x = x * math.exp((interestRate - sigma2) * interval + sigma * math.sqrt(interval) * z)
       sumx = sumx + x
    value = math.exp(-interestRate * maturity) * max(sumx / float(steps) - strikePrice, 0.0)
    return value
replications = 10000
initialSeed = 1234
maturity = 1.0
steps = 32
sigma = 0.3
interestRate = 0.05
initialValue = 50.0
strikePrice = 55.0
interval = maturity / float(steps)

values = [Asianoption(interestRate, sigma, steps, initialValue, strikePrice, maturity, \
i + initialSeed) for i in range(replications)]
print(mean(values))
print(std(values)/math.sqrt(replications)) # standard error
print(std(values)/math.sqrt(replications)/mean(values)) # relative error
```

Figure 4.26: Python Simulation of the Asian option problem.

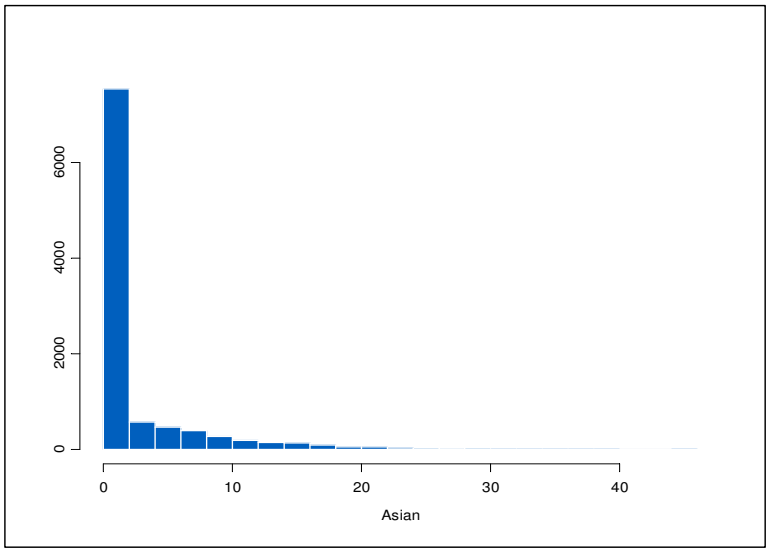

Figure 4.27: Histogram of the realized value of the Asian option from 10,000 replications.

```
import random
import SimPy.Simulation as Sim

class F:
    maxTime = 100    # hours
    seedval = 1234
    period = 60.0
    nPeriods = 8 # periods per day
    meanRegular = 2.5/period # hours
    varRegular = 1.0/period # hours
    stdRegular = math.sqrt(1.0)/period
    meanSpecial = 4.0/period # hours
    varSpecial = 1.0/period # hours
    stdSpecial = math.sqrt(1.0)/period
    tPMshiftchange = 4.0
    numAgents = 15
    numAgentsPM = 9
    numSpecialists = 6
    numSpecialistsPM = 3
    maxRate = 6.24
    aRate= [4.37, 6.24, 5.29, 2.97, 2.03, 2.79, 2.36, 1.04]
    aRateperhour = [aRate[i] * period for i in range(len(aRate))]
    meanTBA = 1/(maxRate * period) # hours
    pSpecial = 0.20
```

Figure 4.28: Global declarations for service center simulation.

The main program for the simulation is in Fig. 4.29. Of particular note are the two **Monitor** statements defining `Regular10` and `Special10`. These will be used to obtain the fraction of regular and special faxes that are processed within the 10-minute requirement by recording a 1 for any fax that meets the requirement, and a 0 otherwise. The mean of these values is the desired fraction.

Also notice is the condition that ends the main simulation loop:

```
self.simulate(until=F.maxTime)
```

Because the simulation ends well after the arrivals cease, any faxes still in the queue will be completed prior to the end of the simulation. When the event calendar is empty, then there are no additional faxes to process, and no pending arrival of a fax. This condition will only hold after 4 PM and once all remaining faxes have been entered.

