Introduction to Simulation

Norm Matloff

January 8, 2008

Contents

1 Cons3.py: A Simple Example to Get Started

Here we present our first simulation program. It will be written in Python, but don't worry if you don't have any background in Python, as it is easy to read and you'll pick it up quickly.[1]

To get a feeling for the topic, let's look at a simple example. Suppose a coin is tossed until we get three consecutive heads. Let X be the number of tosses needed. Let's find P(X > 6) and E(X).

```
# Cons3.py, Python simulation example:  keep tossing a coin until get 3
# heads in a row; let X be the number of tosses needed; find P(X > 6),
# E(X)

# usage:  python Cons3.py

import random  # load the library

r = random.Random(98765)  # sets random number generator seed
sumx = 0  # sum of all X values
count = 0  # number of times X > 6
for rep in range(10000):  # rep = 0,1,...,9999
```

[1]See my Python tutorial for details, at `http://heather.cs.ucdavis.edu/~matloff/python.html`.

```
13    x = 0
14    consechds = 0
15    while True:
16       u = r.uniform(0.0,1.0)  # generate random number in (0,1)
17       if u < 0.5:  # heads
18          consechds += 1
19       else:  # tails
20          consechds = 0
21       x += 1
22       if consechds == 3: break
23    if x > 6: count += 1
24    sumx += x
25 print 'probability more than 6 tosses are needed =', count/10000.0
26 print 'mean number of tosses to get 3 consecutive heads =', sumx/10000.0
```

I regard the key to understanding here to be the notion of a "repeatable experiment." Here the experiment consists of tossing the coin until we get three heads in a row. Our Python **for** loop here, in lines 13-24, we are repeating the experiment 10000 times; lines 13-24 do the experiment once.[2]

Since E(X) is the long-run average of X—overly infinitely many repetitions of the experiment—we approximate it by the short-run average, which is the average of X over those 10000 repetitions. That is done in line 24.

A probability is the long-run proportion of time an event occurs. Here the event $X > 6$ occurs on some of those 10000 repetitions, and our approximate value of $P(X > 6)$ is the proportion of those repetitions in which the event occurs. That is computed in line 23.

Now let's see how the experiment is simulated. Look at line 16. Here we are calling a library function, **uniform()**, which generates uniformly-distributed random numbers in [0,1).[3] The way we simulate the tossing of the coin is to say the coin came up heads if the random uniform variate comes out smaller than 0.5, and say it's tails otherwise. This is done in lines 17-22.

A bit more on the function **uniform**: It's part of a Python library class **random**. In line 7, we tell Python we wish to bring in that library for our use. It does require initializing what is known as a **seed** for random number generation, which we do in line 9. We'll find out in a later unit what this really is, but at this point it doesn't matter much which number we use. My choice of 98765 was pretty abitrary. What is important, though, is that an object is returned from the **random** class, which we assign to **r**. We'll always need to do things through that object, as you can see by the "r." in line 16.

One more point: We mentioned that the output of our program consists of short-run approximations to long-run averages and proportions. In this case, "short-run" was 10000 repetitions of the experiment. Is that enough? We will answer this question too in a later unit.

2 Bit More Realism

Let's now move away from the realm of coins to examplex a little more reflective of the types of problems people simulate in the real world.

We'll also make a bit more sophisticated use of Python, in particular Python classes. See my Python tutorial

[2]Note that Python defines blocks by indentation. The fact that lines 13-24 are indented tells the Python interpreter that we intend these lines as a block.

[3]We'll see in a later unit why the interval includes 0 but excludes 1. But it is not important. Recall that for continuous random variables, the probability of a particular point is 0 anyway.

at http://heather.cs.ucdavis.edu/~matloff/python.html if you are not familiar with these.

In our first case, our realm is the design of computer networks.

2.1 Aloha.py

Today's Ethernet evolved from an experimental network developed at the University of Hawaii, called ALOHA. A number of network nodes would occasionally try to use the same radio channel to communicate with a central computer. The nodes couldn't hear each other, due to the obstruction of mountains between them. If only one of them made an attempt to send, it would be successful, and it would receive an acknowledgement message in response from the central computer. But if more than one node were to transmit, a **collision** would occur, garbling all the messages. The sending nodes would timeout after waiting for an acknowledgement which never came, and try sending again later. To avoid having too many collisions, nodes would engage in **backoff**, meaning that they would refrain from sending for a while even though they had something to send.

In designing such a network, the key is to allow enough backoff to avoid having too many collisions, but not so much that a node often refrains from trying to send even though no other node tries.

One variation is **slotted** ALOHA, which divides time into discrete intervals which I will call "epochs." The simulation program below finds the probability that k nodes currently have messages to send at epoch m.

