Statistical Inference on Simulation Output

Norm Matloff

March 16, 2008

Contents

1 Review of Confidence Intervals

1.1 Example

Suppose buses arrive at a certain bus stop at random times, with interarrival times being independent exponentially distributed random variables with mean 10 minutes. You arrive at the bus stop every day at a certain time, say 90 minutes after the buses start their morning run. What is your mean wait for the next bus? One can show mathematically that it is 10, but suppose we didn't know that, and wished to find the answer via simulation. We could write a program:

```
1  # Bus.py
2
3  import random,sys
4
5  rnd = random.Random(12345)
6
7  def doexpt(opt):
8     arvtm = 0.0
9     nbus = 0
10    while arvtm < opt:
11       nbus += 1
12       arvtm += rnd.expovariate(0.1)
13    return arvtm-opt
14
15 def main():
16    observationpoint = float(sys.argv[1])
17    nreps = int(sys.argv[2])
18    sum = 0.0
19    for i in range(nreps):
20       wait = doexpt(observationpoint)
21       sum += wait
22    print 'the estimated mean wait is', sum/nreps
23
24 if __name__ == '__main__': main()
```

Running the program yields

```
% python Bus.py 90 1000
the estimated mean wait is 10.2182290246
```

Was 1000 iterations enough? How close is this value 10.2182290246 to the true expected value of waiting time?[1]

What we would like to do is something like what the pollsters do during presidential elections, when they say "Ms. X is supported by 62% of the likely voters, with a margin of error of 4%." In fact, we will do exactly this, in the next section.

1.2 Confidence Intervals for Means

In our example in Section 1.1, let W denote the random time one waits for a bus in this situation. We are using the program to estimate the mean of W, $E(W)$, which we will denote by μ. While we're at it, let's denote the variance of W, $Var(W)$ by σ^2.

[1] Of course, we'll continue to ignore the fact that we know that this value is 10.0. What we're trying to do here is figure out how to answer "how close is it" questions in general, when we don't know the true mean.

Let W_i denote the i^{th} waiting time, i = 1,2,... and let $\bar{W}$ denote the sample mean among our n = **nreps** simulated wait times:

$$\bar{W} = \frac{W_1 + ...W_n}{n} \tag{1}$$

Note that $\bar{W}$ is what the program prints out.

The key points are that

- the random variables W_i have the same distribution; for example, we are just as likely to wait more than 12.6 minutes in our first simulated wait as in our second
- the random variables W_i are independent, e.g. our second simulated wait is independent of the first

Let's estimate σ^2 by taking the sample analogs of the population variance,

$$s^2 = \frac{1}{n}\sum_{i=1}^{n}(W_i - \bar{W})^2 = \frac{1}{n}\sum_{i=1}^{n} W_i^2 - \bar{W}^2 \tag{2}$$

We will thus take our estimate of σ to be s, the square root of that quantity.[2]

One can use the Central Limit Theorem to show that

$$0.95 \approx P\left(\bar{W} - 1.96\frac{s}{\sqrt{n}} < \mu < \bar{W} + 1.96\frac{s}{\sqrt{n}}\right) \tag{3}$$

In other words, we are about 95% sure that the interval

$$(\bar{W} - 1.96\frac{s}{\sqrt{n}}, \bar{W} + 1.96\frac{s}{\sqrt{n}}) \tag{4}$$

contains μ. This is called a 95% **confidence interval** for μ.

We could add this feature to our program:

```
1  # Bus2.py, illustration of the bus paradox
2
3  import random,sys
4  from math import sqrt
5
6  rnd = random.Random(12345)
7
8  def doexpt(opt):
9     arvtm = 0.0
10    nbus = 0
11    while arvtm < opt:
12       nbus += 1
13       arvtm += rnd.expovariate(0.1)
14    return arvtm-opt
```

[2]It is customary to divide by n-1 instead of n, for reasons that are largely theoretical. But when n is large, which we are assuming for the Central Limit Theorem, it doesn't make any appreciable difference.

```
15
16  def main():
17     observationpoint = float(sys.argv[1])
18     nreps = int(sys.argv[2])
19     sum = 0.0
20     sum2 = 0.0
21     for i in range(nreps):
22        wait = doexpt(observationpoint)
23        sum += wait
24        sum2 += wait**2
25     wbar = sum/nreps
26     wbar2 = sum2/nreps
27     s = sqrt(wbar2-wbar**2)
28     print 'the estimated mean wait is', wbar
29     print 'with a 95% margin of error of', 1.96*s/sqrt(nreps)
30
31  if __name__ == '__main__': main()
```

In our example above, the interval would work out to (9.58,10.85). We would then say, "We are about 95% confident that the true mean wait time until the next bus is between 9.58 and 10.85."

What does this really mean? This question is of the utmost importance. We will devote the next section to it.