```
class Faxcentersim(Sim.Simulation):
    def run(self, aseed):
        random.seed(aseed)
        self.agents = Sim.Resource(capacity=F.numAgents,
            name="Service Agents", unitName="Agent", monitored = True,
            qType=Sim.PriorityQ, sim=self)
        self.specialagents = Sim.Resource(capacity=F.numSpecialists,
            name="Specialist Agents", unitName="Specialist", monitored=True,
            qType=Sim.PriorityQ, sim=self)
        self.meanTBA = 0.0
        self.initialize()
        s = Source('Source', sim=self)
        a = ArrivalRate('Arrival Rate', sim=self)
        tchange = SecondShift('PM Shift', sim=self)
        self.Regularwait = Sim.Monitor(name="Regular time",
            ylab='hours', sim=self)
        self.Specialistwait = Sim.Monitor(name="Special time",
            ylab='hours', sim=self)
        self.activate(a, a.generate(F.aRateperhour))
        self.activate(s,
            s.generate(resourcenormal=self.agents,
            resourcespecial=self.specialagents), at=0.0)
        self.activate(tchange, tchange.generate(F.tPMshiftchange,
            resourcenormal=self.agents,
            resourcespecial=self.specialagents))
        self.simulate(until=F.maxTime)
    def reporting(self):
        regularcount = self.Regularwait.count()
        regularwait = self.Regularwait.mean()
        regularQ = self.agents.waitMon.timeAverage()
        regularagents = self.agents.actMon.timeAverage()
        regular10min = sum([1.0 if waittime < 1./6 else 0
            for waittime in self.Regularwait.yseries()])
        fractionregular10min = regular10min/regularcount
        specialcount = self.Specialistwait.count()
        specialwait = self.Specialistwait.mean()
        specialQ = self.specialagents.waitMon.timeAverage()
        specialagents = self.specialagents.actMon.timeAverage()
        special10min = sum([1.0 if waittime < 1./6 else 0
            for waittime in self.Specialistwait.yseries()])
        fractionspecial10min = special10min/specialcount
        result = [regularwait, regularQ, regularagents,
            specialwait, specialQ, specialagents,
            fractionregular10min, fractionspecial10min]
        return result
reps=10
faxsimulation = []
for i in range(reps):
    faxsim = Faxcentersim()
    faxsim.run(F.seedval + i)
    result = faxsim.reporting()
    faxsimulation.append(result)

mean(faxsimulation, axis=0)
std(faxsimulation, axis=0)
```

Figure 4.29: Main program and statistics reporting for service center simulation.

Figure 4.30 includes processes that generate faxes (`Source`) and determine how they are routed to agents (`Fax`). As faxes are routed, the simulation determines if they require special handling after the regular agent is complete (`if (checkSpecial < F.pSpecial)`). There are also two **Monitor** present, `Regularwait` and `Specialistwait` which record the total wait time for each fax that is completed.

```
            self.sim.Regularwait.observe(finished)
```

Later, in the `reporting()`) function of the main program shown in Figure 4.29, this **Monitor** will be used to both get the number of total faxes of this type, the number that waited less than 10 minutes before completing processing, and the fraction.

```
        regularcount = self.Regularwait.count()
        regular10min = sum([1.0 if waittime < 1./6 else 0
            for waittime in self.Regularwait.yseries()])
        fractionregular10min = regular10min/regularcount
```

While collecting statistics in this fashion provides the most flexibility as it keeps the wait time observations for future use, we could have instead recorded if the wait time was less than 10 minutes.

```
if finished < 1.0/6:  # 10 minutes
            self.sim.Regularwait.observe(1)
else:
            self.sim.Regularwait.observe(0)
```

Then we could have calculated the fraction by taking the sum of the observations divided by the number of observations.

```
        fractionregular10min = float(sum(self.Regularwait))/
            len(self.Regularwait)
```

The `Fax.reporting()` then uses the **Monitor** for wait time as well as the monitors associated with the `agents` and `specialagents` resources. The `actMon` monitors track how the resources are being used while the `waitMon` monitors track the waiting queue for each monitor. For resources, we tend to be interested in the time average value of the size of the queue or the number of units of resource in use so the `agents.actMon.timeAverage()` function returns the average utilization of the agents while `specialagents.waitMon.timeAverage()` function returns the average number of special faxes in queue.

Ten replications of this simulation with a staffing policy of 15 Entry Agents in the morning and 9 in the afternoon, and 6 Specialists in the morning and 3 in the afternoon, gives 0.98 ± 0.04 for the fraction of regular faxes entered in 10 minutes or less, and 0.84 ± 0.08 for the special faxes. The "$\pm$" are 95% confidence intervals. This policy appears to be close to the requirements, although if we absolutely insist on 80% for the special faxes then additional replications are needed to narrow the confidence interval.