Note that in any simulation, you have to decide how much detail to put into your model, and which parameters to incorporate. Here we had two central parameters, one being a probability p that models the amount of backoff. In the version we will consider here, in each epoch, if a node is "active," i.e. has a message to send, it will either send or refrain from sending, with probability p and 1-p. (Real Ethernet hardware really does something like this, using a random number generator inside the chip.)

The other parameter q is the probability that a node which had been "inactive" generates a message during an epoch, and thus becomes "active." Assume for simplicity that this happens just before the time that all the active nodes decide whether to try to send, so that a newly-arrived message might be sent in the same epoch.

Be sure to keep in mind that in our simple model here, during the time a node is active, it won't generate any additional new messages.

Note that in real life q is basically imposed on us. We have a certain amount of traffic in our network, and we must deal with it. But we can control p, and indeed different values of q would require different values of p for best results.

Here is the program below. Some comments have been added concerning Python itself, to ease the reader's transition to that language.

```
# Aloha.py, Python simulation example:  a form of slotted ALOHA

# here we will look finite time, finding the probability that there are
# k active nodes at the end of epoch m

# usage:  python Aloha.py s p q m k

import random, sys

```

```
10 class node:  # one object of this class models one network node
11    # some class variables
12    s = int(sys.argv[1])  # number of nodes
13    p = float(sys.argv[2])  # transmit probability
14    q = float(sys.argv[3])  # msg creation probability
15    activeset = []  # which nodes are active now
16    inactiveset = []  # which nodes are inactive now
17    r = random.Random(98765)  # set seed
18    def __init__(self):  # object constructor
19       # start this node in inactive mode
20       node.inactiveset.append(self)
21    # class (i.e. not instance) methods
22    def checkgoactive():  # determine which nodes will go active
23       for n in node.inactiveset:
24          if node.r.uniform(0,1) < node.q:
25             node.inactiveset.remove(n)
26             node.activeset.append(n)
27    checkgoactive = staticmethod(checkgoactive)
28    def trysend():
29       numnodestried = 0  # number of nodes which have tried to send
30                          # during the current epoch
31       whotried = None  # which node tried to send (last)
32       # determine which nodes try to send
33       for n in node.activeset:
34          if node.r.uniform(0,1) < node.p:
35             whotried = n
36             numnodestried += 1
37       # we’ll have a successful transmission if and only if exactly one
38       # node has tried to send
39       if numnodestried == 1:
40          node.activeset.remove(whotried)
41          node.inactiveset.append(whotried)
42    trysend = staticmethod(trysend)
43    def reset():  # resets variables after a repetition of the experiment
44       for n in node.activeset:
45          node.activeset.remove(n)
46          node.inactiveset.append(n)
47    reset = staticmethod(reset)
48
49 def main():
50    m = int(sys.argv[4])
51    k = int(sys.argv[5])
52    # set up the s nodes
53    for i in range(node.s):  node()
54    count = 0
55    for rep in range(10000):
56       # run the process for m epochs
57       for epoch in range(m):
58          node.checkgoactive()
59          node.trysend()
60       # len() gives the length of an object
61       if len(node.activeset) == k: count += 1
62       node.reset()
63    print ’P(k active at time m) =’, count/10000.0
64
65 # technical device to make debugging easier, etc.
66 if __name__ == ’__main__’: main()
```

Python arrays, called **lists**, are wonderfully flexible. Among other things, they’re nice for modeling set membership, so I set up arrays **active** and **inactive**. Python’s **in** operator enables me to test for set membership, as seen for instance in line 23. There we have a **for** loop that says we will loop through each of the inactive nodes, and check to see whether they generate a new message and become active (line 24). In lines 25 and 26, we use the Python **list** types built-in methods **remove()** and **append()** to move this node, which has just become active, from the inactive to the active set.

In lines 33-36, we simulate each active node deciding whether to try to send or not, and in line 39 we check to see if only one such attempt was made. If so, that attempt succeeds, and we move that node to the inactive set (lines 40-41).

Note the function **reset()** in lines 43-46, which allows us to reinitialize between repetitions of the experiment. Failure to do so would give us wrong answers. **This is a common type of bug in simulation programming.**

