The Bank

Release 5.5.0

G A Vignaux

March 10, 2015

CONTENTS

CHAPTER

ONE

THE BANK: EXAMPLES OF SIMPY SIMULATION

1.1 Introduction

SimPy is used to develop a simple simulation of a bank with a number of tellers. This Python package provides *Processes* to model active components such as messages, customers, trucks, and planes. It has three classes to model facilities where congestion might occur: *Resources* for ordinary queues, *Levels* for the supply of quantities of material, and *Stores* for collections of individual items. Only examples of *Resources* are described here. It also provides *Monitors* and *Tallys* to record data like queue lengths and delay times and to calculate simple averages. It uses the standard Python random package to generate random numbers.

Starting with SimPy 2.0 an object-oriented programmer's interface was added to the package. It is quite compatible with the current procedural approach which is used in the models described here.

SimPy can be obtained from: http://sourceforge.net/projects/simpy. The examples run with SimPy version 1.5 and later. This tutorial is best read with the SimPy Manual or Cheatsheet at your side for reference.

Before attempting to use SimPy you should be familiar with the Python language. In particular you should be able to use *classes*. Python is free and available for most machine types. You can find out more about it at the Python web site. SimPy is compatible with Python version 2.3 and later.

1.2 A single Customer

In this tutorial we model a simple bank with customers arriving at random. We develop the model step-by-step, starting out simply, and producing a running program at each stage. The programs we develop are available without line numbers and ready to go, in the `bankprograms` directory. Please copy them, run them and improve them - and in the tradition of open-source software suggest your modifications to the SimPy users list. Object-orented versions of all the models are included in the same directory.

A simulation should always be developed to answer a specific question; in these models we investigate how changing the number of bank servers or tellers might affect the waiting time for customers.

1.2.1 A Customer arriving at a fixed time

We first model a single customer who arrives at the bank for a visit, looks around at the decor for a time and then leaves. There is no queueing. First we will assume his arrival time and the time he spends in the bank are fixed.

We define a `Customer` class derived from the SimPy `Process` class. We create a `Customer` object, `c` who arrives at the bank at simulation time `5.0` and leaves after a fixed time of `10.0` minutes.

Examine the following listing which is a complete runnable Python script, except for the line numbers. We use comments to divide the script up into sections. This makes for clarity later when the programs get more complicated. Line 1 is a normal Python documentation string; line 2 imports the SimPy simulation code.

The `Customer` class definition, lines 6-12, defines our customer class and has the required generator method (called `visit`) (line 9) having a `yield` statement (line 11). Such a method is called a Process Execution Method (PEM) in SimPy.

The customer's `visit` PEM, lines 9-12, models his activities. When he arrives (it will turn out to be a 'he' in this model), he will print out the simulation time, `now()`, and his name (line 10). The function `now()` can be used at any time in the simulation to find the current simulation time though it cannot be changed by the programmer. The customer's name will be set when the customer is created later in the script (line 22).

He then stays in the bank for a fixed simulation time `timeInBank` (line 11). This is achieved by the `yield hold,self,timeInBank` statement. This is the first of the special simulation commands that `SimPy` offers.

After a simulation time of `timeInBank`, the program's execution returns to the line after the `yield` statement, line 12. The customer then prints out the current simulation time and his name. This completes the declaration of the `Customer` class.

Line 21 calls `initialize()` which sets up the simulation system ready to receive `activate` calls. In line 22, we create a customer, `c`, with name `Klaus`. All SimPy Processes have a `name` attribute. We `activate Klaus` in line 23 specifying the object (`c`) to be activated, the call of the action routine (`c.visit(timeInBank = 10.0)`) and that it is to be activated at time 5 (`at = 5.0`). This will activate `Klaus` exactly `5` minutes after the current time, in this case after the start of the simulation at `0.0`. The call of an action routine such as `c.visit` can specify the values of arguments, here the `timeInBank`.

