CmpE 343 Lecture Notes
7: Simulating Random Experiments on a Computer

Ethem Alpaydın

December 4, 2014

1 Introduction

We now discuss how to simulate random experiments in software on a computer. A computer is a deterministic machine so keep in mind that any randomness that comes out of a computer program is always pseudo-random. We consider a set of simple random experiments and see how they can be programmed. These should be taken as simplified examples; more sophisticated methods exist for most of those.

2 Generating Psedo-Random Numbers

Every programming environment has a basic random number generator function that returns a pseudo-random integer between 0 and `maxint` where `maxint` is the maximum integer that can be stored on that system, for example if an integer uses four bytes, `maxint` is $2^{32} - 1$.

The most widely and wildly used method for generating pseudo-random numbers is by a *linear congruential generator* where we have calculate the next random number from the previous number through a recurrence relation:

$$X_{n+1} = (aX_n + c) \text{ m}od\ m \tag{1}$$

X_0 is called the *seed* and for the same seed, the generator always returns the same sequence of random numbers. This may be a useful property when for example debugging the software that calls the random number generator, but in practice for things to look more random, we use a different seed every time—we can for example call a system function that returns the current time and use it as the seed.

a and c are integer constants that are chosen so that the period of the generator is as large as possible (maximum period is m). m is taken as a power of two so that the modulo operator can be implemented easily, for example if m is 2^{32}, we take only the last four bytes of the result (after doing the multiplication and addition in higher precision, for example using 64 bits)[1].

If we generate a random integer in this way and then divide it by `maxint`, we get a floating point random number uniformly distributed between 0.0 and 1.0; let us call the function that does that `rand()`. Below, we use `rand()` to generate random variables from other distributions.

3 Uniform Experiment

Let us say we want to generate a uniform random number in the interval $[A, B]$; we can easily do this as

```
u=rand();
x=(B-A)*u+A;
```

Let us see how we can simulate tossing a fair coin. There are two outcomes, heads and tails, and if the coin is fair, their probability should be equal. So what we should do is divide the range of the uniform `rand()` into two equal-sized parts. For example, we say heads if `rand[]` returns a value between 0.0 and 0.5, and we say tails if `rand()` returns a value between 0.5 and 1.0:

[1] For conditions that a, c and m should satisfy and examples of parameters used in some implementations in the past, see `http://en.wikipedia.org/wiki/Linear_congruential_generator`.

```
u=rand();
if (u<0.5) printf("Heads\n"); else printf("Tails\n");
```

There is nothing special about this; we can also say heads if `u>0.5` and tails otherwise; or we can say heads if `(u>0.1) && (u<0.6)`. What is important is that the range of acceptance of heads (or the proportion of the range of acceptance of heads to the total range returned by `rand()` should be equal to the probability of heads.

Let us see now how we can simulate tossing a fair die. There are six outcomes all having the same probability so we should divide the range of `rand()` into six equal-sized intervals. For example:

```
u=rand();
if (u<1/6) printf("One\n");
else if (u<2/6) printf("Two\n");
else if (u<3/6) printf("Three\n");
else if (u<4/6) printf("Four\n");
else if (u<5/6) printf("Five\n");
else printf("Six\n");
```

4 Bernoulli and Multinoulli Experiments

Remember that in a Bernoulli experiment there are two outcomes of "success" and "failure" and the probability of success is p_0 and the probability of failure is $1 - p_0$. For example, a certain drug is effective with probability 0.8, or a coin is biased and the probability of heads is 0.6. We use the same approach where we divide the range of `rand()` into two parts, of length p_0 and $1 - p_0$:

```
% p0 is the parameter: the success probability
u=rand();
if (u<p0) printf("Success\n"); else printf("Failure\n");
```

Let us see how we can simulate draws *with replacement.* Assume we have a bag with two red balls and four blue balls and we draw three balls with replacement. We have a loop that draws one ball at a time:

```
no_red=2; no_blue=4;
tot=no_red+no_blue;
no_draws=3;
for (i=0;i<no_draws;i++)
   u=rand();
   if (u<no_red/tot) printf("Red\n"); else printf("Blue\n");
}
```

Note that here, because we draw with replacement, the number of red or blue balls, or the total in the bag do not change.

What happens if the bag contains red, blue, and green balls, and again we draw three balls with replacement?

```
no_red=2; no_blue=4; no_green=4;
tot=no_red+no_blue+no_green;
no_draws=3;
for (i=0;i<no_draws;i++) {
   u=rand();
   if (u<no_red/tot) printf("Red\n");
   else if (u<(no_red+no_blue)/tot) printf("Blue\n");
   else printf("Green\n");
}
```

This is a multinoulli draw: We choose red if `u` is between 0 and $2/10$, blue if `u` is between $2/10$ and $6/10$ (of length $4/10$), and green if `u` is between $6/10$ and 1 (of length $4/10$).

