Carolyn Cotterman
Stat 243
November 8, 2010

Assignment 2

1 Introduction

In this assignment, I explore algorithms for calculating the means and variances of vectors of numbers, and also explore algorithms for random number generation. My source code and output are contained in the following documents:

- HW2_CC.c (source code)
- hw2_cc_out.txt (output)

2 Mean and Variance Algorithms

The most common way to express the sample variance, s^2, is as follows:

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

where n is the number of observations, x_i is the value of observation i, and $\bar{x}$ is the sample mean. One problem with calculating s^2 according to the above equation is that it necessitates reading the data twice (or, alternatively, storing the data values so that they can be used again). In the first pass through the data, we would calculate $\bar{x}$, and then in the second pass, we would apply equation 1 to find s^2 using our stored value for $\bar{x}$. Reading the data twice (or storing it) is costly in terms of machine resources. All three of the alternative algorithms, presented below, avoid this unnecessary use of machine resources.

I implemented each of the algorithms, below, in order to calculate the mean and variance of a well-behaving vector of numbers. All three methods generated correct results (by comparison to the results provided by Stata). My vector consisted of the numbers 1, 2, 3, -4, -8, 25, with a mean of 3.166667, and a variance of 131.766667.

- **Desk Calculator Algorithm**: By rearranging equation 1, we can derive the following "desk calculator" algorithm:

$$s^2 = \frac{1}{n-1}\left((\sum_{i=1}^{n} x_i^2) - (\sum_{i=1}^{n} x_i)^2\right) \tag{2}$$

  Compared to equation 1, this algorithm has the advantage of only needing to store three numbers in memory. However, as we shall see in the next section, it does not provide correct values when our data's coefficient of variation is very small.

- **Method of Provisional Means**: The "method of provisional means," as its name suggests, involves calculating a "provisional" value for the data's mean, denoted by $\bar{x}^{(k)}$, and a provisional value for the data's variance, denoted by $s^{(k)}$, after k observations have been read in. When k=1 (only the first observation, x_1, has been read), then we have $\bar{x}^{(1)} = x_1$ and $s^{(1)} = 0$. Subsequent provisional values (for $k > 1$) are then calculated as follows:

$$\bar{x}^{(k)} = \bar{x}^{(k-1)} + \frac{1}{k}(x_k - \bar{x}^{(k-1)}) \tag{3}$$

$$s^{(k)} = s^{(k-1)} + (x_k - \bar{x}^{(k-1)})(x_k - \bar{x}^{(k)}) \tag{4}$$

  When all of the data has been read (k=n), then we have the mean and variance of the full data set by recognizing that:

$$s^2 = \frac{s^{(n)}}{n-1} \tag{5}$$

  and

$$\bar{x} = \bar{x}^{(n)} \tag{6}$$

  Like the desk calculator algorithm, this one also allows us to get by with taking up three places in memory. However, as we shall see in the next section, this algorithm is more robust to handling data with a very small coefficient of variation.

- **Centering around the First Observation (i.e.,"The SAS Method")**: This last algorithm that I explore is the one implemented by SAS software. It consists of using the desk calculator algorithm (equation 2), but only after substracting the first observation from each of the other observations. The resulting variance is the correct variance for the data (subtracting a constant from every observation does not effect variance), while the data's mean will be equal to the sum of the resulting mean and the value of the first observation. You may be skeptical at first but, this algorithm avoids the problem that the desk calculator algorithm has in terms of dealing with data that has a very small coefficient of variation!

3 Ill-conditioned Data

In the previous section, I verified that our three algorithms for calculating the mean and variance generate correct answers for a simple vector of well-behaving numbers. But will these algirithms generate the correct answer if I feed them "ill-conditioned" data? To answer this question, I started with a simple sequence of numbers (5, 3, 9, 6, 3, 7, 2, 4, 8, 2, 3, 5, 6), which has a mean of 4.846154 and a variance of 5.141026. I then added a constant, C, to each element of this vector (thus reducing the data's coefficient of variation), and again calculated its mean and variance using our three different algorithms. In theory, the new mean should equal 4.846154 + C while the variance should be the same as before. In fact, I found that the provisional means and SAS methods continued to generate correct answers (up to 6 decimals) even when the data's coefficient of variation was as small as 0.00000001417.

The desk calculator method, however, generated variance values that were progressively far from the mark (while continuing to produce correct values for the sample mean). By systematically increasing the value of C and re-calculating the mean and variance of the resulting sequence of numbers, I was able to identify how low the coefficient of variation could get before the calculated variance was off by a pre-specified amount of interest. My results can be summarized as follows (please see output of code for additional details):

- Coefficient of variation = 0.00000864902 led to a calculated variance that differed from the truth by more than .00001.
- Coefficient of variation = 0.00000003582 led to a calculated variance that differed from the truth by more than 1.
- Coefficient of variation = 0.00000001417 led to a calculated variance that differed from the truth by more than 5.

