Ph3 Mathematica Homework: Week 8

Eric D. Black
California Institute of Technology
v1.2

> Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin. For, as has been pointed out several times, there is no such thing as a random number — there are only methods to produce random numbers, and a strict arithmetic procedure of course is not such a method. *John von Neumann*

1 Random numbers

You can ask Mathematica to give you a random number using the following commands.

1. `RandomReal[{min, max}]` - Gives a pseudorandom real number between `min` and `max`. If you leave out the argument (*i.e.* `RandomReal[]`), Mathematica will assume `min=0` and `max=1`.

2. `RandomInteger[{min, max}]` - Returns a pseudorandom integer between `min` and `max`. Leaving the argument blank returns 0 or 1.

3. `RandomComplex[{z_min, z_max}]` - Returns a complex number inside a rectangle in the complex plane with lower left corner `z_min` and upper right corner `z_max`.

4. `RandomChoice[{a, b, c, ... }]` - Chooses a pseudorandom element from the list `{a, b, c, ... }`.

Now, just because you *ask* for a random number doesn't mean you're going to get one. As Prof. von Neumann reminds us, computers carry out *algorithms*, precise sets of sequential instructions that result in a definite, predetermined result. The results are not random, but in some cases they look random enough that we can use them in place of a truly random sequence where we need one.

A simple example of a pseudorandom number generator is this:

1. Start with the number 1.
2. Divide by 7.
3. Write your answer in scientific notation, with at least seven significant figures in the mantissa, *e.g.*

$$1.428571 \times 10^{-1}$$

4. Choose the the sixth digit in the mantissa (7 in this case). This is your first random number.
5. Divide the result of the first calculation (1.428571×10^{-1}) by 7 again.
6. Again, write your answer in scientific notation to at least seven significant figures (*i.e.* at least one more than the digit you will be choosing to avoid rounding errors), and choose the sixth digit. This is your second random number.
7. Keep doing this over and over again (dividing by seven and harvesting the sixth digit) until you have as many numbers as you need.

Exercise 1: Use this procedure to generate a list of twenty numbers. They will, of course, be integers between zero and nine. Do they look reasonably random to you?

Hint 1: It will be easiest and most consistent if you don't round your mantissa at all, *i.e.* just have Mathematica generate the sequence $1/7^n$ where n goes from one to twenty, display the results in scientific notation to seven or more digits, then select the sixth digit from the mantissa of each element.

Hint 2: You can generate your sequence and then just read off the digits if you want, or, if you don't mind doing a little more work, you can automate the process. In that case you will find the commands `MantissaExponent`

and `RealDigits` useful, along with the list-manipulation skills I have already taught you.

Exercise 2: Plot your numbers using `ListPlot`. This will probably look fairly random to a casual inspection, but looking at them a different way can reveal a bias. Use the `Histogram` command to plot the frequency with which each number occurs. Its argument is your list of numbers, followed by the number of bins you want, *i.e.*

```
Histogram[p, 10]
```

where `p` is the variable that contains your list. Putting the second argument in curly brackets gives you more control over the range, bin width, etc. *e.g.*

```
Histogram[p, {x_min, x_max, bin_width}]
```

Do you think there is a bias here, *i.e.* some digits are under- or overrepresented? If so, can you think of an easy way to test whether the bias is real or accidental?

Of course there is nothing special about the number seven. You could have chosen any number for the division steps, and you could have chosen any digit to harvest. However, if you had harvested the first or second digit your results might have had more bias than you get if you harvest digits of lesser significance. There is an art to these things, after all.

This procedure does a pretty good job of generating random integers, but it has one fatal flaw. It will generate the same sequence every time you run it. If you need a different sequence every time, and it's not out of the question to ask this of a random-number generator, you need to modify it.

Exercise 3: Go back and look at the sequence you generated in Exercise 1 ($1/7^n$), but this time harvest the *first* digit of each mantissa. Display this set of numbers using `ListPlot` and `Histogram`, just as you did in Exercise 2. The bias you see in favor of small values for the leading digit is an example of *Benford's law* [1], and it shows up in far more situations than you might at first expect.

2 Seeding and testing for randomness

I said that you could have chosen anything for the divisor and the harvest digit. The number you started with, 1 in Step #1, is arbitrary as well., and it has a special name. It is called the *seed*. If you picked a different number

to start with each time you ran this procedure you would get a different sequence each time. You could, for example, choose your seed by reading the time of day to the nearest hundredth of a second and taking the last digit. This is, in fact, how a lot of pseudorandom number generators do it. Mathematica is a bit more sophisticated, but the general idea is the same. You can set the seed manually using `SeedRandom`.

