MATH327: Statistical Physics, Spring 2022

Computer Project — Part 1

Instructions

In this first part of the computer project you will numerically analyze ordinary diffusive behaviour in a one-dimensional random walk. This will allow you to verify your numerical results by comparing them with exact analytic predictions based on the law of large numbers and central limit theorem. The verified numerical methods can then be generalized to consider anomalous diffusion in the second part of the project, where exact analytic predictions will not be available.

There are three exercises below, the first two of which include some background information on pseudo-random numbers and inverse transform sampling. While the exercises mention some syntax specific to Python, you may use a different programming option if you prefer. This demo illustrates all the Python programming tools needed for the project. Even running slowly in the cloud via replit.com, the computing for each exercise should complete in a minute or less.

This part of the project is **due by 23:59 on Thursday, 17 February**. Submit it by file upload on Canvas.[1] **Both** your answers to the questions below and the code that produces your results must be submitted. These can be uploaded as separate files or in a combined file, as you prefer. With the exception of Mathematica `.nb` files, it will be quicker for me to check code submitted in its native format (for example, a `.py` file for Python code or a `.m` file for MATLAB code). Anonymous marking is turned on, and I will aim to return feedback promptly in case this may be helpful when working on the second part of the project.

Exercise 1: Pseudo-random numbers

Background

We have discussed how statistical physics is based on considering systems that involve some element of randomness. Because computer programs are deterministic, it is not possible to use them to generate a truly random sequence of numbers.[2] Instead, computer algorithms generate *pseudo*-random numbers, which are entirely sufficient for our purposes.

A sequence of pseudo-random numbers is defined to be a sequence that *looks* random, in the sense that knowing the first $N-1$ elements in the sequence

[1] By submitting solutions to this assessment you affirm that you have read and understood the Academic Integrity Policy detailed in Appendix L of the Code of Practice on Assessment and have successfully passed the Academic Integrity Tutorial and Quiz. The marks achieved on this assessment remain provisional until they are ratified by the Board of Examiners in June 2022.

[2] New quantum technologies are being developed as a way to produce truly random numbers. This is part of the motivation for large investments in quantum technologies around the world.

does not suffice to predict the Nth element with a high probability of correctness. Equivalently, it takes a very long time for the sequence to start repeating itself—such repetition *will* eventually happen, because computers encode numbers in a finite set of bits, which can represent only a finite set of numbers. For example, 32 bits can represent all integers from 0 through $2^{32} - 1 \sim 10^9$ while 64 bits increase the upper bound to $2^{64} - 1 \sim 10^{19}$. Python uses the Mersenne Twister algorithm as its default pseudo-random number generator (PRNG). This algorithm can provide $2^{19937} - 1 \sim 10^{6000}$ numbers before its sequence repeats.

We can view the absence of true randomness as an advantage rather than a limitation. Deterministic pseudo-random numbers allow our computer programs to be reproducible up to the (very high) precision of the computer. Each exercise below starts by initializing the PRNG with a "seed". Given the same seed, the PRNG will subsequently generate the same sequence of pseudo-random numbers. In Python, as shown in the demo, this initialization is done by calling the function `random.seed(s)`, where s is the seed we specify.

Task

The Python function `random.random()` generates a pseudo-random number u with the uniform probability distribution

$$p(u) = \begin{cases} 1 & \text{for } 0 \le u < 1 \\ 0 & \text{otherwise} \end{cases} .$$

Clearly $\int p(x)\,dx = \int_0^1 dx = 1$, as required. What are the exact mean μ and standard deviation σ of this probability distribution?

`[2 marks]`

Initialize the PRNG with seed $s = 327$. For each of the five $R = 10$, 100, 1000 $10{,}000$ and $100{,}000$, generate a sequence of R pseudo-random numbers u_r distributed according to $p(u)$. (Don't re-initialize the PRNG when changing R, or else these sequences will partially duplicate each other.) Use each sequence to estimate the mean and standard deviation via the law of large numbers,

$$\overline{u}_R = \frac{1}{R}\sum_{r=1}^{R} u_r \qquad\qquad \overline{\sigma}_R \equiv \sqrt{\left(\frac{1}{R}\sum_{r=1}^{R} u_r^2\right) - \overline{u}_R^2}. \tag{1}$$

How do your numerical results compare to your exact analytic predictions above? Rounding to four decimal places should suffice for these comparisons.

`[5 marks]`

In class we saw $\left\langle (\overline{u}_R - \mu)^2 \right\rangle \propto 1/R$ (page 14 of the lecture notes). Let's test this numerically by repeating the above computation of $\overline{u}_R$ another 99 times, ignoring $\overline{\sigma}_R$ for simplicity. Together with the result you reported above, this gives a total of 100 estimates of the random variable $(\overline{u}_R - \mu)^2$, which we can use to approximate the expectation value as

$$\overline{(\overline{u}_R - \mu)^2} \equiv \frac{1}{100} \sum_{i=1}^{100} (\overline{u}_R - \mu)_i^2 \,. \tag{2}$$

Rather than reporting your results as numerical values, plot $R \times \overline{(\overline{u}_R - \mu)^2}$ vs. R and see whether the five points appear approximately constant. If so, is the size of this constant roughly what you would expect?