Finally the call of `simulate(until=maxTime)` in line 24 will start the simulation. This will run until the simulation time is `maxTime` unless stopped beforehand either by the `stopSimulation()` command or by running out of events to execute (as will happen here). `maxTime` was set to `100.0` in line 16.

```
""" bank01: The single non-random Customer """
from SimPy.Simulation import *

## Model components ------------------------------

class Customer(Process):
    """ Customer arrives, looks around and leaves """

    def visit(self,timeInBank):
        print now(),self.name," Here I am"
        yield hold,self,timeInBank
        print now(),self.name," I must leave"

## Experiment data -------------------------------

maxTime = 100.0     # minutes
timeInBank = 10.0   # minutes

## Model/Experiment ------------------------------

initialize()
c = Customer(name="Klaus")
activate(c,c.visit(timeInBank),at=5.0)
simulate(until=maxTime)
```

The short trace printed out by the print statements shows the result. The program finishes at simulation time 15.0 because there are no further events to be executed. At the end of the visit routine, the customer has no more actions and no other objects or customers are active.

```
5.0 Klaus  Here I am
15.0 Klaus  I must leave
```

1.2.2 A Customer arriving at random

Now we extend the model to allow our customer to arrive at a random simulated time though we will keep the time in the bank at 10.0, as before.

The change occurs in line 3 of the program and in lines 22, 25, and 26. In line 3 we import from the standard Python `random` module to give us `expovariate` to generate the random time of arrival. We also import the `seed` function to initialize the random number stream to allow control of the random numbers. In line 22 we provide an initial seed of `99999`. An exponential random variate, `t`, is generated in line 25. Note that the Python Random module's `expovariate` function uses the rate (that is, `1.0/mean`) as the argument. The generated random variate, `t`, is used in Line 26 as the `at` argument to the `activate` call.

```
1 """ bank05: The single Random Customer """
2 from SimPy.Simulation import *
3 from random import expovariate, seed
4
5 ## Model components ------------------------
```

```python

class Customer(Process):
    """ Customer arrives at a random time,
        looks around and then leaves """

    def visit(self,timeInBank):
        print now(), self.name," Here I am"
        yield hold,self,timeInBank
        print now(), self.name," I must leave"

## Experiment data -------------------------

maxTime = 100.0    # minutes
timeInBank = 10.0
## Model/Experiment ------------------------------

seed(99999)
initialize()
c = Customer(name = "Klaus")
t = expovariate(1.0/5.0)
activate(c,c.visit(timeInBank),at=t)
simulate(until=maxTime)
```

The result is shown below. The customer now arrives at time 0.641954430556. Changing the seed value would change that time.

```
0.641954430556 Klaus Here I am
10.6419544306 Klaus   I must leave
```

1.3 More Customers

Our simulation does little so far. To consider a simulation with several customers we return to the simple deterministic model and add more `Customers`.

The program is almost as easy as the first example (A Customer arriving at a fixed time). The main change is in lines 22-27 where we create, name, and activate three customers. We also increase the maximum simulation time to `400` (line 16 and referred to in line 29). Observe that we need only one definition of the `Customer` class and create several objects of that class. These will act quite independently in this model.

Each customer stays for a different `timeinbank` so, instead of setting a common value for this we set it for each customer. The customers are started at different times (using `at=`). `Tony's` activation time occurs before `Klaus's`, so `Tony` will arrive first even though his activation statement appears later in the script.

As promised, the print statements have been changed to use Python string formatting (lines 10 and 12). The statements look complicated but the output is much nicer.

```python
1 """ bank02: More Customers """
2 from SimPy.Simulation import *
```

```

## Model components ------------------------

class Customer(Process):
    """ Customer arrives, looks around and leaves """

    def visit(self,timeInBank):
        print "%7.4f %s: Here I am"%(now(),self.name)
        yield hold,self,timeInBank
        print "%7.4f %s: I must leave"%(now(),self.name)

## Experiment data -------------------------

maxTime = 400.0  # minutes

## Model/Experiment ------------------------------

initialize()

c1 = Customer(name="Klaus")
activate(c1,c1.visit(timeInBank=10.0),at=5.0)
c2 = Customer(name="Tony")
activate(c2,c2.visit(timeInBank=7.0),at=2.0)
c3 = Customer(name="Evelyn")
activate(c3,c3.visit(timeInBank=20.0),at=12.0)

simulate(until=maxTime)
```

The trace produced by the program is shown below. Again the simulation finishes before the `400.0` specified in the `simulate` call.

```
 2.0000 Tony: Here I am
 5.0000 Klaus: Here I am
 9.0000 Tony: I must leave
12.0000 Evelyn: Here I am
15.0000 Klaus: I must leave
32.0000 Evelyn: I must leave
```

1.3.1 Many Customers

Another change will allow us to have more customers. As it is tedious to give a specially chosen name to each one, we will call them `Customer00, Customer01,...` and use a separate `Source` class to create and activate them. To make things clearer we do not use the random numbers in this model.