1.3 Meaning of Confidence Intervals

1.3.1 A Weight Survey in Davis

Let's forget simulation for a minute, and consider the question of measuring the mean weight μ of all adults in the city of Davis. Say we sample 1000 people at random, and record their weights, with W_i being the weight of the i^{th} person in our sample.[3]

Say our interval turns out to be (142.6,158.8). We say that we are about 95% confident that the mean weight μ of all adults in Davis is contained in this interval. **What does this mean?** Say we were to perform this experiment many, many times: We'd sample 1000 people at random, then record our interval $(\bar{W} - 1.96\frac{s}{\sqrt{n}}, \bar{W} + 1.96\frac{s}{\sqrt{n}})$; then we'd sample another 1000 people at random, and record what interval we got that time (it would have a different center and a different radius); then we'd do this a third time, a fourth, a fifth and so on. We would get a large number of intervals—and approximately 95% of them would contain μ, the mean weight in the entire adult population of Davis.

Well, in our simulation case, it is *exactly the same situation*. Simulation is a sampling process. Our μ is the mean in the "population" of all bus waits, while $\bar{W}$ is the mean in our sample of 1000 waits. This is not mere analogy; mathematically the two situations are completely identical.

Let's now admit that we actually know that from previous theory that μ is 10. Then another way to see how to interpret confidence intervals is to say that the output of the following program should be about 0.95:

```
1  # Bus3.py, illustration of the bus paradox
2
3  import random,sys
4  from math import sqrt
```

[3]Do you like our statistical pun here? Typically an example like this would concern people's heights, not weights. But it would be nice to use the same letter for random variables as in Section 1.2, i.e. the letter W, so we'll have our example involve peoples weights instead of heights. It works out neatly, because the word *weight* has the same sound as *wait*.

```
5
6  rnd = random.Random(12345)
7
8  def doexpt(opt):
9     arvtm = 0.0
10    nbus = 0
11    while arvtm < opt:
12       nbus += 1
13       arvtm += rnd.expovariate(0.1)
14    return arvtm-opt
15
16 def main():
17    observationpoint = float(sys.argv[1])
18    nreps = int(sys.argv[2])
19    count = 0
20    for i in range(10000):
21       sum = 0.0
22       sum2 = 0.0
23       for i in range(nreps):
24          wait = doexpt(observationpoint)
25          sum += wait
26          sum2 += wait**2
27       wbar = sum/nreps
28       wbar2 = sum2/nreps
29       s = sqrt(wbar2-wbar**2)
30       if abs(wbar-10.0) < 1.96*s/sqrt(nreps): count += 1
31    print count/10000.0
32
33 if __name__ == '__main__': main()
```

In fact, the output of that program was 0.9483, sure enough about 95%.[4]

1.3.2 One More Point About Interpretation

Some statistics instructors give students the odd warning, "You can't say that the probability is 95% that μ is IN the interval; you can only say that the probability is 95% confident that the interval CONTAINS μ." This of course is nonsense. Of course saying μ is in the interval is equivalent to saying that the inteval contains μ!

Where did this craziness come from? Well, way back in the early days of statistics, some instructor was afraid that a statement like "The probability is 95% that μ is in the interval" would make it sound like μ is a random variable. That was a legitimate fear, because μ is not a random variable. The random entity is the interval, not μ. This is clear in our program **Bus3.py** above—the 10 is constant, while **wbar** and **s** vary from interval to interval.

So, it was reasonable for teachers to warn students not to think μ is a random variable. But later on, some statistics teachers just used it as an unthinking rule, not understanding what the original concern was, and as a result made foolish statements.

1.4 Confidence Intervals for Proportions

In our bus example above, suppose we also want our simulation to print out the (estimated) probability that one must wait longer than 6.2 minutes. We could add code for this in our simulation:

