Random Simulations

Jiseok Chae

Department of Mathematical Sciences
KAIST

Week 9

Contents

This week we will see how to generate random numbers in MATLAB, and how random numbers can be used in “mathematical experiments”.

Experiments do not actually prove anything, but...

- Experimental results help us guess the answer; solving a problem with knowing the answer can be easier.
- There are problems where *solving the problem* is proven to be hard; so in some cases experimental results may be the best we can hope.

★ The results shown in this presentation mostly include randomness, so they will be different from your trials.

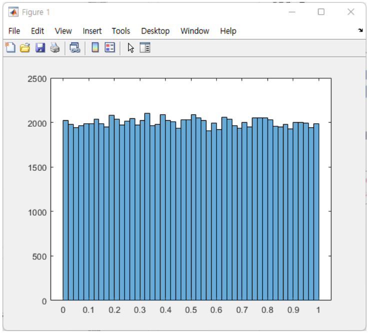
The function `rand` generates a random number, uniformly over the interval $[0, 1]$.

Multiple random numbers can be generated at once.

```
>> rand()

ans =

    0.4077

>> rand(2, 3)

ans =

    0.1435    0.8947    0.1604
    0.2274    0.4823    0.2309
```

Let us verify that the numbers are generated uniformly.
The function `histogram` generates a histogram of input data.

```
>> data = rand(100000, 1);
>> histogram(data)
```



The range of `rand` cannot be customized. To generate a random number uniformly over the interval $[a, b]$, scale by $(b - a)$ and add a.

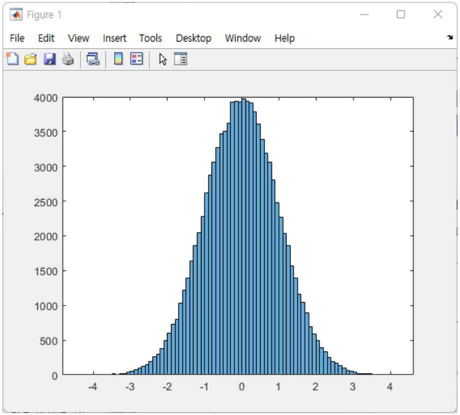
```
>> 2 * rand(2, 4) - 1 % random between -1 and 1

ans =

    0.4072    0.2616   -0.2463    0.0869
   -0.3463    0.4845   -0.7914   -0.9134
```

The function `randn` generates a random number, from a standard normal distribution (mean 0, variance 1).

```
>> randn()

ans =

   -1.2048

>> randn(1, 4)

ans =

    0.7128    0.7555    2.3122   -1.0013
```

Again, we can use `histogram` to verify that the results are from the standard normal distribution.

```
>> histogram(randn(100000, 1))
```



For random integers, we can use the function `randi`. For a positive integer `n`, the command `randi(n)` chooses a random integer bewteen 1 and `n` (both inclusive) with uniform probability.

```
>> randi(6, [3, 4])  % same as rolling 3*4=12 dice

ans =

     6     3     2     3
     2     3     5     1
     4     2     3     3

```

`randi` allows to customize the range. For positive integers `a` and `b`, the command `randi([a, b])` chooses a random integer bewteen `a` and `b` (both inclusive) with uniform probability.

```
>> randi([0, 36], [3, 4])
    % same as spinning a European roulette 12 times

ans =

    36    35     0    19
     2    11    34    29
    36     2    10     1
```

Internally, there is a chaotic mechanism that can generate “random” numbers, but such mechanisms require an initial value to start.
Those initial values are called *seeds*, and controlling the seed can make the results reproducible.

```
>> rand([1, 4])

ans =

    0.4914    0.6993    0.1946    0.7750


>> rand([1, 4])

ans =

    0.7079    0.3651    0.9721    0.0930
```

Internally, there is a chaotic mechanism that can generate “random” numbers, but such mechanisms require an initial value to start.
Those initial values are called *seeds*, and controlling the seed can make the results reproducible.

```
>> rng(109);
>> rand([1, 4])

ans =

    0.6115    0.4914    0.6993    0.1946

>> rng(109);
>> rand([1, 4])

ans =

    0.6115    0.4914    0.6993    0.1946
```

Contents

If a given problem is too complex, we may not be able to compute an exact solution to it.

In those cases, performing experiments (or simulations) over and over and examining how the results come out might be the best stategy.

Hoping that the *law of large numbers* somehow magically helps us, we can infer the true solution by performing multiple random simulations.

Let us try to approximate the volume of a sphere of radius 1.

We generate N random points from the cube $[-1,1]\times[-1,1]\times[-1,1]$.

Then we count the points that lie inside the sphere. Let this number n.

Then intuitively,

$$\frac{n}{N} \approx \frac{\text{volume of the sphere}}{\text{volume of the cube}}$$

but we know that the volume of the cube is 8.