The following listing shows the new program. Lines 6-13 define a `Source` class. Its PEM, here called `generate`, is defined in lines 9-13. This PEM has a couple of arguments: the `number` of customers to be generated and the Time Between Arrivals, `TBA`. It consists of a loop that creates a sequence of numbered `Customers` from `0` to `(number-1)`, inclusive. We create a customer and give it a name in line 11. It is then activated at the current simulation time (the final argument of the `activate` statement is missing so that the default value of `now()` is used as the time). We also specify how long the customer is to stay in the bank. To keep it simple, all customers stay exactly `12` minutes. After each new customer is activated, the `Source` holds for a fixed time (`yield hold,self,TBA`) before creating the next one (line 13).

A `Source`, `s`, is created in line 32 and activated at line 33 where the number of customers to be generated is set to `maxNumber = 5` and the interval between customers to `ARRint = 10.0`. Once started at time `0.0` it creates customers at intervals and each customer then operates independently of the others:

```
1  """ bank03: Many non-random Customers """
2  from SimPy.Simulation import *
3  
4  ## Model components ------------------------
5  
6  class Source(Process):
7      """ Source generates customers regularly """
8  
9      def generate(self,number,TBA):
10         for i in range(number):
11             c = Customer(name = "Customer%02d"%(i,))
12             activate(c,c.visit(timeInBank=12.0))
13             yield hold,self,TBA
14 
15 class Customer(Process):
16     """ Customer arrives, looks around and leaves """
17 
18     def visit(self,timeInBank):
19         print "%7.4f %s: Here I am"%(now(),self.name)
20         yield hold,self,timeInBank
21         print "%7.4f %s: I must leave"%(now(),self.name)
22 
23 ## Experiment data -------------------------
24 
25 maxNumber = 5
26 maxTime = 400.0 # minutes
27 ARRint = 10.0   # time between arrivals, minutes
28 
29 ## Model/Experiment ------------------------------
30 
31 initialize()
32 s = Source()
33 activate(s,s.generate(number=maxNumber,
34                       TBA=ARRint),at=0.0)
35 simulate(until=maxTime)
```

The output is:

```
 0.0000 Customer00: Here I am
10.0000 Customer01: Here I am
12.0000 Customer00: I must leave
```

```
20.0000 Customer02: Here I am
22.0000 Customer01: I must leave
30.0000 Customer03: Here I am
32.0000 Customer02: I must leave
40.0000 Customer04: Here I am
42.0000 Customer03: I must leave
52.0000 Customer04: I must leave
```

1.3.2 Many Random Customers

We now extend this model to allow arrivals at random. In simulation this is usually interpreted as meaning that the times between customer arrivals are distributed as exponential random variates. There is little change in our program, we use a `Source` object, as before.

The exponential random variate is generated in line 14 with `meanTBA` as the mean Time Between Arrivals and used in line 15. Note that this parameter is not exactly intuitive. As already mentioned, the Python `expovariate` method uses the *rate* of arrivals as the parameter not the average interval between them. The exponential delay between two arrivals gives pseudo-random arrivals. In this model the first customer arrives at time `0.0`.

The `seed` method is called to initialize the random number stream in the `model` routine (line 33). It is possible to leave this call out but if we wish to do serious comparisons of systems, we must have control over the random variates and therefore control over the seeds. Then we can run identical models with different seeds or different models with identical seeds. We provide the seeds as control parameters of the run. Here a seed is assigned in line 33 but it is clear it could have been read in or manually entered on an input form.

```
1  """ bank06: Many Random Customers """
2  from SimPy.Simulation import *
3  from random import expovariate,seed
4
5  ## Model components ------------------------
6
7  class Source(Process):
8      """ Source generates customers at random """
9
10     def generate(self,number,meanTBA):
11         for i in range(number):
12             c = Customer(name = "Customer%02d"%(i,))
13             activate(c,c.visit(timeInBank=12.0))
14             t = expovariate(1.0/meanTBA)
15             yield hold,self,t
16
17 class Customer(Process):
18     """ Customer arrives, looks around and leaves """
19
20     def visit(self,timeInBank=0):
21         print "%7.4f %s: Here I am"%(now(),self.name)
22         yield hold,self,timeInBank
23         print "%7.4f %s: I must leave"%(now(),self.name)
```

```

## Experiment data -------------------------

maxNumber = 5
maxTime = 400.0 # minutes
ARRint = 10.0   # mean arrival interval, minutes

## Model/Experiment ------------------------------

seed(99999)
initialize()
s = Source(name='Source')
activate(s,s.generate(number=maxNumber,
                      meanTBA=ARRint),at=0.0)
simulate(until=maxTime)
```