[4] Why not exactly 0.95? Well, first of all, we only did 10000 intervals. But second, remember, the Central Limit Theorem is only approximate, though for n = 1000 here it should be quite close.

```
# Bus4.py, illustration of the bus paradox

import random,sys
from math import sqrt

rnd = random.Random(12345)

def doexpt(opt):
   arvtm = 0.0
   nbus = 0
   while arvtm < opt:
      nbus += 1
      arvtm += rnd.expovariate(0.1)
   return arvtm-opt

def main():
   observationpoint = float(sys.argv[1])
   nreps = int(sys.argv[2])
   sum = 0.0
   sum2 = 0.0
   gt62 = 0  # count of the number of waits > 6.2 minutes
   for i in range(nreps):
      wait = doexpt(observationpoint)
      sum += wait
      sum2 += wait**2
      if wait > 6.2: gt62 += 1
   wbar = sum/nreps
   wbar2 = sum2/nreps
   s = sqrt(wbar2-wbar**2)
   print 'the estimated mean wait is', wbar
   print 'with a 95% margin of error of', 1.96*s/sqrt(nreps)
   p62 = float(gt62)/nreps
   print 'the estimated probability of waiting longer than 6.2 is', p62

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

The value printed out for the probability is 0.544. We again ask the question, how can we determine how close this is to the true population probability?

It turns out that we already have our answer, because a probability is just a special case of a mean. To see this, let

$$Y = \begin{cases} 1, & \text{if } W > 6.2 \\ 0, & \text{otherwise} \end{cases} \tag{5}$$

Then

$$E(Y) = 1 \cdot P(Y = 1) + 0 \cdot P(Y = 0) = P(W > 6.2) \tag{6}$$

Let p denote the probability we are trying to estimate, and let $\widehat{p}$ denote our estimate of it; $\widehat{p}$ is our **p62** in the program.[5] Then one can show that

$$s^2 = \widehat{p}(1 - \widehat{p}) \tag{7}$$

Equation (4) becomes

[5]The quantity is pronounced "p-hat." The "hat" notation is traditional for "estimate of."

$$\left(\widehat{p} - 1.96\sqrt{\widehat{p}(1-\widehat{p})/n}, \widehat{p} + 1.96\sqrt{\widehat{p}(1-\widehat{p})/n} \right) \tag{8}$$

We incorporate that into our program:

```
# Bus5.py, illustration of the bus paradox

import random,sys
from math import sqrt

rnd = random.Random(12345)

def doexpt(opt):
   arvtm = 0.0
   nbus = 0
   while arvtm < opt:
      nbus += 1
      arvtm += rnd.expovariate(0.1)
   return arvtm-opt

def main():
   observationpoint = float(sys.argv[1])
   nreps = int(sys.argv[2])
   sum = 0.0
   sum2 = 0.0
   gt62 = 0  # count of the number of waits > 6.2 minutes
   for i in range(nreps):
      wait = doexpt(observationpoint)
      sum += wait
      sum2 += wait**2
      if wait > 6.2: gt62 += 1
   wbar = sum/nreps
   wbar2 = sum2/nreps
   s = sqrt(wbar2-wbar**2)
   print 'the estimated mean wait is', wbar
   print 'with a 95% margin of error of', 1.96*s/sqrt(nreps)
   p62 = float(gt62)/nreps
   print 'the estimated probability of waiting longer than 6.2 is', p62
   perr = 1.96*sqrt(p62*(1-p62)/nreps)
   print 'with a 95% margin of error of', perr

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

In this case, we get an interval of (0.51,0.57).

Note again that this uses the same principles as our Davis weights example. Suppose we were interested in estimating the proportion of adults in Davis who weigh more than 150 pounds. Suppose that proportion is 0.45 in our sample of 1000 people. This would be our estimate $\widehat{p}$ for the population proportion p, and an approximate 95% confident interval (8)for the population proportion would be (0.42,0.48).

Note also that although we've used the word *proportion* in the Davis weights example instead of *probability*, they are the same. If I choose an adult at random from the population, the probability that his/her weight is more than 150 is equal to the proportion of adults in the population who have weights of more than 150.

And the same principles are used in opinion polls during presidential elections. Here p is the proportion of people who plan to vote for the given candidate. We again use (8).

Note that in both the Davis and election examples, it doesn't matter what the size of the population is. The distribution of $\widehat{p}$, and thus the accuracy of $\widehat{p}$, depends only on p and n, the latter being the number of people

we sample.[6] So if you've ever wondered how a nationwide election poll can get by with sampling only 1200 people, which is a common number, you now know the answer.

In fact, since the maximum possible value of $\widehat{p}(1-\widehat{p})$ is 0.25,[7] the pollsters know that their margin of error with n = 1200 will be at most $1.96 \times 0.5/\sqrt{1200}$, or about 3%.

1.5 Conditional Confidence Intervals

Often in simulations we will be interested in conditional quantities. In a queuing model, for instance, we might wish to estimate the mean wait time of jobs which arrive when the server is busy. We still use the usual formulas, e.g. (4) and (8), but keep in mind that the value of n in those formulas, even though we might have run the simulation for a long time and the n for unconditional quantities would be large.

2 Statistical Inference in Infinite Time Horizon Settings

2.1 Illustrative Example

Recall our example program **Aloha2.py** in our introductory unit on simulation, `http://heather.cs.ucdavis.edu/~matloff/156/PLN/SimIntro.pdf`. Here network nodes would generate messages to send at random times, but also randomly refrain from sending, in order to avoid collisions.

Let Y_i denote the number of active nodes at epoch i. We are interested in what happens as time goes to infinity. But what does that really mean?