5 Binomial and Multinomial Experiments

Remember that in a binomial experiment, we repeat n times the same Bernoulli experiment with success probability p_0 and count the number of successes. For example, let us say we have a coin whose probability of heads is 0.6, we toss it 100 times and we are interested in the number of times heads come up:

```
n=100; p0=0.6;
no_suc=0;
for (i=0;i<n;i++) {
   u=rand();
   if (u<p0) no_suc++;
}
printf("%d successes\n",no_suc);
```

Now let us see a multinomial example. We have a bag of red, blue, and green balls and we draw three balls with replacement (so that it is always the same multinoulli at each draw) and we count the number of reds, blues, and greens seen.

```
no_red=2; no_blue=4; no_green=4;
tot=no_red+no_blue+no_green;
no_draws=3; red_draw=blue_draw=green_draw=0;
for (i=0;i<no_draws;i++) {
   u=rand();
   if (u<no_red/tot) red_draw++;
   else if (u<(no_red+no_blue)/tot) blue_draw++;
   else green_draw++;
}
printf("%d red, %d blue, %d green draws\n",
        red_draw,blue_draw,green_draw);
```

As an aside, let us see how we can simulate draws *without replacement*. This implies that after each draw, the number of balls will decrease by one, so it is not the same Bernoulli (two outcomes, red and blue) repeated independently. What we should do is that after each draw, we need to update the number of balls and so the probabilities change:

```
no_red=2; no_blue=4;
no_draws=2;
for (i=0;i<no_draws;i++)
   tot=no_red+no_blue;
   u=rand();
   if (u<no_red/tot) {
       printf("Red\n"); no_red--;
   }
   else {
       printf("Blue\n"); no_blue--;
   }
}
```

6 Gaussian Experiment

Let us now see how we can draw from a Gaussian distribution with mean μ and variance σ^2. There are more accurate and faster ways of doing this but what we will discuss is one that again uses `rand()`. Remember from the central limit theorem that if we draw independent instances from any distribution, their sum is approximately Gaussian distributed. It is also true if we draw from a uniform distribution too. `rand()` returns a uniform distributed random variable with mean $(1+0)/2 = 1/2$ and variance $(1-0)^2/12 = 1/12$. So if we call `rand()` twelve times and sum them up, the sum is approximately

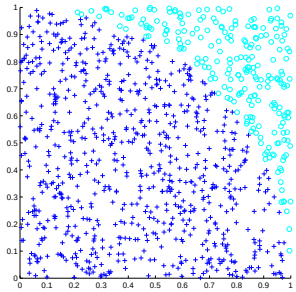

Figure 1: Estimating π with Monte Carlo sampling.

normal with mean $12(1/2) = 6$ and variance $12(1/12) = 1$. Therefore, if we call `rand()` twelve times and sum them up, and then subtract six, we get a standard unit normal Z. Then if we want to get arbitrary $N(\mu, \sigma^2)$, we use the formula $X = \sigma Z + \mu$. Below is the code:

```
% draws from Gaussian with mean mu variance sigma^2
double nrmrand(mu,sigma)
double mu, sigma;
{
     double sum, z, x, rand();
     int i;

     for (sum=0.0,i=0;i<12;i++)
         sum+=rand();
     z=sum-6;
     x=sigma*Z+mu;
     return (x);
}
```

7 Monte Carlo Experiments

We can use random sampling to obtain approximate solutions to some problems; because of the randomness involved, these methods are named after the city where luck plays a large role in its visitors' lives. The following is one simple example.

Let us see how we can estimate π using random sampling. Let us consider the square with sides of length 1, and let us call these sides x and y. If we sample x and y using `rand()`, (x, y) will be a random point uniformly distributed somewhere in this square. Now let us consider the circle whose center is at $(0, 0)$ and radius is 1. For any (x, y), if $x^2 + y^2 < 1$, then (x, y) lies inside the circle. We are considering the upper right quadrant of the circle, which has an area of $\pi/4$. So if we keep repeating random uniform sampling of (x, y) inside the square, we should expect $\pi/4$ of them to fall inside the circle. So if we keep a count, the proportion should converge to $\pi/4$ which we can turn around to estimate π:

```
% estimate \pi using Monte Carlo sampling
for (cnt=i=0;i<N;i++) {
     x=rand(); y=rand();
     if (x*x+y*y<1) cnt++;
}
printf("Estimate for pi is %8.6f\n",4*(cnt/N));
```

We will have a good estimate for π if `rand()` is a good generator and N is large (see Figure 1).