with the following output:

```
 0.0000 Customer00: Here I am
 1.2839 Customer01: Here I am
 4.9842 Customer02: Here I am
12.0000 Customer00: I must leave
13.2839 Customer01: I must leave
16.9842 Customer02: I must leave
35.5432 Customer03: Here I am
47.5432 Customer03: I must leave
48.9918 Customer04: Here I am
60.9918 Customer04: I must leave
```

1.4 A Service counter

So far, the model has been more like an art gallery, the customers entering, looking around, and leaving. Now they are going to require service from the bank clerk. We extend the model to include a service counter which will be modelled as an object of SimPy's `Resource` class with a single resource unit. The actions of a `Resource` are simple: a customer `requests` a unit of the resource (a clerk). If one is free he gets service (and removes the unit). If there is no free clerk the customer joins the queue (managed by the resource object) until it is their turn to be served. As each customer completes service and `releases` the unit, the clerk can start serving the next in line.

1.4.1 One Service counter

The service counter is created as a `Resource` (`k`) in line 38. This is provided as an argument to the `Source` (line 45) which, in turn, provides it to each customer it creates and activates (line 14).

The actions involving the service counter, `k`, in the customer's PEM are:

- the `yield request` statement in line 25. If the server is free then the customer can start service immediately and the code moves on to line 26. If the server is busy, the customer is automatically queued by the Resource. When it eventually comes available the PEM moves on to line 26.
- the `yield hold` statement in line 28 where the operation of the service counter is modelled. Here the service time is a fixed `timeInBank`. During this period the customer is being served.
- the `yield release` statement in line 29. The current customer completes service and the service counter becomes available for any remaining customers in the queue.

Observe that the service counter is used with the pattern (`yield request..; yield hold..; yield release..`).

To show the effect of the service counter on the activities of the customers, I have added line 22 to record when the customer arrived and line 26 to record the time between arrival in the bank and starting service. Line 26 is *after* the `yield request` command and will be reached only when the request is satisfied. It is *before* the `yield hold` that corresponds to the start of service. The variable `wait` will record how long the customer waited and will be 0 if he received service at once. This technique of saving the arrival time in a variable is common. So the `print` statement also prints out how long the customer waited in the bank before starting service.

```
1  """ bank07: One Counter,random arrivals """
2  from SimPy.Simulation import *
3  from random import expovariate, seed
4
5  ## Model components ------------------------
6
7  class Source(Process):
8      """ Source generates customers randomly """
9
10     def generate(self,number,meanTBA,resource):
11         for i in range(number):
12             c = Customer(name = "Customer%02d"%(i,))
13             activate(c,c.visit(timeInBank=12.0,
14                                res=resource))
15             t = expovariate(1.0/meanTBA)
16             yield hold,self,t
17
18 class Customer(Process):
19     """ Customer arrives, is served and  leaves """
20
21     def visit(self,timeInBank,res):
22         arrive = now()         # arrival time
23         print "%8.3f %s: Here I am     "%(now(),self.name)
24
25         yield request,self,res
26         wait = now()-arrive    # waiting time
27         print "%8.3f %s: Waited %6.3f"%(now(),self.name,wait)
28         yield hold,self,timeInBank
```

```
29          yield release,self,res
30
31          print "%8.3f %s: Finished      "%(now(),self.name)
32
33  ## Experiment data -------------------------
34
35  maxNumber = 5
36  maxTime = 400.0   # minutes
37  ARRint = 10.0     # mean, minutes
38  k = Resource(name="Counter",unitName="Clerk")
39
40  ## Model/Experiment ------------------------------
41  seed(99999)
42  initialize()
43  s = Source('Source')
44  activate(s,s.generate(number=maxNumber,
45                        meanTBA=ARRint, resource=k),at=0.0)
46  simulate(until=maxTime)
```