Intuitively, we feel that this system will converge to steady state in some sense. This certainly doesn't mean in the deterministic sense. The latter would mean that

$$\lim_{i \to \infty} Y_i \tag{9}$$

exists, which cannot be true; the Y_i will continue to fluctuate. But we do feel that they will have a long-run average, i.e. that

$$\lim_{i \to \infty} \frac{Y_1 + ... + Y_i}{i} \tag{10}$$

will exist as some constant ν and that

$$\lim_{i \to \infty} E(Y_i) = \nu \tag{11}$$

These things can be proven under certain reasonable mathematical conditions. In fact, under these conditions one can show that

[6] Implicit in our assumption that the W_i are independent is that we are sampling **with replacement**, which means it's possible that our random sampling process might choose the same person twice. But except for cases in which our sample size is a substantial fraction of the population size, the probability of getting the same person twice would be very low, so it doesn't matter.

[7] Use calculus to find the maximum value of f(x) = x(1-x).

$$\lim_{i\rightarrow} P(Y_i = k) \tag{12}$$

exists for each k. For instance, the probability that there are, say, 3 nodes active at time i will converge as i gets large.

The questions of interest here will be

- How long do we need to run the simulation to get a reasonably accurate estimate of ν?
- How can we find a valid confidence interval for ν?

2.2 Mathematical Statement of the Problem

This task would be straightforward in the case of estimating $\nu_8 = E(Y_8)$. We could simulate, say, 1000 replications of the system through time 8. Letting Y_{8j} denote the j^{th} of these replications, we would these random variables would be *independent* and *identically distributed* with mean ν_8. Equation (4) would apply perfectly. The width of the confidence interval would then give us a good idea as to the accuracy of our estimate of ν.

The situation in (10) is not nearly so straightforward. There the Y_i are *neither* independent *nor* identically distributed. Let's look at this a bit more:

- The Y_i are not independent. The number of active nodes at time 28, say, will have some effect on the number at time 29.
- The Y_i are not identically distributed. The process does converge to a steady state, but it has a different distribution at each finite time. For example, $Y_1 = 0$ with certainty, while $\lim_{i\rightarrow\infty} P(Y_i = 0)$ will be quite different.

The problem of not being identically distributed can be phrased as saying that near the beginning of the simulation, we are in a **transient period**, during which the process is settling down to steady state. Of course, technically speaking, the transient period is infinite, since we "reach" steady state only after an infinite amount of time. But in practical terms, we come reasonably close enough to steady state after a finite amount of time. In (12), for instance, the limiting probability for k = 2 might be, say, 0.64, but after a certain number of epochs it will be within 0.02 of that.

2.3 Determining the Transient Period

The problem of the Y_i not being identically distributed is somewhat easier to deal with. To discuss that, let's start by looking at (10):

```
1  # AlohaC2.py, form of slotted ALOHA
2
3  import random, sys
4
5  class node:  # one object of this class models one network node
6     s = 4  # number of nodes
```

```
7      b = 0.3  # backoff parameter; refrain from sending with probability b
8      q = 0.2  # msg creation probability
9      activeset = []  # which nodes are active now
10     inactiveset = []  # which nodes are inactive now
11     r = random.Random(98765)
12     def __init__(self):
13        node.inactiveset.append(self)
14     def checkgoactive():  # generate nodes which change to active
15        for n in node.inactiveset:
16           if node.r.uniform(0,1) < node.q:
17              node.inactiveset.remove(n)
18              node.activeset.append(n)
19     checkgoactive = staticmethod(checkgoactive)
20     def trysend():  # the active nodes try to send, or not
21        numnodestried = 0
22        whotriedlast = None
23        for n in node.activeset:
24           if node.r.uniform(0,1) > node.b:
25              whotriedlast = n
26              numnodestried += 1
27        if numnodestried == 1:
28           node.activeset.remove(whotriedlast)
29           node.inactiveset.append(whotriedlast)
30     trysend = staticmethod(trysend)
31     def reset():
32        for n in node.activeset:
33           node.activeset.remove(n)
34           node.inactiveset.append(n)
35     reset = staticmethod(reset)
36
37  def main():
38     for i in range(node.s): node()
39     sum = 0
40     for epoch in range(10000):  # simulate until time 9999
41        node.checkgoactive()
42        node.trysend()
43        sum += len(node.activeset)  # add Yi to total
44        print epoch, float(sum)/(epoch+1)  # to be redirected to a file
45
46  if __name__ == '__main__': main()
```

Here we've adapted the program to look only at a specific value of **b**, and to look at many epochs instead of replications for epoch 8. Most importantly, we are printing out the values of (10) as we go along. Figure 1 shows the graph of these values.