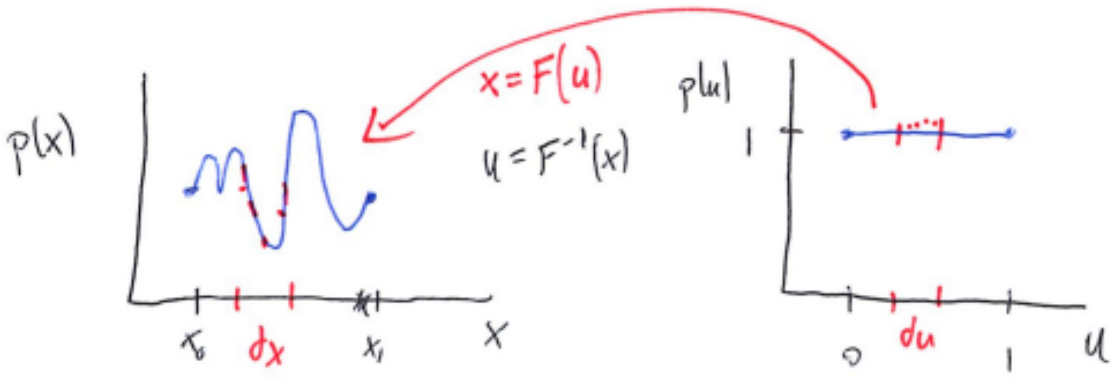
Hints: Include 0 on the y-axis of your plot to maintain a sense of scale. The Matplotlib Python plotting library provides (via its `pyplot` module) the option `xscale('log')` that sets a logarithmic scale for the x-axis, to produce even spacing between these five R.

[8 marks]

Exercise 2: Inverse transform sampling

Background

The uniform distribution is a bit boring. Inverse transform sampling is a technique that allows us to consider more interesting probability distributions, while still generating pseudo-random numbers using the `random.random()` function. The idea is illustrated by the sketch below.



In words, we take our uniformly distributed u_r and act on them with some invertible transformation F to define $x_r = F(u_r)$ that follow the distribution of interest, $p(x)$. We require $p(u)du = p(x)dx$, which allows us to relate $p(x)$ and the transformation $F(u)$:

$$p(x) = p(u)\frac{du}{dx} = p(u)\frac{d}{dx}F^{-1}(x),$$

hence the name “inverse transform sampling”. This relation lets us either engineer an appropriate transformation $F(u)$ to produce a desired distribution $p(x)$, or determine the distribution that results from a given transformation.

Task

Based on the uniformly distributed pseudo-random numbers u generated by `random.random()`, define

$$x = F(u) = \arcsin\left(u - \frac{1}{2}\right). \tag{3}$$

What is the probability distribution $p(x)$ of these random numbers x? What are the minimum and maximum possible values that x can take? What are the resulting exact mean μ and standard deviation σ of $p(x)$?

`[5 marks]`

Reset by initializing the random number generator with seed 327. Now, using the `arcsin` function provided by the Numerical Python (NumPy) library, generate $R = 1{,}000{,}000$ pseudo-random numbers x_r via Eq. 3. Use these to numerically estimate the mean and standard deviation of $p(x)$, analogously to Eq. 1 in the previous exercise. How do your numerical results compare to your exact μ and σ above? Rounding to four decimal places should suffice for this comparison.

`[5 marks]`

In a single plot, compare the histogram of the $1{,}000{,}000$ $\{x_r\}$ to the analytic $p(x)$ you found above. Do your numerical results match your prediction? Roughly 51 bins in the histogram should suffice for this comparison.

Hint: The demo shows how Matplotlib can `plot` a function $p(x)$ on top of a histogram produced using its `hist` routine.

`[5 marks]`

Exercise 3: Random walks

(a) Central limit theorem

Now consider a random walk that consists of N steps, with the length of each step set by a pseudo-random number x_i obtained using Eq. 3. By independently generating R different N-step random walks you can analyze the final positions of the walks,

$$X_r(N) = \sum_{i=1}^{N} x_i \qquad\qquad r = 1, 2, \cdots, R.$$

Based on the central limit theorem, what are the analytic predictions for $\langle X(N) \rangle$ and the diffusion length

$$\ell_2(N) \equiv \sqrt{\langle [X(N)]^2 \rangle - \langle X(N) \rangle^2},$$

each as a function of N? (**Hint:** $\ell_2(N)$ would be called $\Delta X(N)$ in the notes; the new nomenclature is preparation for the second part of the project.)

`[2 marks]`

(b) Fixed number of steps

Reset by initializing the random number generator with seed 327. With fixed $N = 100$, generate R 100-step random walks for each of the four $R = 10$, 100, 1000 and $10{,}000$. Use the resulting X_r to numerically estimate

$$\overline{X(N)}_R \equiv \frac{1}{R} \sum_{r=1}^{R} X_r(N) \qquad\qquad \overline{\ell_2(N)}_R \approx \sqrt{\left(\frac{1}{R} \sum_{r=1}^{R} [X_r(N)]^2 \right) - \overline{X(N)}_R^2}.$$

How do your numerical results compare to your exact analytic predictions for $N = 100$? Rounding to four decimal places should suffice for this comparison.

`[8 marks]`

(c) Diffusion constant

Reset by initializing the random number generator with seed 327. Then fix $R = 10{,}000$ and compute $\overline{\ell_2(N)}_R$ for *every* $N = 1, 2, \cdots, 500$. Rather than reporting results as numerical values, plot $\overline{\ell_2(N)}_R$ vs. N. (**Hint:** You can ignore potential correlations between $\overline{\ell_2(N)}_R$ for different values of N.)

`[4 marks]`

Now fit your numerical results to the function

$$\overline{\ell_2(N)}_R = C + D\sqrt{N}.$$

Include your fit in your plot of $\overline{\ell_2(N)}_R$ vs. N. How do your fit results for C and D compare to the exact analytic predictions from the central limit theorem? (**Hint:** NumPy's `polyfit` routine can handle fits linear in $\sqrt{N}$.)

`[6 marks]`