Examining the trace we see that the first two customers get instant service but the others have to wait. We still only have five customers (line 35) so we cannot draw general conclusions.

```
 0.000 Customer00: Here I am
 0.000 Customer00: Waited  0.000
 1.284 Customer01: Here I am
 4.984 Customer02: Here I am
12.000 Customer00: Finished
12.000 Customer01: Waited 10.716
24.000 Customer01: Finished
24.000 Customer02: Waited 19.016
35.543 Customer03: Here I am
36.000 Customer02: Finished
36.000 Customer03: Waited  0.457
48.000 Customer03: Finished
48.992 Customer04: Here I am
48.992 Customer04: Waited  0.000
60.992 Customer04: Finished
```

1.4.2 A server with a random service time

This is a simple change to the model in that we retain the single service counter but make the customer service time a random variable. As is traditional in the study of simple queues we first assume an exponential service time and set the mean to `timeInBank`.

The service time random variable, `tib`, is generated in line 26 and used in line 27. The argument to be used in the call of `expovariate` is not the mean of the distribution, `timeInBank`, but is the rate `1/timeInBank`.

We have also collected together a number of constants by defining a number of appropriate variables and giving them values. These are in lines 31 to 42.

```python
""" bank08: A counter with a random service time """
from SimPy.Simulation import *
from random import expovariate, seed

## Model components ------------------------

class Source(Process):
    """ Source generates customers randomly """

    def generate(self,number,meanTBA,resource):
        for i in range(number):
            c = Customer(name = "Customer%02d"%(i,))
            activate(c,c.visit(b=resource))
            t = expovariate(1.0/meanTBA)
            yield hold,self,t

class Customer(Process):
    """ Customer arrives, is served and leaves """

    def visit(self,b):
        arrive = now()
        print "%8.4f %s: Here I am     "%(now(),self.name)
        yield request,self,b
        wait = now()-arrive
        print "%8.4f %s: Waited %6.3f"%(now(),self.name,wait)
        tib = expovariate(1.0/timeInBank)
        yield hold,self,tib
        yield release,self,b
        print "%8.4f %s: Finished      "%(now(),self.name)

## Experiment data -------------------------

maxNumber = 5
maxTime = 400.0 # minutes
timeInBank=12.0 # mean, minutes
ARRint = 10.0   # mean, minutes
theseed= 12345

## Model/Experiment ------------------------------

seed(theseed)
k = Resource(name="Counter",unitName="Clerk")

initialize()
s = Source('Source')
activate(s,s.generate(number=maxNumber,meanTBA=ARRint,
                      resource=k),at=0.0)
simulate(until=maxTime)
```

And the output:

```
 0.0000 Customer00: Here I am
 0.0000 Customer00: Waited  0.000
 0.1227 Customer00: Finished
 5.3892 Customer01: Here I am
 5.3892 Customer01: Waited  0.000
 9.6460 Customer01: Finished
22.8307 Customer02: Here I am
22.8307 Customer02: Waited  0.000
25.4137 Customer02: Finished
27.4258 Customer03: Here I am
27.4258 Customer03: Waited  0.000
29.5422 Customer03: Finished
35.7731 Customer04: Here I am
35.7731 Customer04: Waited  0.000
42.5805 Customer04: Finished
```

This model with random arrivals and exponential service times is an example of an M/M/1 queue and could rather easily be solved analytically to calculate the steady-state mean waiting time and other operating characteristics. (But not so easily solved for its transient behavior.)

1.5 Several Service Counters

When we introduce several counters we must decide on a queue discipline. Are customers going to make one queue or are they going to form separate queues in front of each counter? Then there are complications - will they be allowed to switch lines (jockey)? We first consider a single queue with several counters and later consider separate isolated queues. We will not look at jockeying.

1.5.1 Several Counters but a Single Queue

Here we model a bank whose customers arrive randomly and are to be served at a group of counters, taking a random time for service, where we assume that waiting customers form a single first-in first-out queue.