```
class Source(Sim.Process):
    """ Source generates customers randomly """

    def generate(self, resourcenormal, resourcespecial):
        i = 0
        while(self.sim.now() < F.nPeriods and self.sim.meanTBA>0):
            f = Fax(name="Fax%02d" % (i), sim=self.sim)
            self.sim.activate(f,  \
                f.visit(agent=resourcenormal, specialagent = resourcespecial))
            t = random.expovariate(1.0 / self.sim.meanTBA)
            yield Sim.hold, self, t
            i += 1


class Fax(Sim.Process):
    """ Fax arrives and is procossed

        Processed first by regular staff, then specialized staff if needed
        Assume that anyone working on a fax will continue working until it finishes.
    """

    def visit(self, agent, specialagent):
        arrive = self.sim.now()
        yield Sim.request, self, agent
        wait = self.sim.now() - arrive
        tis = -1.0
        while (tis < 0):
            tis = random.normalvariate(F.meanRegular, F.stdRegular)
        yield Sim.hold, self, tis
        yield Sim.release, self, agent
        checkSpecial = random.random()
        if (checkSpecial < F.pSpecial):
            tspecial = -1.0
            while (tspecial < 0):
                tspecial = random.normalvariate(F.meanSpecial, F.stdSpecial)
            yield Sim.request, self, specialagent
            yield Sim.hold, self, tspecial
            yield Sim.release, self, specialagent
            finished = self.sim.now() - arrive
            self.sim.Specialistwait.observe(finished)
        else:
            finished = self.sim.now() - arrive
            self.sim.Regularwait.observe(finished)
```

Figure 4.30: Processes for arriving agents.

```
class ArrivalRate(Sim.Process):
    """ update the arrival rate every hour

        Reads in the arrival rate table and updates the arrival rate every hour
        One hour after the last hour begins, changes arrival rate to 0
    """
    def generate(self, arrivalrate):
        for i in range(len(arrivalrate)):
            self.sim.meanTBA = 1.0/(arrivalrate[i])
            yield Sim.hold, self, 1.0
        # After the end of the day, set the arrival rate = 0
        self.sim.meanTBA = 0.0

class SecondShift(Sim.Process):
    """ Trigger the change in shifts for agents

        The effect should be to move the wait queue to the new set of agents

    """

    def generate(self, tshiftchange, resourcenormal, resourcespecial):
        yield Sim.hold, self, tshiftchange
        reduceagents = F.numAgents - F.numAgentsPM
        reducespecial = F.numSpecialists - F.numSpecialistsPM

        for i in range(reduceagents):
            a = Agentoff(name="removeagent%02d" % (i), sim=self.sim)
            self.sim.activate(a, a.visit(resourcenormal))
        for j in range(reducespecial):
            a = Agentoff(name="removespecial%02d" % (i) , sim=self.sim)
            self.sim.activate(a, a.visit(resourcespecial))

class Agentoff(Sim.Process):
    def visit(self, agent):
        tremain = F.maxTime - self.sim.now()+1
        # use priority 100 to ensure agent is removed as soon as current fax is done
        yield Sim.request, self, agent, 100
        yield Sim.hold, self, tremain
        yield Sim.release, self, agent
```

Figure 4.31: Processes handling changes over time.