Convergence seems to occur after about 1000 epochs. What that says is that to deal with the problem of bias during the transient period, we can just throw out the first 1000 epochs of our simulation. Instead of computing

$$\frac{Y_1 + ... + Y_{10000}}{10000} \tag{13}$$

we would compute

$$\frac{Y_{1001} + ... + Y_{10000}}{9000} \tag{14}$$

Actually, even discarding 1000 observation is much too conservative. The expected values in (11) converge much faster than that. The slowness of convergence here is due to the fact that we have only one observation at each epoch. If we were to do many replications at each epoch (i.e. run the above code, with 10000

epochs, in many replications), we'd see much faster convergence. Short of doing that, though, we can take the conservative route.

2.4 Has the Simulation Run Long Enough?

We see that (10) has settled down to around 2.95. For informal purposes, this may be good enough.

But in many applications of simulation, such as writing research publications and government reports, we must present not only the sample estimate but also a margin of error. So, again we must ask how to find a confidence inteval for ν.

We saw above how to deal with the fact that our observations Y_i are not identically distributed. What should we do about their lack of independence? There are three main methods for handling this, which we will now describe.

2.5 The Replications Method

Here we would run, say, 1000 replications of the simulation up to time 10000, getting 1000 values of (14). These values would be independent and identically distributed, and thus we could form a confidence interval for ν using (4).

This is a clean way to solve the problem, but obviously with considerable extra run time.

2.6 The Batch Means Method

Though our Y_i are not independent, they are approximately independent when spaced far enough apart. For instance, Y_{26} and Y_{39} should have almost nothing to do with each other.

So, we could take our 9000 observations $Y_{1001}, ..., Y_{10000}$ and divide them into batches of size, say, 50. $Y_{1001}, ..., Y_{1050}$ would be the first batch, $Y_{1051}, ..., Y_{1100}$ would be the second batch, and so on. Let B_k denote the mean Y value in the k^{th} batch. Then the B_k would be approximately independent, by virtue of the spacing, and approximately identically distributed, by virtue of their having occurred past the transient period. Thus we could use them in (4), with 9000/50 = 180 in place of n, $\sum_{k=1}^{180} B_k/180$ in place of $\bar{W}$, and with s calculated from (2) for the B_k.

This is probably the most commonly-used method.

2.7 The Regenerative Process Method

This method is the most elegant of the three, though it is not always feasible. The key to this method is to find a state of the system in which "time starts over." Some readers may find this reminiscent of the fact that exponential distributions are memoryless, and what we are discussing here is similar, but much broader.

Example 1:

Consider an example from our introductory unit on SimPy, at `http://heather.cs.ucdavis.edu/~matloff/156/PLN/DESimIntro.pdf`. We have two machines, whose up times are exponentially distributed with intensity 1.0, and whose repair times are exponentially distributed with intensity 0.5. The

repairperson is not summoned until both machines are down. We are interested in the long-run proportion ρ of the time in which both machines are up simultaneously.

Consider the times at which the system enters the state in which both machines are up, which we will call the "full-capacity state." Our simulation has them start in this state. After a while, one of the machines will go down, then the other. Then one of the machines, say machine B, will begin repair, and after that is done, the other machine, A, will begin repair. Actually, while A is being repaired, B may go down again and so on, but eventually we will reach a situation in which we have entered into the full-capacity state for the second time. This is our first "time starts over" point, which is called a **regeneration point**. The subsequent regeneration points are the subsequent times at which we enter full capacity state.

We must ask the question, does "time start over" at our proposed regeneration times? Probabilistically speaking, is the distribution of the process from such a time onward the same as it was starting at time 0.0? The answer is yes, but only because up times are exponentially distributed and thus memoryless. If up times were to have some other distribution, things would not work.

For instance, say at certain regeneration point is triggered by a repair of A, with B having already been up for a while. If up time is not memoryless, the fact that B has been up for a while changes the probability that it will go down soon. So, without the exponential distribution here, time would NOT "start over" when A comes up.

But if up times are exponentially distributed, the times at which we enter the full-capacity state are indeed times in which the system "starts over," and acts independently from the past. In fact, we would not even need to have repair times be exponentially distributed. distribution.[8]

There are other sets of regeneration points in this process as well. One such set consists of the times at which the repairperson is summoned. This set is especially important, because it does not depend on the assumption of exponential distributions for either up time or down time.

Note carefully that we continue to make the independence assumption for the set of up and down times of the machines. In other words, if say Machine A has a long up time in its first up period, we are assuming that this does not make the machine any more or less likely to have a long up time in its second up period, etc.

Example 2:

Consider the G/G/1 queue, where the 1 means there is one server and the two Gs stand for "general," meaning that the interarrival times and service times have general distributions, not necessarily exponential. Though one might at first think that the lack of exponential distributions would imply lack of regeneration points, this is definitely not the case. In particular, the points in time in which an arriving job encounters an idle server are indeed regeneration points.