The *only* difference between this model and the single-server model is in line 42. We have provided two counters by increasing the capacity of the `counter` resource to 2. These *units* of the resource correspond to the two counters. Because both clerks cannot be called `Karen`, we have used a general name of `Clerk`.

```
1 """ bank09: Several Counters but a Single Queue """
2 from SimPy.Simulation import *
3 from random import expovariate, seed
4
5 ## Model components ------------------------
6
7 class Source(Process):
8     """ Source generates customers randomly """
9
```

```
    def generate(self,number,meanTBA,resource):
        for i in range(number):
            c = Customer(name = "Customer%02d"%(i,))
            activate(c,c.visit(b=resource))
            t = expovariate(1.0/meanTBA)
            yield hold,self,t

class Customer(Process):
    """ Customer arrives, is served and leaves """

    def visit(self,b):
        arrive = now()
        print "%8.4f %s: Here I am     "%(now(),self.name)
        yield request,self,b
        wait = now()-arrive
        print "%8.4f %s: Waited %6.3f"%(now(),self.name,wait)
        tib = expovariate(1.0/timeInBank)
        yield hold,self,tib
        yield release,self,b
        print "%8.4f %s: Finished      "%(now(),self.name)

## Experiment data -------------------------

maxNumber = 5
maxTime = 400.0 # minutes
timeInBank = 12.0 # mean, minutes
ARRint = 10.0   # mean, minutes
theseed = 999999

## Model/Experiment ------------------------------

seed(theseed)
k = Resource(capacity=2,name="Counter",unitName="Clerk")

initialize()
s = Source('Source')
activate(s, s.generate(number=maxNumber,meanTBA=ARRint,
                         resource=k),at=0.0)
simulate(until=maxTime)
```

The waiting times in this model are much shorter than those for the single service counter. For example, the waiting time for `Customer02` has been reduced from `51.213` to `12.581` minutes. Again we have too few customers processed to draw general conclusions.

```
 0.0000 Customer00: Here I am
 0.0000 Customer00: Waited  0.000
 2.3596 Customer01: Here I am
 2.3596 Customer01: Waited  0.000
12.2925 Customer02: Here I am
14.6555 Customer03: Here I am
15.1717 Customer00: Finished
```

```
15.1717 Customer02: Waited  2.879
23.2991 Customer01: Finished
23.2991 Customer03: Waited  8.644
24.6826 Customer02: Finished
33.7110 Customer04: Here I am
33.7110 Customer04: Waited  0.000
40.0949 Customer03: Finished
58.4710 Customer04: Finished
```

1.5.2 Several Counters with individual queues

Each counter is now assumed to have its own queue. The programming is more complicated because the customer has to decide which queue to join. The obvious technique is to make each counter a separate resource and it is useful to make a list of resource objects (line 56).

In practice, a customer will join the shortest queue. So we define the Python function, `NoInSystem(R)` (lines 17-19) which returns the sum of the number waiting and the number being served for a particular counter, `R`. This function is used in line 28 to list the numbers at each counter. It is then easy to find which counter the arriving customer should join. We have also modified the trace printout, line 29 to display the state of the system when the customer arrives. We choose the shortest queue in lines 30-32 (the variable `choice`).

The rest of the program is the same as before.

```
1  """ bank10: Several Counters with individual queues"""
2  from SimPy.Simulation import *
3  from random import expovariate,seed
4
5  ## Model components ------------------------
6
7  class Source(Process):
8      """ Source generates customers randomly"""
9
10     def generate(self,number,interval,counters):
11         for i in range(number):
12             c = Customer(name = "Customer%02d"%(i,))
13             activate(c,c.visit(counters))
14             t = expovariate(1.0/interval)
15             yield hold,self,t
16
17 def NoInSystem(R):
18     """ Total number of customers in the resource R"""
19     return (len(R.waitQ)+len(R.activeQ))
20
21 class Customer(Process):
22     """ Customer arrives, chooses the shortest queue
23         is served and leaves
24     """
25
```

```
    def visit(self,counters):
        arrive = now()
        Qlength = [NoInSystem(counters[i]) for i in range(Nc)]
        print "%7.4f %s: Here I am. %s"%(now(),self.name,Qlength)
        for i in range(Nc):
            if Qlength[i] == 0 or Qlength[i] == min(Qlength):
                choice = i  # the chosen queue number
                break

        yield request,self,counters[choice]
        wait = now()-arrive
        print "%7.4f %s: Waited %6.3f"%(now(),self.name,wait)
        tib = expovariate(1.0/timeInBank)
        yield hold,self,tib
        yield release,self,counters[choice]

        print "%7.4f %s: Finished"%(now(),self.name)

## Experiment data -------------------------

maxNumber = 5
maxTime = 400.0 # minutes
timeInBank = 12.0 # mean, minutes
ARRint = 10.0   # mean, minutes
Nc = 2          # number of counters
theseed = 9191

## Model/Experiment ------------------------------

seed(theseed)
kk = [Resource(name="Clerk0"),Resource(name="Clerk1")]
initialize()
s = Source('Source')
activate(s,s.generate(number=maxNumber,interval=ARRint,
                      counters=kk),at=0.0)
simulate(until=maxTime)
```