```
class F:
    maxTime = 100.0    # hours
    theseed = 9999

    period = 60.0
    nPeriods = 8
    meanRegular = 2.5/period # hours
    varRegular = 1.0/period # hours
    stdRegular = math.sqrt(1.0)/period
    meanSpecial = 4.0/period # hours
    varSpecial = 1.0/period # hours
    stdSpecial = math.sqrt(1.0)/period

    tPMshiftchange = 4.0
    numAgents = 15
    numAgentsPM = 9
    numSpecialists = 6
    numSpecialistsPM = 3

    maxRate = 6.24
    aRate= [4.37, 6.24, 5.29, 2.97, 2.03, 2.79, 2.36, 1.04] # per minute
    aRateperhour = [aRate[i] * period for i in range(len(aRate))] # per hour
    meanTBA = 1/(maxRate * period) # hours
    pSpecial = 0.20
```

Figure 4.32: Parameters for service center simulation.

4.6.1 Issues and Extensions

1. There are many similarities between the programming for this simulation and the event-based simulation of the $M/G/1$ queue.

2. The fax entry times were modeled as being normally distributed. However, the normal distribution admits negative values, which certainly does not make sense. What should be done about this? Consider mapping negative values to 0, or generating a new value whenever a negative value occurs. Which is more likely to be realistic and why?

Exercises

1. For the hospital problem, simulate the current system in which the receptionist's service time is well modeled as having an Erlang-4 distribution

with mean 0.6 minutes. Compare the waiting time to the proposed electronic kiosk alternative.

2. Simulate an $M(t)/G/\infty$ queue where G corresponds to an Erlang distribution with fixed mean but try different numbers of phases. That is, keep the mean service time fixed but change the variability. Is the expected number if queue sensitive to the variance in the service time?

3. Modify the SAN simulation to allow each activity to have a different mean time to complete (currently they all have mean time 1). Use a Collection to hold these mean times.

4. Try the following numbers of steps for approximating the value of the Asian option to see how sensitive the value is to the step size: $m = 8, 16, 32, 64, 128$.

5. In the simulation of the Asian option, the sample mean of 10,000 replications was 2.198270479, and the standard deviation was 4.770393202. Approximately how many replications would it take to decrease the relative error to less than 1%?

6. For the service center, increase the number of replications until you can be confident that that suggested policy does or does not achieve the 80% entry in less than 10 minutes requirement for special faxes.

7. For the service center, find the minimum staffing policy (in terms of total number of staff) that achieves the service-level requirement. Examine the other statistics generated by the simulation to make sure you are satisfied with this policy.

8. For the service center, suppose that Specialists earn twice as much as Entry Agents. Find the minimum cost staffing policy that achieves the service-level requirement. Examine the other statistics generated by the simulation to make sure you are satisfied with this policy.

9. For the service center, suppose that the staffing level can change hourly, but once an Agent or Specialist comes on duty they must work for four hours. Find the minimum staffing policy (in terms of total number of staff) that achieves the service-level requirement.

10. For the service center, pick a staffing policy that fails to achieve the service level requirements by 20% or more. Rerun the simulation with a replication being defined as exactly 8 hours, but do not carry waiting faxes over to the next day. How much do the statistics differ using the two different ways to end a replication?

11. The function `NSPP_Fax` is listed below. This function implements the thinning method described in Sect. 4.2 for a nonstationary Poisson process with piecewise-constant rate function. Study it and describe how it works.

```
def NSPP_Fax(arrivalrate, MaxRate, NPeriods, periodlength, stream):
    '''  Non-stationary Poisson Process

    This function generates interarrival times from a NSPP with
    piecewise constant arrival rate over a fixed time of NPeriod time units

