Chapter 1

Random number generators and random processes

If we look around, we notice that many processes are nondeterministic, i.e. we are not certain in their outcome. We are not 100 % certain if the rain will happen tomorrow; the banks do not have certainty about their loan return; sometimes, we do not know if our car will start or not. Our civilization makes the best effort to make life predictable, i.e. we are quite certain that a roof will not leak tomorrow and a grocery store will have fruits for sale. Still, we cannot exclude the element of uncertainty from our calculation. To model such uncertain or random processes, we use random number generators (RNG).

1.1 Statistics and probability introduction

1.1.1 Discrete event probability

Before we begin our discussion of RNGs, we need to set certain definitions. Suppose we record observations of a certain process which generates multiple discrete outcomes or events. Think, for example, about throwing a six-sided die which can produce numbers (outcomes) from 1 to 6. The mathematical and physical definition of the probability (p) of a discrete event 'x' is given by

Probability of event ('x')

$$p_x = \lim_{N_{total}\to\infty} \frac{N_x}{N_{total}} \tag{1.1}$$

where
N_x is the number of registered event 'x'
N_{total} is the total number of all events.

We would have to throw our die many (ideally infinite) times to see what is the probability to get a certain number. This is very time consuming, so most of the time with a finite number of trials, we get only the *estimate* of the probability[1]. Sometimes, we can assign such probabilities if we know something about the process. For example, if we assume that our 6-sided dice is symmetric, there is no reason for one number to be more probable than the other, thus probability of any outcome is $1/6$ for this dice.

1.1.2 Probability density function

While in real life all events are discrete, mathematically it is convenient to work with events which are continuous, i.e. no matter how small a spacing we choose, there is always an event possible next to the other one within this spacing. Think, for example, about the probability of pulling a random real number from the interval from 0 to 1. This interval (as well as any other non zero interval) has an infinite amount of real numbers, and, thus, the probability to pull any particular number is zero.

In this situation, we should talk about probability density of an event 'x', which is calculated by the following method: we split our interval into 'm' equidistant bins, run our random process many times, and calculate

The probability density estimate of an event 'x'

$$p(x) = \lim_{N_{total}\to\infty} \frac{N_{x_b}}{N_{total}} \tag{1.2}$$

where
N_{x_b} is the number of events which land into the same bin as 'x';
N_{total} is the total number of all events.

As you can see from the above definition, if we make number of bins (m) infinite and sum

[1] In some situation, we cannot assign a probability to an outcome at all, at least in the sense of eq. (1.1). For example, we cannot say what is the probability to find life on a planet of Alpha Centauri star, we have not yet done any measurements, i.e. our N_{total} is 0. There are other ways to do it via conditional probabilities, but this is outside the scope of this chapter.

over all bins

$$\sum_{i=1}^{m} N_i/N_{totol} = \int p(x)dx = 1. \tag{1.3}$$

As in the case of probabilities of discrete events, sometimes we can assign the probability density distribution a priori.

1.2 Uniform random distribution

The uniform distribution is a very useful probability distribution. You can see its extensive use in chapter 12 and section 9.6. As the name suggests, the density function of this distribution is uniform, i.e. it is the same everywhere. This means that the probability of pulling a number is the same in a given interval. For convenience, the default interval is set from 0 to 1. If you need a single number in this interval, just execute the MATLAB's built-in `rand` function.

```
>> rand()
ans = 0.8147
```

Your result will be different, since we are dealing with random numbers.

Let's check the uniformity of the MATLAB generator.

```
r=rand(1,N);
hist(r,m);
```

The first command generates `N` random numbers; the second splits our interval into `m` bins, counts how many times a random number gets into the given bin, and plots the histogram (thus the name `hist`), i.e. number of events for every bin. You can see the results shown in the fig. 1.1. It is clear that we hit a given bin roughly the same number of times, i.e. the distribution is uniform. When the number of bins is relatively large in comparison to the number of events, we start to see bin-to-bin counts variation (as in the right panel of fig. 1.1). If we increase the number of random events, this bin-to-bin difference will reduce.