The results show how the customers choose the counter with the smallest number. `Customer02` has to wait until `Customer00` finishes at time `15.1717`. There are, however, too few arrivals in these runs, limited as they are to five customers, to draw any general conclusions about the relative efficiencies of the two systems.

```
 0.0000 Customer00: Here I am. (0, 0)
 0.0000 Customer00: Waited  0.000
 3.3207 Customer01: Here I am. (1, 0)
 3.3207 Customer01: Waited  0.000
 6.6460 Customer02: Here I am. (1, 1)
12.1423 Customer03: Here I am. (2, 1)
14.4842 Customer01: Finished
14.4842 Customer03: Waited  2.342
15.0046 Customer04: Here I am. (2, 1)
```

```
17.8774 Customer00: Finished
17.8774 Customer02: Waited 11.231
40.0907 Customer02: Finished
53.5888 Customer03: Finished
53.5888 Customer04: Waited 38.584
57.2187 Customer04: Finished
```

1.6 Monitors and Gathering Statistics

The traces of output that have been displayed so far are valuable for checking that the simulation is operating correctly but would become too much if we simulate a whole day. We do need to get results from our simulation to answer the original questions. What, then, is the best way to summarize the results?

One way is to analyze the traces elsewhere, piping the trace output, or a modified version of it, into a *real* statistical program such as *R* for statistical analysis, or into a file for later examination by a spreadsheet. We do not have space to examine this thoroughly here. Another way of presenting the results is to provide graphical output.

SimPy offers an easy way to gather a few simple statistics such as averages: the `Monitor` and `Tally` classes. The `Monitor` records the values of chosen variables as time series (but see the comments in Final Remarks).

1.6.1 The Bank with a Monitor

We now demonstrate a `Monitor` that records the average waiting times for our customers. We return to the system with random arrivals, random service times and a single queue and remove the old trace statements. In practice, we would make the printouts controlled by a variable, say, `TRACE` which is set in the experimental data (or read in as a program option - but that is a different story). This would aid in debugging and would not complicate the data analysis. We will run the simulations for many more arrivals.

A Monitor, `wM`, is created in line 42. It `observes` the waiting time mentioned in line 24. We run `maxNumber=50` customers (in the call of `generate` in line 45) and have increased `maxTime` to `1000` minutes.

```
1  """ bank11: The bank with a Monitor"""
2  from SimPy.Simulation import *
3  from random import expovariate,seed
4
5  ## Model components ------------------------
6
7  class Source(Process):
8      """ Source generates customers randomly"""
9
10     def generate(self,number,interval,resource):
11         for i in range(number):
12             c = Customer(name = "Customer%02d"%(i,))
```

```
            activate(c,c.visit(b=resource))
            t = expovariate(1.0/interval)
            yield hold,self,t

class Customer(Process):
    """ Customer arrives, is served and leaves """

    def visit(self,b):
        arrive = now()
        yield request,self,b
        wait = now()-arrive
        wM.observe(wait)
        tib = expovariate(1.0/timeInBank)
        yield hold,self,tib
        yield release,self,b

## Experiment data -------------------------

maxNumber = 50
maxTime = 1000.0 # minutes
timeInBank = 12.0  # mean, minutes
ARRint = 10.0    # mean, minutes
Nc = 2           # number of counters
theseed = 12345

## Model/Experiment   ----------------------

seed(theseed)
k = Resource(capacity=Nc,name="Clerk")
wM = Monitor()
initialize()
s = Source('Source')
activate(s,s.generate(number=maxNumber,interval=ARRint,
                      resource=k),at=0.0)
simulate(until=maxTime)

## Result  ----------------------------------

result = wM.count(),wM.mean()
print "Average wait for %3d completions was %5.3f minutes."% result
```

The average waiting time for 50 customers in this 2-counter system is more reliable (i.e., less subject to random simulation effects) than the times we measured before but it is still not sufficiently reliable for real-world decisions. We should also replicate the runs using different random number seeds. The result of this run is:

```
Average wait for  50 completions was 10.312 minutes.
```

1.6.2 Multiple runs

To get a number of independent measurements we must replicate the runs using different random number seeds. Each replication must be independent of previous ones so the Monitor and Resources must be redefined for each run. We can no longer allow them to be global objects as we have before.