    arrivalrate - Array of arrival rates over a common length period
    MaxRate - The maximum value of ARate
    NPeriods - Number of time periods in ARate
    periodlength - time units between (possible) changes in arrival rate
    '''
    random.seed(stream)
    pthinning = [(1-hourlyrate/MaxRate) for hourlyrate in arrivalrate]
    t = 0.0
    arrivaltimes = []
    totaltime = NPeriods * periodlength
    while t < totaltime:
        deltat = random.expovariate(MaxRate)
        t = t + deltat
        if t < totaltime:
            pthin = pthinning[int(floor(t/periodlength))]
            uthin = random.random()
            if uthin > pthin:
                arrivaltimes.append(float(t)) # add arrival since not thinned
    return arrivaltimes
```

12. Beginning with the event-based $M/G/1$ simulation, implement the changes necessary to make it an $M/G/s$ simulation (a single queue with any number of servers). Keeping $\lambda = 1$ and $\tau/s = 0.8$, simulate $s = 1, 2, 3$ servers and compare the results. What you are doing is comparing queues with the same service capacity, but with 1 fast server as compared to two or more slower servers. State clearly what you observe.

13. Modify the SimPy event-based simulation of the $M/G/1$ queue to simulate an M/G/1/c retrial queue. This means that customers who arrive to find c customers in the system (including the customer in service) leave immediately, but arrive again after an exponentially distributed amount of time with mean `MeanTR`. Hint: The existence of retrial customers should not affect the arrival process for first-time arrivals.

14. This problem assumes a more advanced background in stochastic processes. In the simulation of the $M(t)/M/\infty$ queue there could be a very large number of events on the event calendar: one "Arrival" and one "Departure" for *each* car currently in the garage. However, properties of the exponential distribution can reduce this to no more than two events. Let $\beta = 1/\tau$ be the departure rate for a car (recall that τ is the mean parking time). If at any time we observe that there are N car in the garage (no matter how long they have been there), then the time until the first of

these cars departs is exponentially distributed with mean $1/(N\beta)$. Use this insight to build an $M(t)/M/\infty$ simulation with at most two pending events, next arrival and next departure. Hint: Whenever an arrival occurs the distribution of the time until the next departure changes, so the scheduled next departure time must again be generated.

15. The phone desk for a small office is staffed from 8 AM to 4 PM by a single operator. Calls arrive according to a Poisson process with rate 6 per hour, and the time to serve a call is uniformly distributed between 5 and 12 minutes. Callers who find the operator busy are placed on hold, if there is space available, otherwise they receive a busy signal and the call is considered "lost." In addition, 10% of callers who do not immediately get the operator decide to hang up rather than go on hold; they are not considered lost, since it was their choice. Because the hold queue occupies resources, the company would like to know the smallest capacity (number of callers) for the hold queue that keeps the daily fraction of lost calls under 5%. In addition, they would like to know the long-run utilization of the operator to make sure he or she will not be too busy. Use SimPy to simulate this system and find the required capacity for the hold queue. Model the callers as a **Process** and the operator as a **Resource**. Use the functions `random.expovariate` and `random.uniform` for random-variate generation. Estimate the fraction of calls lost (record a 0 for calls not lost, a 1 for those that are lost so that the sample mean is the fraction lost). Use the statistics collected by class **Resource** to estimate the utilization.

16. Software Made Personal (SMP) customizes software products in two areas: financial tracking and contact management. They currently have a customer support call center that handles technical questions for owners of their software from the hours of 8 AM to 4 PM Eastern Time.

    When a customer calls they the first listen to a recording that asks them to select among the product lines; historically 59% are financial products and 41% contact management products. The number of customers who can be connected (talking to an agent or on hold) at any one time is essentially unlimited. Each product line has its own agents. If an appropriate agent is available then the call is immediately routed to the agent; if an appropriate agent is not available, then the caller is placed in a hold queue (and listens to a combination of music and ads). SMP has observed that hang ups very rarely happen.

    SMP is hoping to reduce the total number of agents they need by cross-training agents so that they can answer calls for any product line. Since the agents will not be experts across all products, this is expected to increase the time to process a call by about 5%. The question that SMP has asked you to answer is how many cross-trained agents are needed to provide service at the same level as the current system.

    Incoming calls can be modeled as a Poisson arrival process with a rate of 60 per hour. The mean time required for an agent to answer a question

is 5 minutes, with the actual time being Erlang-2 for financial calls, and Erlang-3 for contact management calls. The current assignment of agents is 4 for financial and 3 for contact management. Simulate the system to find out how many agents are needed to deliver the same level of service in the cross-trained system as in the current system.

Bibliography

[1] SimPy Development Team, *SimPy Simulation Package*, http://simpy.sourceforge.net/, 2012.

[2] Norm Matloff, *A Discrete-Event Simulation Course Based on the SimPy Language*, http://heather.cs.ucdavis.edu/∼matloff/simcourse.html, 2011.

[3] Eric Jones, Travis Oliphant, Pearu Peterson, and others, *SciPy: Open Source Scientific Tools for Python*, http://www.scipy.org/, 2001.