1.3 Random number generators and computers

The word random means that we cannot predict the outcome based on previous information. How can a computer, which is very accurate, precise, and **deterministic** generate random numbers? **It cannot!**

The best we can do is to generate a sequence of *pseudo* random numbers. By "pseudo" we mean that starting from the same initial conditions the computer will generate exactly

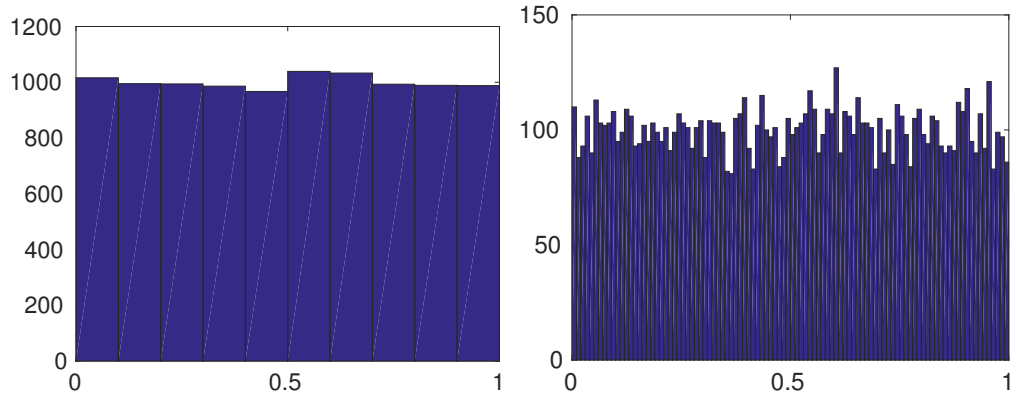

Figure 1.1: Histogram of N=10000 randomly and uniformly distributed events binned into m=10 (left) and m=100 (right) bins.

the same sequence of numbers (*very handy for debugging*). But otherwise, it will look like random numbers and will have **statistical** properties of random numbers. In other words, if we do not know the RNG algorithm, we cannot predict the next generated number based on a known sequence of already produced numbers, while the numbers obey the required probability distribution.

1.3.1 Linear Congruential Generator

A very simple algorithm to generate uniformly distributed integer numbers in 0 to $m-1$ interval is *Linear Congruential Generator* (LCG). It uses the following recursive formula

LCG recursive formula

$$r_{i+1} = (a \times r_i + c) \ \mathbf{mod}\ m \tag{1.4}$$

here
m is the integer modulus;
a is the multiplier, $0 < a < m$;
c is the increment, $0 \leq c < m$;
r_1 is the seed value, $0 \leq r_1 < m$;
mod is the modulus after division by m operation.

All pseudo random generators have a period (see section 1.3.2), and this one is no exception. Once r_i repeats one of the previous values, the sequence will repeat as well.

This LCG can have at most a period of m distinct numbers, since this is how many distinct outcomes the mod m operation has. A bad choice of a, c, m, r_1 will lead to an even shorter period.

Example

The LCG with these parameters $m = 10$, $a = 2$, $c = 1$, $r_1 = 1$ generates only 4 out of 10 possible distinct numbers: $r = [1, 3, 7, 5]$, and then the LCG repeats itself.

We show a possible realization of the LCG algorithm with MATLAB below

Listing 1.1: `lcgrand.m` (available at http://physics.wm.edu/programming_with_MATLAB_book/ch_random_numbers_generators/code/lcgrand.m)

```
function r=lcgrand(Nrows,Ncols, a,c,m, seed)
% Linear Congruential Generator - pseudo random number generator

        r=zeros(Nrows, Ncols);
        r(1)=seed; % this equivalent to r(1,1)=seed;

        for i=2:Nrows*Ncols;
                r(i)= mod( (a*r(i-1)+c), m);
                % notice r(i)  and r(i-1)
                % there is a way to address multidimensional array
                % with only one index
        end
        r=r/(m-1); %normalization to [0,1] interval
end
```

The LCG is fast and simple, but it is a very bad random numbers generator[2]. Sadly, quite a lot of numerical libraries still use it for historical reasons, so be aware. Luckily for us, MATLAB uses a different algorithm by default.

1.3.2 Random number generator period

Even the best pseudo random generators cannot have a period larger than 2^B, where B is the number of all available memory storage bits. This can be shown by the following consideration. Suppose we had a particular bit combination, then we run an RNG and get a new random number. This must somehow modify the computer memory state. Since the memory bit can be only in on or off state, i.e. there are only 2 states available, the total number of the different memory states is 2^B. Sooner or later, the computer will go over all available memory states, as the result of the RNG calculations, and the computer will have the same state as it already had. At this point, the RNG will repeat the sequence of generated numbers.

[2] Do not use the LCG whenever your money or reputation is at stake!

A typical period of an RNG is much smaller than 2^B, since it is unpractical to use all available storage only for RNG needs. After all, we still need to do something useful beyond random number generations, i.e. some other calculations which will require memory too.

While the RNG period can be huge it is not infinite. For example, the default MATLAB's random number generator has the period $2^{19937} - 1$.

