3D Random Walk

LYU Liuke

PHYS3061: Lab Report 1

Contents

1 Introduction

1.1 Random Number Generator

The random number generator implemented here is a simple Linear Congruential Generator(LCG), which is generally a recursive mapping within a certain integer domain. for a set of parameters (a,c,m):

$$x_{N+1} = (x_N * a + c) \mod m$$
$$\xi \colon \{1, 2, 3, \ldots, m-1\} \to \{1, 2, 3, \ldots, m-1\}$$

Some special sets of (a,c,m) give seemingly very random and uniform distribution of results in recursive mapping. Based on this feature, we consider this recursive method as a random number generator.

In this experiment (a=1559, c=313, m=13229) are chosen. The main characteristic of these parameters is that they are all primes, in order to try to avoid periodicity behaviors in the random sequance. This special set is chosen by testing different combinations of (a,c,m) in a small parameter space.

1.2 Random Walk

A random walk, in simple terms, is a random sequance representing the accumulative sum of a random variable. In a simple 3D case:

$$\vec{X_N} = \sum_{i=1}^{n} \vec{s_i}$$
$$\text{where} \quad \vec{s} \in V = \{(1,0,0), (-1,0,0), (0,1,0), (0,-1,0), (0,0,1), (0,0,-1)\}$$
$$\text{Assigning} \quad P(\vec{v}) \equiv 1/6 \quad \text{for} \quad \vec{v} \in V$$

Probability analysis give this famous relationship:

$$< |\vec{X_N}|^2 >= \sqrt{N}$$

This relationship can be verified by a Monte Carlo Method, averaging the results over a large number of samples. Alternatively, we can use it to evaluate the randomness of our LCG generator.

2 Code

2.1 Linear Congruential Generator

```
def LCG():
    '''LCG Random Number generator'''
    # seed, a, c, m are all prime numbers, to avoid periodicity
    a = 1559
    c = 313  # for unifrom random float step size
    #c = 647 # for fixed integer step size
```

```
    m = 13229

    t = time.time() #Using current time to set the seed
    state = int(   (t - int(t)) * 10000   ) % m

    logging_dict(locals()) #Recording the parameters in results.log

    '''LCG is a generator object'''
    while True:
        yield state / m
```

LCG is the linear congruential generator with parameters a=1559, c=313 (647), m=13229. The seed of the generator is determined by decimal time. The structure of it is a python generator, which is a sequance of predefined operations. This gives a good structure to continuously generate any length of random numbers and also avoids defining a global state variable. The outputs of this function (after "yield") is normalized to (0,1) by dividing it by m.

2.2 3D Random Walk

```
def RandomWalk_fs(steps, RNG='npRNG', dim=3):
    '''random walk of uniformly distributed float step sizes'''
    if RNG=='LCG':
        walks = np.fromiter(LCG, dtype=float, count= dim * steps).reshape( (steps, dim) )
    else:
        walks = np.random.random( (steps, dim) )
    walks = walks * 2 -1
    pos = np.array(walks).cumsum(axis=0) # cumulative sum of walks in every step
    return pos

def RandomWalk(steps, RNG='npRNG', dim=3):
    '''random walk of a fixed integer valued step size'''
    walks = np.zeros((steps,dim), dtype=int)
    if RNG=='LCG':
        dims = random_choice(dim, size=steps)
        ahead = random_choice([-1,1], size=steps)
    elif RNG=='npRNG':
        dims = np.random.choice(dim, size=steps)
        ahead = np.random.choice([-1,1], size=steps)

    walks[np.arange(steps), dims] = ahead

    pos = np.array(walks).cumsum(axis=0) # cumulative sum of walks in every step
    # returning a 2D array of size (steps, 3)
    return pos
```

As can be seen, I created two versions of random walk function, one with uniform random floating step size and the other with fixed integer step size. They are for different types of purposes, and requires different set of parameters of LCG to get reasonable results. Since function RandomWalk_fs is by nature a mere assembling of three mutually independent 1D random walks, I tend to use the second one in which each step the walker randomly choose a direction among x, y, z and randomly chooses to go 1 step forward or backward. In such case, the three axes are no longer independent.

In the second function, I made use of another function random_choice, which is a mimic of np.random.choice function, but with my LCG as the random number generator.