Let us suppose that in this example we are interested in μ, the long-run average queue length.

2.7.1 What We'll Be Doing Here

Keep in mind that in both these examples, there is no problem in estimating the quantity of interest, ρ and μ, respectively. The question at hand is how to find valid confidence intervals for these quantities. It will happen, though, that we get a new estimator in the process as well.

[8] Note that we have also tacitly assumed that the two machines act independently of each other, that for a given machine its successive up times are independent and so on.

Let's call the periods between successive regeneration points **regeneration cycles**. The basic idea is that the activities of our process in one cycle is independent of, and identically distributed with, the corresponding activities in the other cycles. For instance in Example 1, the total time in which the system is in full capacity during the second such period is independent of, and has the same distribution as, the corresponding time in the first period.[9]

2.7.2 Notation

Let V_t denote the state of our system at time t. V_t is a vector of quantities that fully describe the current state of the system. For instance, in our machine repair example above, V_t would consist of the following quantities at time t:

- 1 or 0, according to whether machine A is up
- 1 or 0, according to whether machine B is up
- time A has been up, if it's up
- time B has been up, if it's up
- 1 or 0, according to whether machine A is being repaired
- 1 or 0, according to whether machine B is being repaired
- time A has been under repair, if it's being repaired
- time B has been under repair, if it's being repaired

We need the population quantity γ that we are trying to estimate to be of the form

$$\gamma = \lim_{t \to \infty} \frac{1}{t} \int_0^t h(V_u)\, du \tag{15}$$

for some function h.

For instance, in our machine repair example we had

$$h(t) = \begin{cases} 1, & \text{if full capacity at time t} \\ 0, & \text{otherwise} \end{cases} \tag{16}$$

Here, the integral in (15) is the total time we are in full capacity during (0,t). Dividing that integral by t gives us the proportion of time we are in full capacity during (0,t).

In our queuing system example, $h(V_t)$ would simply be the length of the queue at time t. In this case, its integral from 0 to t is the sum of the queue lengths during that time, multiplied by the amount of time each length occurred.

[9] You might object to the assertion here that the full-capacity times for the two periods have the same distribution. You may ask, "What if the second period is longer? Won't that mean that its full-capacity time will tend to be longer?" That is true, but it merely says that, *conditioned on the length of the period*, the full-capacity times will not have the same distribution. But unconditionally, the two distributions will be identical.

Let T_i denote the i^{th} regeneration point, with $T_0 = 0.0$. Let $D_i = T_i - T_{i-1}$ represent the duration of the i^{th} regeneration cycle. We run the simulation for n cycles.

Now set

$$C_i = \int_{T_{i-1}}^{T_i} h(V_u)\, du \tag{17}$$

so that for instance in Example 1 above, C_5 is the amount of time the system spends in the full-capacity state during the fifth regeneration cycle.

As usual, define sample means:

$$\overline{C} = \frac{1}{n} \sum_{i=1}^{n} C_i \tag{18}$$

$$\overline{D} = \frac{1}{n} \sum_{i=1}^{n} D_i \tag{19}$$

2.7.3 Estimator and Confidence Interval

Renewal theory (a topic treated in our unit at `http://heather.cs.ucdavis.edu/~matloff/256/PLN/Renewal.pdf`) can be used to prove, and it is intuitive, that ρ is equal to the ratio between the mean time in full capacity per cycle divided by the mean cycle time,

$$\gamma = \frac{E(C)}{E(D)} \tag{20}$$

where C and D are random variables having the common distribution of the C_i and D_i, respectively. Then the natural estimate of γ based on our simulation output is

$$\widehat{\gamma} = \frac{\overline{C}}{\overline{D}} \tag{21}$$

Now consider the random variables $X_i = C_i - \gamma D_i$. Remember, we don't know the value of γ, but these random variables exist.

Because of the nature of the regeneration cycles, the X_i are independent and identically distributed. From (20), we know that they have mean 0.

Thus the Central Limit Theorem (CLT) tells us that the random variable

$$\frac{\overline{X}}{\sqrt{Var(X)/n}} \tag{22}$$

is approximately distributed N(0,1) when n is large, where $\overline{X}(X_1 + ...X_n)/n = \overline{C} - \gamma\overline{D}$.