Why is it so important? Recall that the Monte Carlo integration method error is $\sim 1/\sqrt{N}$ (see section 9.6.2). This holds true only when N is less than the period of the used random number generator (T). For $N > T$, the Monte Carlo method cannot give uncertainty better than $\sim 1/\sqrt{T}$, since it would sample the same T random numbers over and over. To see it, suppose that someone wants to know an average public opinion. He should choose many random people to ask their opinion. If he starts asking the same two people over and over, it does not improve the estimate of the public opinion. Similarly, the Monte Carlo method will not improve for $N > T$: it will keep doing calculations but there will be no improvement on the result's precision.

1.4 How to check a random generator

As we discussed above, computers alone are unable to generate truly random numbers without additional hardware which uses randomness of some natural process (for example, a radioactive decay or quantum noise in the precision measurements). So, we should be concerned only with RNG statistical properties. If a given RNG generates a pseudo random sequence, which has properties of random numbers then we should be happy to use it.

National Institute of Standards and Technology (NIST) provides software and several guidelines to check RNGs [1], though it is not easy and probably impossible to check all required random number properties.

1.4.1 Simple RNG test with Monte Carlo integration

If only the statistical properties of the RNG are important, we can test them by checking that the integral deviation calculated with the Monte Carlo algorithm drops by $1/\sqrt{N}$ from its true value.

In the following code, we test properties of our LCG with the following coefficients $m = 100$, $a = 2$, $c = 1$, and $r_1 = 1$. For our purposes, we can calculate the numerical integral estimate of any non constant function; here we calculate

$$\int_0^1 \sqrt{1 - x^2}\, dx = \pi/4. \tag{1.5}$$

with the Monte Carlo algorithm and see if the integration error drops by $1/\sqrt{N}$. For comparison, we use the MATLAB built-in algorithm (`rand`) as well. The code of this test is shown below

Listing 1.2: `check_lcgrand.m` (available at http://physics.wm.edu/programming_with_MATLAB_book/ch_random_numbers_generators/code/check_lcgrand.m)

```
% We will calculate the deviation of the Monte Carlo numerical
% integration algorithm from the true integral value
% of the function below
f=@(x) sqrt(1-x.^2);
% integral of above on 0 to 1 interval is pi/4
% since we have shape of a quoter of the circle
trueInteralValue = pi/4;

Np=100; % number of trials
N=floor(logspace(1,6,Np)); % number of random points in each trial

% initialization of error arrays
erand=zeros(1,Np);
elcg =zeros(1,Np);

a = 2; c=1; m=100; seed=1;
for i=1:Np
    % calculate integration error
        erand(i)=abs( sum( f(rand(1,N(i))) )/N(i) - trueInteralValue);
        elcg(i) =abs( sum( f(lcgrand(1,N(i),a,c,m,seed)) )/N(i) -
           trueInteralValue);
end

loglog(N,erand,'o', N, elcg, '+');
set(gca,'fontsize',20);
legend('rand', 'lcg');
xlim([N(1),N(end)]);
xlabel('Number of requested random points');
ylabel('Integration error');
```

To run the test we execute

```
check_lcgrand
```

The results of the two RNG methods comparison are shown in fig. 1.2. We can see that the integration errors keep dropping as we increase the number of points utilized by the Monte Carlo integration when we use the MATLAB's `rand` RNG. Eyeballing the dependence of the

error on N, we can see that the errors drop roughly by an order of magnitude as N increased by two orders of magnitude. This is typical for $1/\sqrt{N}$ behavior. Therefore, MATLAB's generator passes our check. The results with our LCG are quite different: the errors stop decreasing once we reach N around 100; by the time we reach $N \approx 1000$, the errors maintain an almost constant level. The position of the "elbow" for the LCG data roughly (within an order of magnitude) coincides with the period of the LCG. Recall that the period cannot be larger than 100 in this case, since $m = 100$.

