Political Approval Rating:

In this project, you will use some important properties of random numbers to assess the results of a poll that ranks the public’s approval of some person or issue. Suppose you asked everyone in the country what they thought of the president’s performance, on a scale of 0% to 100%, where 0% is terrible and 100% is wonderful. If you averaged all the answers, you would get the true average opinion of everyone in the country. We call that the *population mean*. Unfortunately, asking everyone is expensive, so pollsters just ask a few *representative* people, perhaps 100.

Even when you take a truly representative sample, the average of what your samples say won’t equal the average of what everyone would say. The measured average will have a *variance*. Let:

x = a particular sample’s approval value

**x** = population mean approval value = average approval of everyone = the “truth”

Using bold angled brackets to denote averaging, we define:

variance = **<**(x – **x**)2**>** =

average of the square of (each sample’s approval – population mean approval).

With a little algebra, one can show that:

**<**(x – **x**)2**>** = (**<**x2**>** - **<**x**>**2) + (**<**x**>** - **x**)2

The first term on the right-hand side is:

**<**x2**>** - **<**x**>**2 =

(average of square of each sample’s approval) – (average sample approval squared)

This is something we can compute from our sample data only.

The second term on the right-hand side is:

(**<**x**>** - **x**)2 = square of (average sample approval – population’s mean approval)

We’ll call this the *variance of the average*. This helps us estimate the error in our sampling experiment. What we usually want to know is:

standard deviation of the average = sqrt[(**<**x**>** - **x**)2] = Math.abs(**<**x**>** - **x**)

This tells approximately how far off our computed result is. It could be off in either direction by as much as two or three times the standard deviation.

In real life, we almost never know the “true” population mean, **x**, so we can’t just use Math.abs(**<**x**> ‑ x**). Instead, we have to estimate our error indirectly. On the average, it turns out that:

(**<**x**>** - **x**)2 ≈ (**<**x2**>** - **<**x**>**2) / (samples – 1)

That is, the variance of the average ≈ raw variance / (samples – 1).

Thus, we can estimate the standard deviation of an experimental determination of an average by calculating:

standard deviation of mean ≈ sqrt[(**<**x2**>** - **<**x**>**2) / (samples – 1)]

We must always take more than one sample, and the more samples we take, the smaller our error becomes, but it decreases only as the square root of the number of samples.

Here’s a suggested UML class diagram:

**ApprovalDriver**

+main(args : String[]) : void

**Approval**

-populationMean : double

-mean : double

-variance : double

+doPoll(samples : int) : void

+getMean() : double

+getVariance() : double

+setPopulationMean() : void

+getPopulationMean() : double

+getSample : double

+verifyModel() : void

The main method calls verifyModel to prove that the getSample algorithm is OK. Then it calls setPopulationMean to set the population mean approval at an unknown random value between 0.0 and 1.0. Then main asks the user for a desired number of samples, and it instantiates an Approval object, poll. Then it asks poll to call doPoll and prints poll’s mean value, as a percentage. It gets poll’s raw variance and divides it by (samples – 1) to obtain an estimate of the variance of the measured average, relative to the underlying true population mean. It prints the standard deviation of variation in the measured average, as a percentage. Finally, it reveals the “true” underlying population mean, as a percentage, for comparison.

The doPoll method loops for the specified number of samples, accumulating value and value squared. After the looping, it divides each accumulation by the specified number of samples to obtain the respective averages. Then it sets mean, and:

variance ← avgOfValueSquared – avgValue \* avgValue.

The setPopulationMean method just makes populationMean equal to whatever value is returned by Math.random.

The getSample method implements the following algorithm, which provides a simple representation of the population’s distribution of approval values:

sample ← Math.random

where ← Math.random

if where < populationMean

sample ← populationMean + (1 – populationMean) \* sample

else

sample ← populationMean – populationMean \* sample

The verifyModel method establishes an independent population mean, and loops through one million samples, to confirm that the average of all of them approximately equals the established population mean, and to confirm that all samples are within the allowed range.

* 1. Write a Java program that uses the above methods to produce something like the following results:

Sample session:

populationMean = 0.588476003095624 for 1000000 samples

maxValue = 0.9999997135618746

minValue = 9.054368410588154E-7

avgValue = 0.588630443979852

Enter number of samples in poll: *50*

poll's average = 52%

standard deviation of poll's average = 4%

population average = 49%

* 1. Write another driver for the Approval class that conducts a large number of polls and verifies experimentally the relationship:

(**<**x**>** - **x**)2 ≈ (**<**x2**>** - **<**x**>**2) / (samples – 1)

for arbitrary user-specification of number of samples.

Do this by looping through a user-specified number of iterations. In each iteration, take a separate poll, and accumulate raw variance (right-side numerator) and the square of that poll’s mean minus population mean (left side). After the loop, divide each accumulation by the number of iterations, and divide the quotients to produce an average estimate of (samples - 1).

Sample session:

PopulationMean = 0.6785929035920892

Enter number of polls: *100000*

Enter number of samples per poll: *100*

samplesMinusOne = 98.93854576326652