For a generic random variable X having the distribution of the X_i,

$$Var(X) = Var(C) + \gamma^2 Var(D) - 2\gamma Cov(C,D) \tag{23}$$

Replacing by sample analogs in (23) gives us (28). It can be shown that the CLT is still valid if we make this substitution in (22). So, the quantity

$$\frac{\overline{X}}{s_X/n^{0.5}} \tag{24}$$

has an approximate N(0,1) distribution. So,

$$\begin{aligned} 0.95 &\approx P\left(-1.96\frac{s_X}{\sqrt{n}} < \overline{C} - \gamma\overline{D} < 1.96\frac{s_X}{\sqrt{n}}\right) &(25)\\ &= \left(-1.96\frac{s_X}{\overline{D}\sqrt{n}} < \widehat{\gamma} - \gamma < 1.96\frac{s_X}{\overline{D}\sqrt{n}}\right) &(26)\end{aligned}$$

Thus an approximate 95% confidence interval for γ is

$$(\widehat{\gamma} - 1.96\frac{s_X}{\overline{D}n^{0.5}}, \widehat{\gamma} + 1.96\frac{s_X}{\overline{D}n^{0.5}}) \tag{27}$$

where

$$s_X^2 = \frac{1}{n}\sum_{i=1}^{n}(C_i - \bar{C})^2 + \widehat{\gamma}^2\frac{1}{n}\sum_{i=1}^{n}(D_i - \bar{D})^2 - 2\widehat{\gamma}\frac{1}{n}\sum_{i=1}^{n}(C_i - \bar{C})(D_i - \bar{D}) \tag{28}$$

2.7.4 Example: Machine Repair Simulation

Let's take our earlier machine repair example from Section 2.7. We'll use the first set of regeneration points described there, the times at which we enter full-capacity state. This requires us to assume exponential distributions for the up times (though not the repair times), but it makes the illustration of the formation of a confidence interval easier.

Here is the code:

```
#!/usr/bin/env python

# MachRep3Regen.py

# SimPy example:  Variation of Mach1.py, Mach2.py.  Two machines, but
# sometimes break down.  Up time is exponentially distributed with mean
# 1.0, and repair time is exponentially distributed with mean 0.5.  In
# this example,there is only one repairperson, and she is not summoned
# until both machines are down.  We find the proportion of time that
# both machines are up simultaneously (mathematically this can be shown
# to be 0.181818...), extending our point estimate for that proportion
# to an approximate 90% regenerative confidence interval.  We choose as our
# regeneration points the times at which we enter the "full capacity"
# state, i.e. both machines up (valid only if up time is exponentially
```

```
# distributed).

from SimPy.Simulation import *
from random import Random,expovariate
from math import *

class G:  # globals
   Rnd = Random(12345)
   RepairPerson = Resource(1)

class MachineClass(Process):
   MachineList = []  # list of all objects of this class
   UpRate = 1/1.0
   RepairRate = 1/0.5
   TotalUpTime = 0.0  # total up time for all machines
   NextID = 0  # next available ID number for MachineClass objects
   NUp = 0  # number of machines currently up
   CMon = Monitor()
   DMon = Monitor()
   # these concern the regeneration cycles
   StartCycle = 0.0  # time the current cycle started
   BeginningOfCycle = True  # both machines are up now; once one machine
                            # goes down, they will never both be up
                            # again during this cycle
   BothUpThisCycle = None  # amount of time both machines up in this cycle
   def __init__(self):
      Process.__init__(self)
      self.StartUpTime = None  # time the current up period started
      self.ID = MachineClass.NextID   # ID for this MachineClass object
      MachineClass.NextID += 1
      MachineClass.MachineList.append(self)
      MachineClass.NUp += 1  # start in up mode
   def Run(self):
      while 1:
         self.StartUpTime = now()
         yield hold,self,G.Rnd.expovariate(MachineClass.UpRate)
         UpTime = now() - self.StartUpTime
         MachineClass.TotalUpTime += UpTime
         MachineClass.NUp -= 1
         if MachineClass.BeginningOfCycle:
            MachineClass.BothUpThisCycle = now() - MachineClass.StartCycle
            MachineClass.BeginningOfCycle = False
         # update number of up machines
         # if only one machine down, then wait for the other to go down
         if MachineClass.NUp == 1:
            yield passivate,self
         # here is the case in which we are the second machine down;
         # either (a) the other machine was waiting for this machine to
         # go down, or (b) the other machine is in the process of being
         # repaired
         elif G.RepairPerson.n == 1:
            reactivate(MachineClass.MachineList[1-self.ID])
         # now go to repair
         yield request,self,G.RepairPerson
         yield hold,self,G.Rnd.expovariate(MachineClass.RepairRate)
         MachineClass.NUp += 1
         if MachineClass.NUp == 2:
            MachineClass.CMon.observe(MachineClass.BothUpThisCycle)
            MachineClass.DMon.observe(now()-MachineClass.StartCycle)
            MachineClass.StartCycle = now()
            MachineClass.BeginningOfCycle = True
         yield release,self,G.RepairPerson

def Regen(CMon,DMon,Z):
   CBar = CMon.mean()
   DBar = DMon.mean()
   GammaHat = CBar/DBar
   CDBar = 0.0
```

```
83      N = len(CMon)
84      for i in range(N):
85         CDBar += CMon[i][1] * DMon[i][1]
86      CDBar /= N
87      CDBar -= CBar * DBar
88      S2 = CMon.var() + GammaHat**2 * DMon.var() - 2 * GammaHat * CDBar
89      Radius = Z * sqrt(S2) / (DBar*sqrt(N))
90      return GammaHat,Radius
91
92  def main():
93      initialize()
94      for I in range(2):
95         M = MachineClass()
96         activate(M,M.Run())
97      MaxSimtime = 1000.0
98      simulate(until=MaxSimtime)
99      # for a 90% CI, use 1.65
100     (Center,Radius) = Regen(MachineClass.CMon,MachineClass.DMon,1.65)
101     print 'center, radius of CI:', Center, Radius
102
103 if __name__ == '__main__':  main()
104
```