We will define a function, `model` with a parameter `runSeed` so that the random number seed can be different for different runs (lines 40-50). The contents of the function are the same as the `Model/Experiment` section in the previous program except for one vital change.

This is required since the Monitor, `wM`, is defined inside the `model` function (line 43). A customer can no longer refer to it. In the spirit of quality computer programming we will pass `wM` as a function argument. Unfortunately we have to do this in two steps, first to the `Source` (line 48) and then from the `Source` to the `Customer` (line 13).

`model()` is run for four different random-number seeds to get a set of replications (lines 54-57).

```
1  """ bank12: Multiple runs of the bank with a Monitor"""
2  from SimPy.Simulation import *
3  from random import expovariate,seed
4  
5  ## Model components ------------------------
6  
7  class Source(Process):
8      """ Source generates customers randomly"""
9  
10     def generate(self,number,interval,resource,mon):
11         for i in range(number):
12             c = Customer(name = "Customer%02d"%(i,))
13             activate(c,c.visit(b=resource,M=mon))
14             t = expovariate(1.0/interval)
15             yield hold,self,t
16 
17 class Customer(Process):
18     """ Customer arrives, is served and leaves """
19 
20     def visit(self,b,M):
21         arrive = now()
22         yield request,self,b
23         wait = now()-arrive
24         M.observe(wait)
25         tib = expovariate(1.0/timeInBank)
26         yield hold,self,tib
27         yield release,self,b
28 
29 ## Experiment data -------------------------
30 
31 maxNumber = 50
32 maxTime = 2000.0  # minutes
33 timeInBank = 12.0   # mean, minutes
```

```
ARRint = 10.0      # mean, minutes
Nc = 2             # number of counters
theSeed = 393939

## Model  ----------------------------------

def model(runSeed=theSeed):
    seed(runSeed)
    k = Resource(capacity=Nc,name="Clerk")
    wM = Monitor()

    initialize()
    s = Source('Source')
    activate(s,s.generate(number=maxNumber,interval=ARRint,
                          resource=k,mon=wM),at=0.0)
    simulate(until=maxTime)
    return (wM.count(),wM.mean())

## Experiment/Result  ----------------------------------

theseeds = [393939,31555999,777999555,319999771]
for Sd in theseeds:
    result = model(Sd)
    print "Average wait for %3d completions was %6.2f minutes."% result
```

The results show some variation. Remember, though, that the system is still only operating for 50 customers so the system may not be in steady-state.

```
Average wait for  50 completions was   3.66 minutes.
Average wait for  50 completions was   2.62 minutes.
Average wait for  50 completions was   8.97 minutes.
Average wait for  50 completions was   5.34 minutes.
```

1.7 Final Remarks

This introduction is too long and the examples are getting longer. There is much more to say about simulation with *SimPy* but no space. I finish with a list of topics for further study:

- **GUI input**. Graphical input of simulation parameters could be an advantage in some cases. *SimPy* allows this and programs using these facilities have been developed (see, for example, program `MM1.py` in the examples in the *SimPy* distribution)
- **Graphical Output**. Similarly, graphical output of results can also be of value, not least in debugging simulation programs and checking for steady-state conditions. SimPlot is useful here.
- **Statistical Output**. The `Monitor` class is useful in presenting results but more powerful methods of analysis are often needed. One solution is to output a trace and read that into a large-scale statistical system such as *R*.

- **Priorities and Reneging in queues**. *SimPy* allows processes to request units of resources under a priority queue discipline (preemptive or not). It also allows processes to renege from a queue.
- **Other forms of Resource Facilities**. *SimPy* has two other resource structures: `Levels` to hold bulk commodities, and `Stores` to contain an inventory of different object types.
- **Advanced synchronization/scheduling commands**. *SimPy* allows process synchronization by events and signals.

1.8 Acknowledgements

I thank Klaus Muller, Bob Helmbold, Mukhlis Matti and other developers and users of SimPy for improving this document by sending their comments. I would be grateful for further suggestions or corrections. Please send them to: *vignaux* at *users.sourceforge.net*.

1.9 References

- Python website: http://www.Python.org
- SimPy website: http://sourceforge.net/projects/simpy