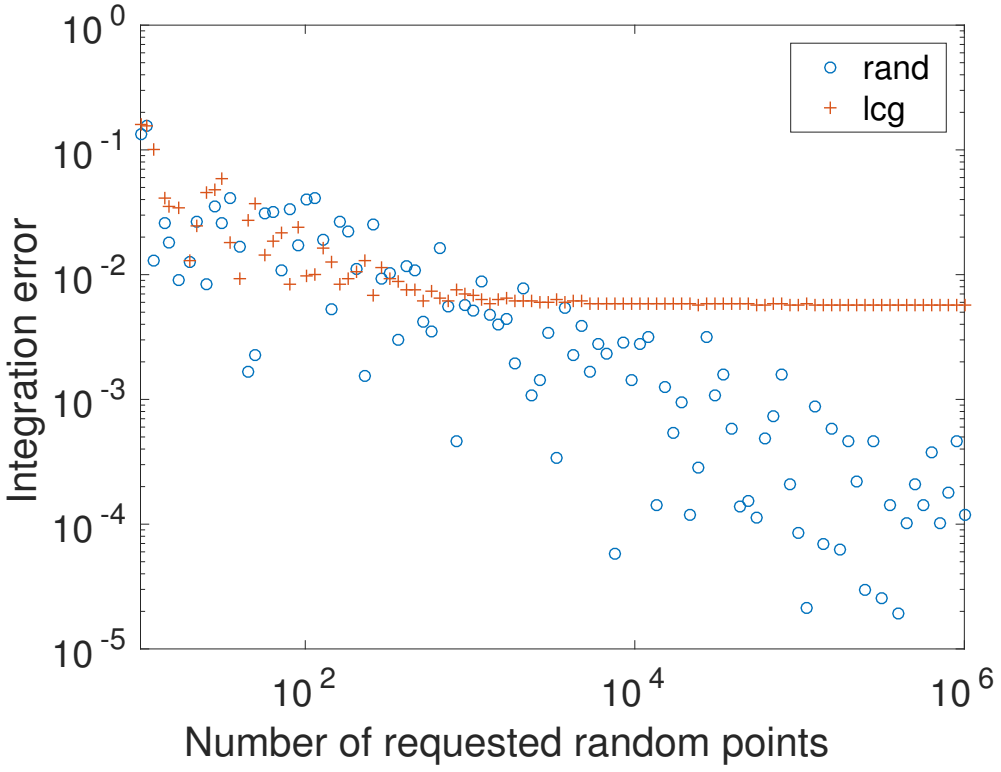

Figure 1.2: The comparison of Monte Carlo integration done with good MATLAB's built-in RNG (circles) and our LCG (crosses) with coefficients $m = 100$, $a = 2$, $c = 1$, and $r_1 = 1$.

1.5 MATLAB's built-in RNGs

In this chapter, we focused our attention on the uniform pseudo random number generator, which is implemented in MATLAB as `rand` function. It generally takes two arguments: the number of rows (`Nrows`) and columns (`Ncols`) in the requested matrix of random numbers. The typical way to call it is as `rand(Nrows, Ncols)`. For example

```
>> rand(2, 3)
ans =
  0.1270    0.6324    0.2785
```

```
  0.9134    0.0975    0.5469
```

MATLAB can also produce random numbers distributed according to the standard normal distribution. Use `randn` for this. Compare the histograms of the normal distribution obtained with `r=randn(10000); hist(r,m)` shown in the fig. 1.3 with the histograms of the uniform distribution in fig. 1.1.

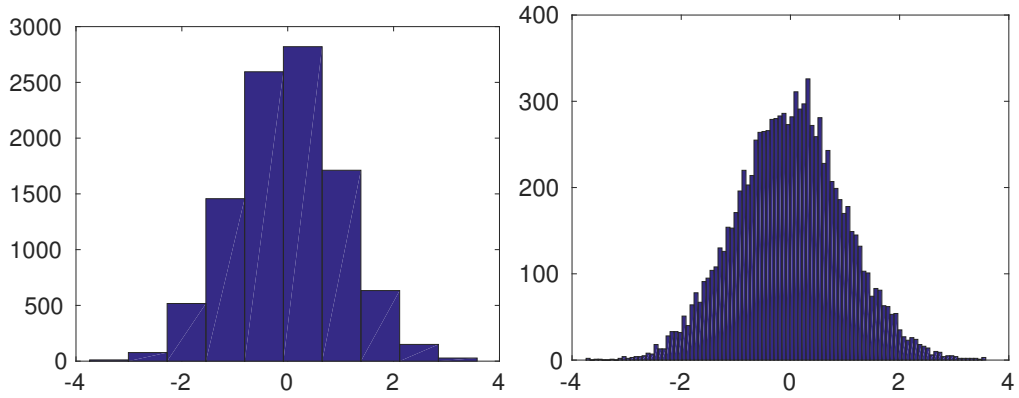

Figure 1.3: Histogram of 10000 randomly and normally distributed events binned into 10 (left) and 100 (right) bins.

If you need to have the same pseudo random number sequence in your calculations, read how to properly use the `rng` function. This function controls the initial "state" of the MATLAB's RNGs.

1.6 Self-Study

Problem 1: Consider the LCG random generator with $a = 11$, $c = 2$, $m = 65535$, and $r_1 = 1$. What is the best case scenario for the length or period of the random sequence of this LCG? Estimate the actual length of the non repeating sequence.

Problem 2: Try to estimate the lower bound of the length of the non repeating sequence for MATLAB's built-in `rand` generator.

Bibliography

[1] A statistical test suite for the validation of random number generators and pseudo random number generators for cryptographic applications. http://csrc.nist.gov/groups/ST/toolkit/rng/documentation_software.html. Accessed: 2016-10-09.