2.7.5 Feasibility

Note that in a simulation with a very complicated state vector, a regeneration point may be difficult to find. Moreover, the regeneration cycles may be extremely long. These considerations make this method difficult to use in some applications.

3 Learning More

The use of statistics is widespread throughout the world. Unfortunately, much of it is *mis*use, even by professional statisticians. We saw an example in Section 1.3.2.

For a careful look at the foundations and applications of statistics, see my online course, at `http://heather.cs.ucdavis.edu/~matloff/probstatbook.html`.

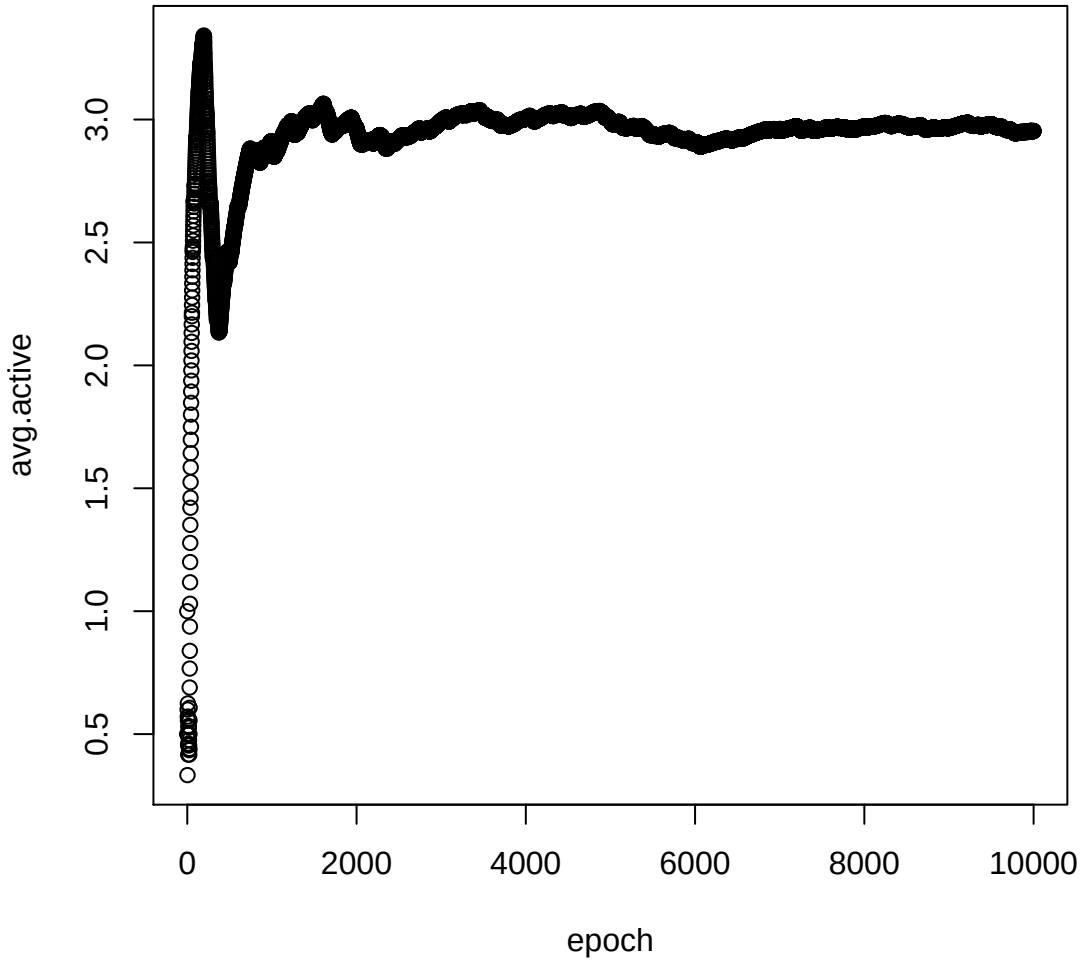

Figure 1: Mean Yi