Exercise 4: Run `RandomReal` a couple of times, and verify that it gives a different answer each time. Now run the following pair of commands together (one right after the other, in the same cell) a few times.

```
SeedRandom[1]
RandomReal[{0,1}]
```

If you specify the same seed beforehand, you can run `RandomReal` until the cows come home, and it will always generate the same answer. This, if nothing else, ought to convince you that this process is deterministic rather than random, and that's why we call it *pseudorandom*. How you test for randomness is a whole field in and of itself, and it is deeper than we have time to go into here. However, you can do a lot with the simple `ListPlot` and `Histogram` tests we performed earlier. If you ever have need to do this for real, look up the *spectral test*. It's one of the more powerful tests of randomness, and it is fairly easy to code.

3 Monte-Carlo methods

Now that you know how to get pseudorandom numbers, I'm going to show you a use for them. Calculations involving random numbers are common in physics, and they are usually lumped together under the general title of Monte-Carlo methods. Monte-Carlo is a resort town in the very-small European country of Monaco, on the Mediterranean coast, near the border between France and Italy. They do a lot of gambling there, which also involves random numbers, and naming these methods after a European gambling town sounded more refined and sophisticated than calling them "Vegas methods."

There is no single procedure that can be called *the* official Monte-Carlo method. Anything utilizing random numbers counts. For example, you could use the following method to calculate a value for π [2, 3].

1. Draw a square, each side of which is one unit long. Its area will then be one unit squared.

2. Draw a circle inside this square, as shown in Figure 1. Its radius will be half a unit, so its area will be $\pi/4$. Note that this is also the ratio of the area of the circle to the area of the square. If we could measure the areas of both, take that ratio, and multiply it by four, we would have a value for π.

3. Measuring areas is tedious and inexact. Instead, just sprinkle some random points (evenly distributed!) around inside the square, as shown in Figure 1. The fraction that lands inside the circle should be reasonably close to $\pi/4$.

4. In this example we don't even need a computer to do the calculation. In Figure 1 I've generated a hundred points. If you count the number of points that land inside the circle you will get seventy-eight, and the value of π is the ratio of that number to the total, multiplied by four.

$$\pi \approx 4\frac{78}{100} = 3.12$$

This result is within about 1% of the actual value, which is pretty good for how little work we put into it. You can improve the accuracy by choosing more random points, of course, but more importantly you can quantify the uncertainty in your answer using the square-root rule for counting statistics.

Exercise 5: Estimate the uncertainty in the above result. Should we expect the 1% accuracy we got there every time, or did we get lucky? How many points would we have to use to guarantee (or at least have a reasonable expectation of) this level of accuracy every time.

Exercise 6: Write a code for this Monte-Carlo π calculation, and run it with the number of points you obtained in Exercise 5.

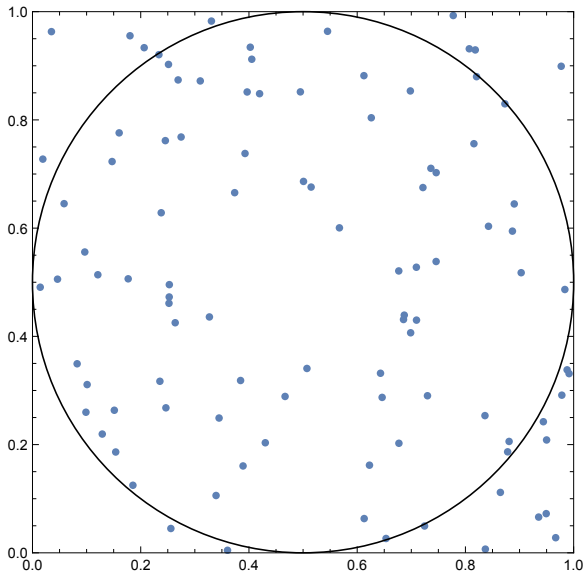

Figure 1: Monte-Carlo method for evaluating π.

References

[1] `https://en.wikipedia.org/wiki/Benford's_law`

[2] Richard W. Hamming, *Methods of Mathematics Applied to Calculus, Probability, and Statistics*, Dover Publications, Inc., Mineola, New York (1985).

[3] Ifan G. Huges and Thomas P. A. Hase, *Measurements and Their Uncertainties: A Practical Guide to Modern Error Analysis*, Oxford University Press Inc., New York (2010).