Let us now convert this observation into a MATLAB script...

sphere_volume.m

```
N = 10000;

n = 0;
for i = 1 : N
    x = 2 * rand(3, 1) - 1;
    if x.' * x <= 1
        n = n + 1;
    end
end

V = n / N * 8;
disp(V);
disp(4/3 * pi);
```

Test results:

```
>> sphere_volume
    4.1872

    4.1888

>> sphere_volume
    4.2040

    4.1888
```

The approximation is sometimes good, sometimes bad. Having errors is expected, as random quantities are involved.

Increasing N or averaging the results from multiple trials can reduce error.

Randomization can be applied to linear algebra, and here is a simple but interesting example.

Suppose we have a linear operator $T : \mathbb{R}^n \to \mathbb{R}^n$. Let A be its matrix representation.

We want to compute $\mathrm{trace}(A)$, but we don't know what A (or T) is. The only thing we can do is to get the result of $A\mathbf{x} = T(\mathbf{x})$ for a vector $\mathbf{x}$. We can choose the vector $\mathbf{x}$ freely though.

This may be the case where you are given a function, but the design of that function is too complicated to be interpreted.

Of course, you can reconstruct A by computing $A\mathbf{e}_1$, $A\mathbf{e}_2$, $\ldots$, $A\mathbf{e}_n$ where $\mathbf{e}_i$ are the standard basis vectors of $\mathbb{R}^n$.

However, this requires n matrix-vector multiplications. If n is very large, we better have a more efficient method.

If obtaining a good enough approximation is sufficient, then we can use randomization to accomplish our task faster.

★ If you are not familiar with probability and statistics, you may skip to page 21 and take the statement in that page as granted.

Let $X_1, X_2, \dots, X_n$ be *independent* random variables having standard normal distribution.

Then, we know the followings.

- Their expectation is $\mathbb{E}[X_i] = 0$.
- Their variance is $\mathrm{Var}(X_i) = 1$.
- In particular, because $\mathrm{Var}(X_i) = \mathbb{E}[X_i^2] - (\mathbb{E}[X_i])^2$, it holds that

$$1 = \mathrm{Var}(X_i) = \mathbb{E}[X_i^2] - (\mathbb{E}[X_i])^2 = \mathbb{E}[X_i^2].$$

- By independence, if $i \neq j$, then $\mathbb{E}[X_i X_j] = \mathbb{E}[X_i]\,\mathbb{E}[X_j] = 0$.

Let a_{ij} be the (i,j)-entry of A. Then for any vector $\mathbf{x} = [x_1\ x_2\ \dots\ x_n]^\top$, we know that

$$\mathbf{x}^\top A\mathbf{x} = \sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij}x_ix_j.$$

This is just how matrices and vectors are multiplied, so nothing changes if x_i are replaced by the random variables X_i.

But then, because of the linearity of expectation,

$$\begin{aligned}\mathbb{E}[\mathbf{x}^\top A\mathbf{x}] = \mathbb{E}\left[\sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij}X_iX_j\right] &= \sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij}\,\mathbb{E}[X_iX_j] \\ &= \sum_{i=1}^{n} a_{ii} = \operatorname{trace}(A).\end{aligned}$$

Therefore, we have just showed the following:

> *If we sample an n dimensional vector $\mathtt{x}$ so that each entry is from a standard normal distribution, then $\mathtt{x}^\top A\mathtt{x}$ will be a random sample whose expectation is $\mathrm{trace}(A)$.*

So our strategy is as follows.

- Sample the n dimensional vector $\mathtt{x}$, using the `randn` function.
- Compute $\mathtt{x}^\top A\mathtt{x}$.
- Take the average of the random samples $\mathtt{x}^\top A\mathtt{x}$.

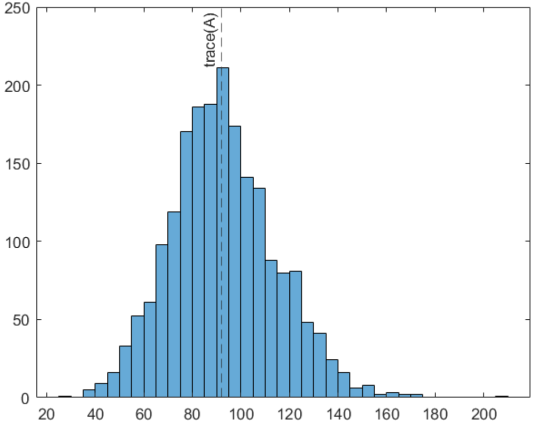
Then the computed average would hopefully be a decent approximation of $\mathrm{trace}(A)$, if the law of large numbers works.

Let us write a MATLAB script that follows our strategy...

random_trace.m

```
n = 200;

A = rand([n, n]); % pretend we don't know this...

iter = 50; % 50 is a large number in statistics... :D
res = 0;
for i = 1:iter
    x = randn([n, 1]);
    est = x' * A * x;
    res = res + est;
end

res = res / iter;
disp(res)
```

The following histogram shows the results from 2000 iterations of approximations done by the script in the previous slide.



Thank you!