2.2 Inv.py

```
# Inv.py, inventory simulation example

# each day during the daytime, a number of orders for widgets arrives,
# uniformly distributed on {0,1,2,...,k}; the number of widgets in an order
# is uniformly distributed on {1,2,...,m}; each evening, if the inventory has
# fallen below r during the day, the inventory is restocked to level s;
# inventory is initially s

# we will find E(N), where N is the number of days until the first
# instance of an unfilled order; N = 1,2,...

# usage:  python Inv.py r s k m

import random, sys

class g:  # global variables
   r = int(sys.argv[1])  # restocking signal level
   s = int(sys.argv[2])  # restocking replenishment level
   k = int(sys.argv[3])  # range for distr of number of orders
   m = int(sys.argv[4])  # range for distr of number of widgets per order
   tottimetobl = 0  # total wait time for those msgs
   inv = None  # inventory level
   rnd = random.Random(98765)  # set seed

def simdaytime():
   norders = g.rnd.randint(0,g.k)
   for o in range(norders):
      nwidgets = g.rnd.randint(1,g.m)
      g.inv -= nwidgets

def simevening():
   if g.inv < g.r: g.inv = g.s

def main():
   nreps = int(sys.argv[5])
   for rep in range(nreps):  # simulate nreps repetitions of the experiment
      day = 1
      g.inv = g.s  # inventory full at first
      while True:
         simdaytime()
         if g.inv < 0: break
         simevening()
         day += 1
      g.tottimetobl += day
      g.tottimetobl2 += day*day
   print 'mean time to get a backlog =', g.tottimetobl/float(nreps)

if __name__ == '__main__': main()
```

3 Moving to "Time Average" Models

Here, instead of the notion of a "repeatable" experiment, the situation here is that of a "continuing" experiment. In our ALOHA example above, we were interested in how things behave at a fixed epoch (m). But we could ask what happens further and further back in time. For example, we could ask, what is the long-run mean wait for messages to be sent successfully, with "long-run" meaning infinitely many epochs. For fixed q, what is the best value of p, i.e. what value of p minimizes the mean wait?

Implicit in such questions is the assumption that things converge to a limit. For example, let X_i denote the time the i^{th} message must wait to get through, and let N_m denote the number of messages that have been successfully transmitted by epoch m. Then the mean wait for messages through epoch m is

$$\bar{X}_m = \frac{X_1 + X_2 + ... + X_{N_m}}{N_m}$$

For many types of systems, one can show that

$$\lim_{m \rightarrow \infty} \bar{X}_m$$

exists and is equal to some constant c. That's why such a simulation is sometimes called a **steady-state** simulation (the term **ergodic** is also used), with the type we looked at earlier being referred to as a **terminating** simulation.

Our job is to use simulation to approximate c. Our programs below do exactly that.

3.1 Aloha2.py

```
# Aloha2.py, Python simulation example:  a form of slotted ALOHA

# goal is to study response time (mean number of attempts needed to send
# a message), as a function of p

# usage:  python Aloha2.py s p q

import random, sys

class node:  # models one network node
   # some class variables
   s = int(sys.argv[1])  # number of nodes
   p = float(sys.argv[2])  # transmit probability
   q = float(sys.argv[3])  # msg creation probability
   totsent = 0  # number of msgs sent successfully by all nodes so far
   totwait = 0  # total wait time for those msgs
   activeset = []  # which nodes are active now (empty list for now)
   inactiveset = []  # which nodes are inactive now
   r = random.Random(98765)  # set seed
   def __init__(self):  # object constructor
      # start this node in inactive mode
      node.inactiveset.append(self)
      # the number of epochs this node's current msg has been waiting:
      self.wait = None  # currently undefined
   # class methods
   def checkgoactive():  # determine which nodes will go active
      for n in node.inactiveset:  # n is local to this function
         if node.r.uniform(0,1) < node.q:
            node.inactiveset.remove(n)
```

```
30             node.activeset.append(n)
31             n.wait = 0
32    checkgoactive = staticmethod(checkgoactive)  # make this a class method
33    def trysend():
34       numnodestried = 0  # number of nodes which have tried to send
35                          # during the current epoch
36       whotried = None  # which node tried to send (last)
37       # determine which nodes try to send
38       for n in node.activeset:
39          n.wait += 1
40          if node.r.uniform(0,1) < node.p:
41             whotried = n
42             numnodestried += 1
43       if numnodestried == 1:
44          node.totsent += 1
45          node.totwait += whotried.wait
46          node.activeset.remove(whotried)
47          node.inactiveset.append(whotried)
48    trysend = staticmethod(trysend)
49
50 def main():
51    for i in range(node.s):  node()
52    for rep in range(10000):  # simulate 10000 (approx. "infinity") epochs
53       node.checkgoactive()
54       node.trysend()
55    print 'mean time to get through =', node.totwait/float(node.totsent)
56
57 if __name__ == '__main__': main()
```

You probably noticed that we added a few things needed for our bookkeeping regarding wait times. Look at line 31, for instance. In the loop here in lines 27-31, we're going through all nodes **n** in the inactive list, simulating their possible generation of new messsages. If a node does generate a new messsage (line 28), we move it from the inactive to the active set (lines 29-30)—AND we "start the wait clock" for this message on line 31.[4]

Similarly, at each epoch, each waiting message will see its wait clock advance by 1, in line 39. When a message finally does get through (line 43), we increment our count of successfully transmitted messages (line 44) and their total wait (line 45). Our final simulation output for mean wait (i.e. "c" above) is done in line 55.