Final note: In doing this problem, I learned that to evaluate whether two doubles (call them d1 and d2) are equal to one another, you may not want to use a simple comparison statement (e.g., d1 == d2). Not all integers can be represented in binary form precisely, and computations result in tiny rounding errors that may be inconsequential in terms of the level of accuracy that you care about. As far as you're concerned, d1 and d2 may be indistinguishable from one another even though the computer evaluates the statement, "d1 == d2" as false. Rather than using a simple comparison statement, then, it may be wise to test whether d1 and d2 are within a certain interval of one another (e.g., $|d1 - d2| < epsilon$).

4 Uniform Random Numbers

Generating legitimately random numbers is a huge challenge. In this section, I describe the method by which I generated a series of uniform "pseudo-random numbers" using what is known as the multiplicative congruential generator. For lingual simplicity, I will refer to the "pseudo-random numbers" that I generate as simply "random numbers." Even though they are not perfectly random, their properties are close enough to that of truly randomly generated numbers to satisfy my purposes.

According to the multiplicative congruential generator method, uniform random numbers (call them u_i) on the interval from 0 to 1 can be generated according to the following equations:

$$x_i = mod(ax_{i-1}, m) \tag{7}$$

$$u_i = x_i/m \tag{8}$$

In my implementation of this algorithm, I set m = 2^32 and a = 8003 (this value satisfies the criteria that mod(a,8) equals 3 or 5 – a condition necessary for generating the maximal period.) I wrote my code such that each time the random number generator is called, the value of "seed" (initial value to use for x_{i-1}) is updated so that multiple calls to my function will produce unique random number sequences, unless the seed were to be manually reset

to a desired value each time. (This is analogous to how the random number generator in R behaves.)

I implemented the described method several times, creating arrays of n U(0,1) numbers of various lengths. My results are as follows:

- n = 10: mean is 0.459205 and variance is 0.113436
- n = 30: mean is 0.614676 and variance is 0.061669
- n = 1,000: mean is 0.497792 and variance is 0.083841
- n = 50,000: mean is 0.499697 and variance is 0.083537

The theoretical mean of a uniform distribution from 0 to 1 is .5 and its variance is 1/12 = .083333. My results are consistent with this theory, and demonstrate convergence as I increase the number of observations.

5 Normal Random Numbers

In this section, I explore two different methods for generating normal random numbers.

5.1 Method 1: Polar Method

I implemented the Polar Method for generating N(0,1) random numbers according to the following procedure:

1. Generate $U_1, U_2 \sim U(0,1)$
2. Calculate $V_j = 2u_j - 1, \quad i = 1, 2, ...$
   $v_j \sim U(-1, 1)$
3. Calculate $S = V_1^2 + v_2^2$
4. If $S \geq 1$, go to step 1.
5. $X_1 = V_1\sqrt{\frac{-2\ln(S)}{S}}$ and $X_2 = V_2\sqrt{\frac{-2\ln(S)}{S}}$
   Each of X_1 and $X_2 \sim N(0,1)$ and independent

I generated 10 samples consisting of 20 observations each, and found that the means of my samples were reasonably close to 0 while the variances were around 1. After a hard day's work, these results were quite satisfying, as they supported the notion that I had indeed created vectors of N(0,1) random numbers.

5.2 Method 2: Summing of Uniform Random Numbers

This method of generating N(0,1) random numbers consists of first generating six vectors of Unif(0,1) random numbers using the function that I developed previously. I could easily convert these vectors into vectors of Unif(-1,1) random numbers by multiplying each element by 2 (resulting in Unif(0,2) random numbers), and then subtracting 1. Next, I added each of the six vectors together (element-by-element), which yielded a vector of N(0,2) random numbers. Finally, I scaled the numbers by a factor of $(1/\sqrt{2})$ in order to generate a vector of N(0,1) random numbers.

As with the polar method, I generated 10 samples consisting of 20 observations each, and found that the means of my samples were reasonably close to 0 while the variances were around 1. They would even more closely correspond to a N(0,1) distribution if I had combined more than 6 Unif(-1,1) vectors to generate them.

5.3 Run Time Comparison for Methods of N(0,1) Generation

Using the timer.h files that Phil Spector generously provided, I was able to compare the computing time associated with creating uniform random numbers under the 2 methods described, above. Since the time required to generate just a couple of random numbers is neglible, I needed to generate many random numbers in order to get a more accurate run-time comparison across methods that would not be too strongly influenced by factors unrelated to the algorithms. To generate 10,000 vectors of 10,000 observations each, the polar method took 97.19 seconds while the second method took 77.42 seconds: the polar method appears to take about 25% longer.

6 Conclusion

In completing this assignment, I have learned more about writing and working with functions, handling arrays, and dealing with rounding errors. I have learned that rounding errors can cause large problems if one does not develop algorithms carefully. There might be multiple ways to do a certain calculation, and while any one of them may be just as good as the next for certain situations, it is important to be aware that they may not behave the same when you scale up (some may run slower), and/or use "ill-conditioned" data.