3 Application

3.1 Pólya's Random Walk Problem

In this part we make use of the function RandomWalk to solve a problem named after Pólya. The problem is:

For a random walk on a n-dimensional lattice, what is the probability of it returning to the origin?

Since this is a lattice random walk, each step shall be a fixed integer step along one of the three normal directions, we shall use function RandomWalk. The script I used for this problem is called returning.py

The idea is simple, we conduct a large number of random walks (10000 particles) of sufficient steps (20000 steps), and record how many of them comes back. We conduct this experiment several times (10 times), and if the resultant probability have a small enough standard deviation, we believe that this result is close to the theoretical prediction. I conducted this experiment for dimension 1 - 5, the results are as follows:

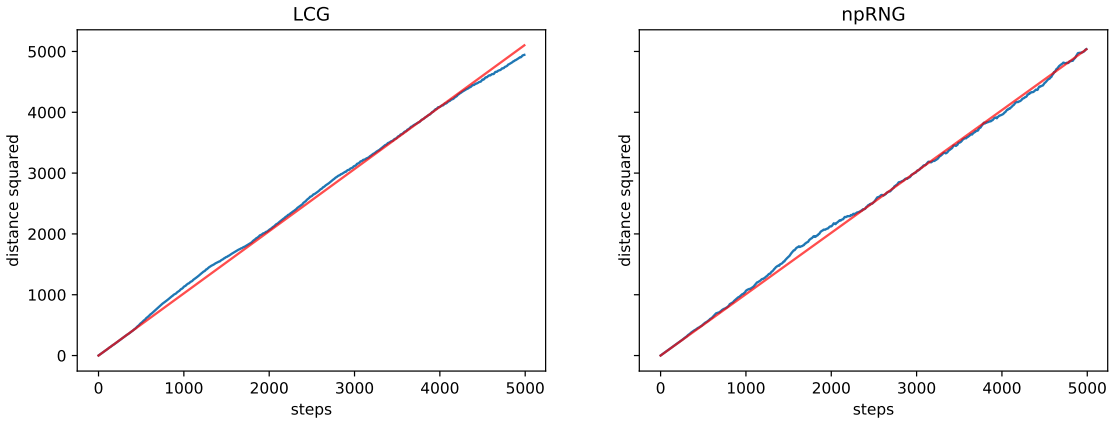
dim	p_exp(d)	std	p(d)
1	0.9941	0.0007	1
2	0.753	0.003	1
3	0.337	0.005	0.340537
4	0.193	0.005	0.193206
5	0.134	0.002	0.135178

As can be seen, other than the result of dimension 2, the other experimental values match the theoretical ones. This is expected, since we can not take an infinite amount of steps for run time consideration, the resultant value will be smaller than the theoretical prediction. In our case, 20000 steps is enough for one dimensional case but not for two dimensional case. While in 3, 4, 5 dimensional cases the results match because the probability converge faster as we increase the dimension.

4 Results and Analysis

Using the mean squared distance relationship, we can test the randomness of our LCG generator. By nature, random walks are sequances that have relatively large fluctuations, so in oreder to conduct a reasonable test, a large number of samples need to be taken to get an average result with relatively small deviations.

Below is a MSD vs N plot of 5000 steps, averaged over 500 particles. Comparing our LCG generator with numpy generator:



The slope of the LCG graph is 1.022 while that of numpy generator is 1.009. At least for random walk with steps equal or below 5000, our generator gives a reasonably good result Although further investigation showed that beyong 6000 steps, the linearity began to break. This could still be expected since 6000 is almost half of m, which limits the period of our generator.

5 Conclusion

To concude, we have implemented a linear congruential random number generator and chosen an effective set of parameters so that it remains random for a quite long period of time. We then made use of the LCG generator to generate random walk sequances. Averaging over many particles, we made a linear plot between MSD and steps of the random walk, which proved that our generator remains quite random below 6000 steps. In addition to this, we also made use of the Random Walk function to help us solve a probability problem.

References

[1] Weisstein,Eric W. "Pólya's Random Walk Constants." From MathWorldld: A Wolfram Web Resource. http://mathworld.wolfram.com/PolyasRandomWalkConstants.html