But did you notice something missing? In our earlier ALOHA example, we did 10000 repetitions of simulating the ALOHA system through epoch m. But here **we do only one repetition** but simulate the system through 10000 epochs. How can we get away with this?

The answer won't really emerge until a later unit, but the intuition is that after a large number of epochs, the process has "settled down", and from that point onward we are in effect doing something like repetitions of a repeatable experiment.

3.2 Inv2.py

Here is another inventory example:

```
# Inv2.py, inventory simulation example; like Inv.py, except that it is
# ergodic, and the inventory policy is simply to have a new shipment of
```

[4] We inititalized that variable to None rather than 0 in line 24. This wasn't necessary, but it is good documentation, with the value None reminding the reader of the program that there is no message yet, and thus no wait time. In fact, it would be nice to add a line after line 47, resetting **whotried.wait** back to None, for this reason.

```
# r widgets come in each day, though not more than the warehouse
# capacity w

# we will find E(X), where X is the number of days until a widget is
# shipped; X = 0 if it is shipped out the first day

# usage:  python Inv2.py r w k m startinv nepochs

import random, sys

class g:  # globals
   r = None  # restocking level
   w = None  # warehouse capacity
   k = None  # range for distr of number of orders
   m = None  # range for distr of number of widgets per order
   inv = None  # inventory level
   day = 1  # day number in our simulated time
   rnd = random.Random(98765)  # set seed

class orderclass:  # one object of this class represents one order
   # class variables
   queue = []  # queue of orders
   qlen = 0  # number of orders in queue
   nshipped = 0  # number of widgets shipped so far
   totwait = 0  # total wait time for the widgets shipped so far
   nshippedq = 0  # number of widgets shipped so far which had
                  # encountered a queue from previous night at arrival
   totwaitq = 0  # total wait time for the widgets shipped so far which
                 # had encountered a queue from previous night at arrival
   qleftoverfromlastnight = False  # backlog left from previous night?
   def __init__(self,nw):  # new order of nw widgets
      self.numpending = nw  # widgets not shipped yet
      self.dayorderreceived = g.day  # day the order was received
      # boolean to record whether there had been a queue left over from
      # last night when this order arrived
      self.arrivedtoq = orderclass.qleftoverfromlastnight
      # join the queue
      orderclass.queue.append(self)
      orderclass.qlen += 1
   def simdaytime():
      orderclass.qleftoverfromlastnight = len(orderclass.queue) > 0
      # giving priority to the old orders is equivalent to adding the new
      # orders at the end of the queue
      numneworders = g.rnd.randint(0,g.k)
      for o in range(numneworders):
         nwidgets = g.rnd.randint(1,g.m)
         neworder = orderclass(nwidgets)
      while True:  # go through all the pending orders, until inventory gone
         if orderclass.queue == []: return
         o = orderclass.queue[0]  # current head of queue
         if o.numpending <= g.inv:  # enough inventory for this order?
            partfill = False  # did not just do a partial fill of the order
            orderclass.queue.remove(o)
            orderclass.qlen -= 1
            sendtoday = o.numpending
            g.inv -= sendtoday
         else:  # fill only part of the order
            partfill = True
            sendtoday = g.inv
            o.numpending -= g.inv
            g.inv = 0
         orderclass.nshipped += sendtoday
         waittime = g.day - o.dayorderreceived
         orderclass.totwait += sendtoday * waittime
         if o.arrivedtoq:
            orderclass.nshippedq += sendtoday
            orderclass.totwaitq += sendtoday * waittime
         if partfill: return  # don’t look at any more orders today
```

```
            # o.__dict__ (this comment for debugging!)
   simdaytime = staticmethod(simdaytime)
   def simevening():
      if g.inv + g.r <= g.w: g.inv += g.r
      else: g.inv = g.w
   simevening = staticmethod(simevening)

def main():
   g.r = int(sys.argv[1])
   g.w = int(sys.argv[2])
   g.k = int(sys.argv[3])
   g.m = int(sys.argv[4])
   g.inv = int(sys.argv[5])  # inventory level
   ndays = int(sys.argv[6])  # number of days to be simulated
   for g.day in range(ndays):
      orderclass.simdaytime()
      orderclass.simevening()
   print 'mean wait =', orderclass.totwait/float(orderclass.nshipped)
   print 'mean wait for those encountering a queue =', \
      orderclass.totwaitq/float(orderclass.nshippedq)

if __name__ == '__main__': main()
```